ABSTRACT

AUTOMATED PART OF SPEECH INDUCTION TO IMPROVE UNDERSTANDING OF THE PARTS OF SPEECH

While the concept of parts of speech is often treated as being well understood, a review of an assortment of applications shows imperfect consistency and little or no justification for the conventions used. I suggest that automated part of speech induction could be used to improve understanding of the parts of speech.

I discuss an established algorithm for part of speech induction and four papers that used that algorithm. I then explain how I modified that algorithm to more clearly reflect the intuitive idea that words that occur in similar contexts should be grouped together.

I discuss my implementation of the resulting algorithm and the results of testing it on the Brown Corpus.

Finally, I discuss what appear to be the main problems with my algorithm and possible basis for solutions to these problems.

Galen Balcom
August 2010
AUTOMATED PART OF SPEECH INDUCTION TO IMPROVE UNDERSTANDING OF THE PARTS OF SPEECH

by
Galen Balcom

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Interdisciplinary Studies in the Division of Graduate Studies California State University, Fresno August 2010
APPROVED

For the Division of Graduate Studies:

We, the undersigned, certify that the thesis of the following student meets the required standards of scholarship, format, and style of the university and the student's graduate degree program for the awarding of the master's degree.

________________________________________
Galen Balcom
Thesis Author

________________________________________
Sean Fulop (Chair)
Linguistics

________________________________________
Brian Agbayani
Linguistics

________________________________________
Shigeko Seki
Computer Science

For the University Graduate Committee:

________________________________________
Dean, Division of Graduate Studies
AUTHORIZATION FOR REPRODUCTION
OF MASTER’S THESIS

I grant permission for the reproduction of this thesis in part or in its entirety without further authorization from me, on the condition that the person or agency requesting reproduction absorbs the cost and provides proper acknowledgment of authorship.

Permission to reproduce this thesis in part or in its entirety must be obtained from me.

Signature of thesis writer: _______________________________
ACKNOWLEDGMENTS

Thank you to Dr. Sean Fulop for his guidance and support in completing this thesis. Thank you also to Dr. Shigeko Seki and Dr. Brian Agbayani for being on my thesis committee.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION.</td>
<td>1</td>
</tr>
<tr>
<td>PARTS OF SPEECH</td>
<td>2</td>
</tr>
<tr>
<td>Overview of Current and Past Use</td>
<td>2</td>
</tr>
<tr>
<td>Motivating Ideas</td>
<td>4</td>
</tr>
<tr>
<td>PAST WORK</td>
<td>6</td>
</tr>
<tr>
<td>Background Concepts</td>
<td>6</td>
</tr>
<tr>
<td>Past Work</td>
<td>9</td>
</tr>
<tr>
<td>MY WORK</td>
<td>13</td>
</tr>
<tr>
<td>Motivation and Background</td>
<td>13</td>
</tr>
<tr>
<td>Algorithm and Implementation</td>
<td>16</td>
</tr>
<tr>
<td>Experiments and Results</td>
<td>19</td>
</tr>
<tr>
<td>DISCUSSION AND CONCLUSIONS</td>
<td>26</td>
</tr>
<tr>
<td>Discussion</td>
<td>26</td>
</tr>
<tr>
<td>Conclusions</td>
<td>28</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>29</td>
</tr>
<tr>
<td>APPENDIX</td>
<td>32</td>
</tr>
</tbody>
</table>
INTRODUCTION

In this paper I first give an overview of the current state of the concept of parts of speech which I argue demonstrates that the concept is not as fully understood as seems to be widely believed. I suggest that using computational linguistics techniques to classify words based on their context in a corpus has the potential to provide an empirical foundation to a system of parts of speech. A corpus is a large collection of text. In the algorithms I discuss, the corpus is represented as a sequence of “words” where words may be defined to include punctuation or other text elements that would not normally be called words. Classifying words in this way is called automatic part of speech induction.

In the following section I explain some mathematical tools needed for the following discussion. I then discuss an algorithm for automated part of speech induction and four previous papers that dealt with this algorithm.

The third section deals with my modification of the algorithm presented in the preceding section. I start from the goal of more directly reflecting the basic idea that words with similar contexts should be grouped together and discuss possible ways of measuring similarity of contexts. I then explain my implementation of the modified algorithm. After discussing implementation I discuss the results obtained when I tested the program on the Brown Corpus.

The final section is a general discussion of the possible use of automated part of speech induction to improve understanding of parts of speech and issues for future research.
PARTS OF SPEECH

Although the parts of speech have a long history and are often treated as being well established fact, there is surprisingly little justification that they fulfill their purpose effectively or even agreement as to what that purpose is.

Overview of Current and Past Use

Modifying the systems proposed by the Stoics, Dionysius of Thrax distinguishes eight parts of speech: noun, verb, participle, article, pronoun, preposition, adverb, and conjunction. The main change made by Dionysius was not separating common nouns and proper nouns (Kneale and Kneale 1962). Although this was over 2,000 years ago, there has been remarkably little change in the system of parts of speech used since then.

Dictionaries tend to provide very little information about the system of parts of speech they use. Neither *Webster’s Third New International Dictionary* (Gove 1976) nor *The Oxford English Dictionary* (Simpson and Weiner 1989) provide a full list of the parts of speech which they assign to words. Under the heading “Functional Labels,” *Webster’s Third New International Dictionary* lists the abbreviations it uses for “The eight traditional parts of speech” namely adjective, adverb, conjunction, interjection, noun, preposition, pronoun, and verb as well as information on functional labels for non-word entries and the position of part of speech or other functional labels in an entry. *The Oxford English Dictionary* describes its formatting and placement of part of speech labels and directs the reader to a table of abbreviation to find the abbreviations used for parts of speech.
While discussions of generative syntax often indicate the system of parts of speech to be used, they generally provide at most a limited justification for that system. Robert Freidin (1992) in *Foundations of Generative Syntax* states it will take the classes noun, verb, adjective, and preposition as primitives and make no attempt to justify them. He then suggests that the further classes: adverb, conjunction, determiner, auxiliary verb, and quantifier can be distinguished. He suggests that phrases like “all the men” provide justification for the distinction of quantifiers from determiners. He notes that the ungrammaticality of phrases such as “*some the men*” complicate the analysis but makes no attempt to resolve these difficulties. Aside from this brief discussion of quantifiers, the additional classes are also presented with no attempt at justification.

*A Grammar of Contemporary English* (Quirk et al. 1972) provides more extensive discussion but does not clearly identify a specific system of parts of speech nor does it provide a clear definition of the purpose of the classifications. It distinguishes the classes noun, adjective, adverb, verb, article, demonstrative, pronoun, preposition, conjunction, and interjection along with subcategories within those classes.

Tagsets for automated part of speech tagging by their nature always specify the system of parts of speech to be used. However, there is little if any discussion of the appropriateness of the system. Tagsets regularly distinguish many more parts of speech than other systems, but the broad divisions used remain variations of the system written about by Dionysius of Thrax. For example, the relatively small CLAWS5 Tagset has 62 tags and divides adjectives into comparative, superlative and unmarked. Adverbs are divided into adverb particles, wh-adverbs and unmarked adverbs. Conjunctions are divided into coordinating conjunctions, subordinating conjunctions, and a separate tag for the conjunction “that.” The
Tagset distinguishes possessive determiners, general determiners, wh-determiners and articles. Nouns are divided into singular, plural, nouns that do not have distinct singular and plural forms, and proper nouns. Pronouns are divided into indefinite pronouns, personal pronouns, wh-pronouns, and reflexive pronouns. Prepositions are divided into “of” and other prepositions. The Tagset distinguishes the verb “be,” the verb “do,” the verb “have,” modal auxiliary verbs, and lexical verbs. For each of these, except modal auxiliary verbs, it further distinguishes the “base form,” -ing form, infinitive, past participle, and -s form. Punctuation is divided into general punctuation, left bracket, right bracket, and quotation mark. The remaining tags do not fall easily into categories. They are cardinal number, existential “there,” interjection, the null tag, ordinal, the possessive morpheme, infinitive marker “to,” items which are not words of the English lexicon, the negative “not” or “n’t,” and alphabetical symbol. No discussion of the selection of the divisions used is provided. (Leech 2010)

Motivating Ideas

The lack of discussion of the basis for systems of parts of speech leaves the question of what an appropriate basis would be. It seems reasonable to assume that words that have the same part of speech should behave the same way with respect to some measure. Possible measures include morphology, context, and semantics. In a system based on morphology, words would be classified according to what affixes they take. Definitions such as “verbs are words that have a past tense” are characteristic of this basis. Systems based on context group words that occur in the same contexts. They can vary based on how context is defined. Using context as a basis allows the possibility of deriving a system of parts of speech directly from a corpus. Context is also probably the most
appropriate basis for a system intended for use in generative syntax although for this purpose it would be best to define context in terms of syntactic structure which would greatly complicate derivation from an unannotated corpus. Definitions such as “nouns denote a person, place, thing or idea” use semantics as a basis.

Bloomfield (1933) argued that a word’s syntactic function in a language, not its meaning, should be used to determine its part of speech. He bases this belief on the variation in grammatical classes across languages and the tendency of apparent correspondence between part of speech and real world meaning to break down for some words. Sapir (1949) concluded that part of speech systems are language specific.

Ji Donghong (1998) suggests the possibility of using syntactic distribution to determine the parts of speech in Chinese where there is insufficient affixing to form a morphological basis for determining parts of speech. He summarizes a discussion of this possibility which shows a very wide range of opinions. Opinions varied from stating that syntactic distribution is the standard and most useful basis for determining parts of speech to arguing that should not be used in this way or that the concept of parts of speech was inherently problematic and perhaps should not be used.
PAST WORK

Background Concepts

Before I begin a discussion of past work on part of speech induction, it will be necessary to explain several concepts relating to statistical natural language processing. These concepts are explained here. I will assume basic knowledge of probability.

A standard method for estimating the probability of a word which is used throughout this discussion is to take the number of occurrences of the word in the corpus and divide it by the total number of word occurrences in the corpus. This method is also used for estimating probabilities of word classes and sequences of words or word classes.

An n-gram model is a model that predicts the probability of the next word based on the preceding n-1 words. An n-gram model where n = 2 is called a bigram model. Thus a bigram model predicts the probability of the next word based on the immediately preceding word. If we have a mapping of words into a set of equivalence classes, we can then construct an n-gram model over equivalence classes. An equivalence class is a group of words that will be treated as being equivalent for the purposes of the model. Such a model is called a class n-gram model. I will often abbreviate equivalence class as class. The papers I discuss in this section primarily use class bigram models. That is models that estimate the probability of the next class based on the immediately preceding class.

A bigram model can be represented as a table of probabilities. Both the rows and the columns of the table would be labeled with the elements of the
vocabulary that is either words, for a standard word bigram model, or classes for a class bigram model. The columns correspond to the first word (or class) in the bigram and the rows correspond to the second word (or class) in the bigram. The entry indicated by the combination of a row and a column gives the probability of that bigram.

For implementation in a computer this pattern is slightly modified. Rather than store two lists of the vocabulary elements as labels, some arbitrary ordering of the vocabulary is adopted allowing the labels of the rows and columns to be omitted as the first row corresponds to the first word, the second row corresponds to the second word and so on and likewise for columns. Another possible change from the basic representation above is to store unnormalized counts rather than probabilities. If needed, the probabilities can be calculated from the counts and the count of the total number of bigrams in the corpus. The table may be stored as a two dimensional array or by using a one dimensional array to simulate a two dimensional array. This can be done by considering the first N elements of the array to be the first column, the second N elements of the array to be the second column, and so on where N is the number of vocabulary elements and thus the number of rows in the table.

To fully specify a class bigram model it is also necessary to specify which words belong to which classes. This can be done with an array. By using a standardized ordering of the words, each element of the array gives the class of the corresponding word.

An n-gram model or class n-gram model can be used to calculate the probability of a word sequence. For a bigram model the probability of a word sequence by finding: the probability that a sequence starts with the first word, the probability of the second word following the first word, the probability of the third
word following the second word, and so on until the end of the sequence. The probabilities would be multiplied together to give the probability of the sequence.

We naturally want some way of evaluating how well a probabilistic model fits the data. We would like to find the most probable model given the data. An expression for this conditional probability can be found by using Bayes’ Theorem.

\[ P(H|D) = \frac{P(D|H)P(H)}{P(D)} \]

In this equation, H represents the model and D represents the data. In the context of language modeling, D represents a corpus and H represents a language model such as an n-gram model or a class n-gram model. This is not very helpful since we do not know P(H), the prior probability of the model, or P(D), the prior probability of the data. Since there is no way to meaningfully estimate these probabilities we cannot evaluate the expression. Instead we use P(D|H), the probability of the data given the model, in place of P(H|D), the probability of the model given the data as a measure of how well the model fits the data. This could be considered as assuming that all models are equally likely a priori although the infinite number of possible models complicates formal use of this interpretation.

P(D|H) is often referred to as the likelihood. Since the number of possible sequences of words with the same length as the corpus is so large, the likelihood is consistently extremely small. To avoid the difficulties of dealing with such incredibly small numbers it is common to use the logarithm of the likelihood which is referred to as the log-likelihood.

Another measure of how well a model fits the data is cross entropy which gives the number of bits needed to specify the corpus using the model. For more
discussion of cross entropy see page 14. Often an equivalent measure called perplexity is used. The perplexity is two to the power of the cross entropy. The perplexity can be interpreted as the number of sides on a fair die with the same degree of uncertainty as the model.

Past Work

The following algorithm for dividing words into classes was developed in (Brown et al. 1992).

Begin with some mapping of words to clusters
Loop until convergence
   For each word, w
      Examine effect of moving w to each cluster
      Move w to the cluster that maximizes the log-likelihood of class bigram model

The algorithm finds a local maximum of the log-likelihood but not necessarily the global maximum. In Brown’s paper this algorithm was used as a secondary step in producing a clustering. The initial clustering for this algorithm was determined by another algorithm that started from assigning each word to its own class and the merging classes until the number of classes reached the desired number. The apparent motivation of this work was to develop class n-gram models as a possible alternative to word based n-gram models for various applications. Having fewer independent parameters and assigning fewer strings zero probability were cited as advantages of class n-gram models over standard
word based n-gram models. The two step algorithm turned out to be computationally impractical for vocabularies over 5,000 words.

Ney, Essen, and Kneser (1994) discussed using Brown’s algorithm for secondary processing on its own. They found that the initial mapping did not have much effect on the final clustering reached but did affect the number of iterations required. The fastest convergence was obtained by assigning each of the n-1 most common words to its own cluster and the rest of the words in the remaining cluster, where n is the number of classes being used. The removal of the need for the many class mergers before beginning this algorithm made the procedure computationally feasible for large vocabularies. They explained that since, at each iteration, the log-likelihood increases (or stays the same if no word is moved) and the likelihood cannot be greater than 1, the algorithm must eventually converge. They also note that the order in which the words are examined may affect the results as each change to the clustering affects future changes. They choose to examine the words in order of decreasing frequency based on the fact that we know more about the distribution of more frequent words. The motivation of this work was similar to that of (Brown et al. 1992).

They tested this algorithm on two corpuses, the English LOB corpus and a German corpus. With each corpus, they used the algorithm to find a set of clusters and created a class bigram model using these clusters. They tried several different numbers of clusters between 30 and 700 clusters. For each bigram model they produced they evaluated the perplexity on a held out test set. They found that the lowest perplexity for the induced class bigram models was obtained with 120 classes for the German corpus and with 350 classes for the English corpus. In both cases the perplexity of the class bigram models was lower than that of a word bigram model. The lowest perplexity induced class bigram model for English had
lower perplexity than a class bigram model in which the classes were determined by linguists. The model of the German corpus with linguist determined classes had lower perplexity than any of the induced models of that corpus. This was attributed to the limited training data from the smaller German corpus.

Martin, Liermann, and Ney (1998) did further work on this algorithm. Like in the previous papers, the aim here was to use the clusters to form class n-gram models. They analyzed the computational complexity of the algorithm, discussed efficient implementation and adapted the algorithm to use the log-likelihood of a class trigram model. They also did experiments with the algorithm on the *Wall Street Journal* corpus. They used three different sized subsets of the corpus for training and a fourth subset for testing. They found that class bigram models with more clusters consistently had lower perplexity than those with fewer clusters. More training data also consistently reduced the perplexity of models evaluated on the test data. All the class bigram models derived from each training set were found to have higher perplexity than the word bigram model derived from that training set.

Clark (2003) uses the algorithm as a base to add a method of incorporating morphological information into the clustering. Unlike the other papers discussed Clark’s aim was not specifically word classes for class n-gram models. Instead, the stated goal was the unsupervised induction of parts of speech without particular restriction on their intended use. He used several variations of the algorithm to produce clusterings for small corpora of seven languages. He used the “conditional entropy evaluation” to evaluate the results. It was not clear from the article how the number of clusters used for each language was determined or how many clusters were used. He also did clustering experiments with the Penn Treebank. He used several variations of the algorithm to produce clusterings with
32, 64, and 128 classes. He evaluated these tests by determining the perplexity of class bigram models using the clusters produced similar to the tests performed in the other papers. In most cases these experiments suggested that the incorporation of morphological and frequency information into the clustering improved results.

The following is an expression for the log-likelihood of a class bigram model from (Martin, Liermann, and Ney 1998). Where \( N(\cdot) \) represents the number of occurrences of the sequence in parentheses in the corpus, \( w \) ranges over the vocabulary and \( g_v \) and \( g_w \) range over the classes.

\[
\sum_w N(w) \log N(w) + \sum_{(g_v,g_w)} N(g_v,g_w) \log \frac{N(g_v,g_w)}{N(g_v)N(g_w)}
\]

(Brown et al. 1992) observes that the first summation can be interpreted as the negative of the entropy of the word unigram model and the second summation can be interpreted as the average mutual information of adjacent classes. Since only the second summation depends on the assignment of classes, assigning classes to maximize this summation is equivalent to assigning classes to maximize the entire expression.
MY WORK

Motivation and Background

When the goal is to produce a class bigram model, it makes sense to evaluate clusterings based on the properties of the resulting class bigram model. Such a method of evaluation makes much less sense when the goal is more generally to find parts of speech. Since my goal here is the latter, a different evaluation method is needed. I will work from the conceptual definition that words of the same part of speech occur in the same context. This leaves the question of how context will be defined and how much alike patterns of occurrence must be to be sufficiently the same. I defined the context of a word to be its signature with respect to the word classes.

The signature of a word or word class is the probability distribution over possible contexts. In this situation the context is the word or word class immediately to the left and the word or word class immediately to the right. I use signatures with respect to word classes which reduces sparsity issues.

The signature of a word or word class can be represented as a table where the row indicates the proceeding word class and the column indicates the following word class. The entry then represents the probability of the word or word class occurring in that context.

This basic idea is generally modified slightly when storing signatures on a computer. These modifications are the same as those for bigram models. For convenience, I will reiterate them here. Rather than store two lists of the vocabulary elements as labels, some arbitrary ordering of the vocabulary is adopted allowing the labels of the rows and columns to be omitted as the first row.
corresponds to the first word, the second row corresponds to the second word and so on and likewise for columns. Another possible change from the basic representation above is to store unnormalized counts rather than probabilities. If needed, the probabilities can be calculated from the counts and the count of the total number of occurrences of the word or word class in question in the corpus.

The table may be stored as a two dimensional array or by using a one dimensional array to simulate a two dimensional array. This can be done by considering the first N elements of the array to be the first column, the second N elements of the array to be the second column, and so on where N is the number of vocabulary elements and thus the number of rows in the table.

In order to explain how to measure the degree of difference between signatures, I must first introduce some concepts from information theory. Entropy is the number of bits required on average to encode the value of a random variable. The entropy of a discrete random variable, X, is written as $H(X)$ and defined by the following equation.

$$H(X) = - \sum_{i=0}^{n} p(x_i) \log (p(x_i))$$

Cross entropy is one measure of the difference between two probability distributions. Cross entropy of two probability distributions, p and q, is the average number of bits required to encode an event from p with an encoding scheme based on q. The cross entropy of two discrete probability distributions, p and q, is written as $H(p, q)$ and defined by the following equation.
\[ H(p, q) = - \sum_x p(x) \log(q(x)) \]

\( H(p,q) \) can never be less than \( H(p) \). Holding \( p \) constant the minimum cross entropy is achieved when \( q = p \). In that case, \( H(p,q) = H(p,p) = H(p) \). Cross entropy is not symmetric, that is, \( H(p,q) \) may be different from \( H(q,p) \). If \( q(x) = 0 \) for some \( x \) for which \( p(x) > 0 \), \( H(p,q) \) becomes undefined.

Kullback-Leibler divergence is another measure of the difference between two probability distributions. While related to the cross entropy, the Kullback-Leibler divergence of two identical probability distributions is always zero regardless of the entropy of the distribution. The Kullback-Leibler divergence of two discrete probability distributions, \( p \) and \( q \), is written as \( D_{KL}(p||q) \) and is defined by the following equation.

\[ D_{KL}(p||q) = \sum_x p(x) \log \left( \frac{p(x)}{q(x)} \right) \]

Like cross entropy, Kullback-Leibler divergence is not symmetric and may become undefined when dealing with events assigned zero probability by a model.

Jensen-Shannon divergence is another measure of the difference between two probability distributions. It is based on Kullback-Leibler divergence but is symmetric and defined for any pair of probability distributions. The Jensen-Shannon divergence of two probability distributions, \( p \) and \( q \), is written as \( JSD(p||q) \) and is defined in terms of the Kullback-Leibler divergence in the following equation.
\[ JSD(p||q) = \frac{1}{2} D_{KL}(p||m) + \frac{1}{2} D_{KL}(q||m) \]

Where \( m = 1/2 (p+q) \).

**Algorithm and Implementation**

I selected the Jensen-Shannon divergence as the measure of difference between the signature of a word and the signature of a class. I selected the Jensen-Shannon divergence because it is symmetric and handles zero probabilities smoothly. This second property is especially important for this application as zero probabilities occur somewhat frequently when frequencies are used directly as probabilities. Using the Jensen-Shannon divergence avoids the need to apply a smoothing algorithm to avoid zero probabilities. The resulting algorithm can be expressed as follows.

Start with some mapping of words to clusters
Loop until convergence or maximum iterations
   For each word, \( w \)
      For each class, \( c \)
         Calculate the JS divergence between \( \text{sig}(w) \) and \( \text{sig}(c) \)
         Move \( w \) to the class with the lowest divergence (don’t move it if it is already there)

I modified code used for the experiments in (Clark 2003) to implement the modified algorithm. I removed the portions of Clark’s code that dealt with the use of morphological and frequency information. The resulting program (see Appendix) consists of seven files: clusters.cpp, clusters.h, matrix.cpp, matrix.h,
neyessen.cpp, simplecorpus.cpp, and simplecorpus.h. Clusters.cpp and clusters.h define the class “Clusters” which implements the clustering algorithm and stores the resulting clusters. Matrix.cpp and matrix.h define classes that implement matrices of integers or of floating point values. Neyessen.cpp contains the main() function which handles the command line input and calls functions from the other files in order to read the corpus and produce a clustering. Simplecorpus.cpp and simplecorpus.h define the class “SimpleCorpusOne” which handles reading and storing the corpus. I will focus on the classes “Clusters” and “SimpleCorpusOne”.

When a corpus is read into a SimpleCorpusOne object, each word type is assigned an integer in order of its first occurrence in the corpus. The corpus is then represented as a sequence of these integers which is stored in a vector (a container class from the C++ Standard Template Library that implements an array like data structure). The program also stores the number of tokens in the corpus, the number of types in the corpus, and data structures to allow translating words from strings into the assigned integers and back.

The Clusters class implements the clustering algorithm. One object of this class is created during a run of the program. The constructor for the Clusters class an integer representing the number of clusters to be produced, a SimpleCorpusOne object representing the corpus to be used and a Boolean indicating whether to randomize the initial mapping of words to clusters or assign each of the n-1 most common words to its own cluster and the rest of the words to the remaining cluster where n is the number of clusters. The corpus and the number of clusters are both stored as members of the Clusters object. The mapping of words to clusters is stored as a vector of in which the index indicates the word using the same assignment of words to numbers as used in the SimpleCorpusOne object and the entry indicates the cluster to which that word is mapped. The constructor
initializes this array according to the method specified. The constructor also creates a vector that lists the words (represented as integers) in order of decreasing frequency. The location of the first occurrence of each word is stored in the vector “first” by considering the index to indicate the word according to the correspondence between words and integers used throughout the program and the entry to indicate the position where position zero is the first token in the corpus. The vector, “next,” is created to store the location of the next occurrence of the word at each position in the corpus. Together these two vectors allow finding all occurrences of any given word in the corpus. Signatures of all words and classes are computed and stored. The signatures are stored as MatrixInt objects which are defined in matrix.cpp and matrix.h to implement the method of storing a table as a one dimensional array discussed earlier. In order to allow the first and final tokens to contribute to the signatures of their respective types, the edges of the corpus are treated as belonging to a special class that cannot have other members. This class is numbered numberClasses while the normal classes are numbered zero through numberClasses - 1.

The implementation of the clustering algorithm starts with the function clusterNeyEssen. This function repeatedly calls the function reclusterNeyEssen until either reclusterNeyEssen returns zero indicating that no changes were made or the maximum number of iterations stored in the global variable MAX_ITERATIONS is reached. MAX_ITERATIONS has a default value of 20 or can be set from the command line.

The function reclusterNeyEssen loops through all the word types in order of decreasing frequency. It calls the function bestCluster for each word with frequency greater than the frequency cutoff stored in the global variable FREQ_CUTOFF which has a default value of -1 (allowing all words to be
processed) or can be set from the command line. The function `reclusterNeyEssen` returns the number of words that were moved by the function `bestCluster`.

The function `bestCluster` takes an integer representing a word as an argument. It loops through the clusters to find the cluster whose signature that has the lowest divergence from the word’s signature. It calls the function `divergence` to compute the divergence between signatures. If the word is not currently in the cluster with the lowest divergence it is moved using the function `moveWord`.

The function `divergence` takes two `MatrixInt` objects representing the unnormalized signatures and two integers representing the totals needed for normalization. The matrices must have the same dimensions. The function computes the Jensen-Shannon divergence between the two signatures by looping through the indices of the matrices, computing the value of the normalized signatures at that index and using these normalized values to compute the elements of the summation which depend on them.

The function `moveWord` takes three integers representing the word to be moved the cluster it is to be moved from and the cluster it is to be moved into. The function changes the entry for the word in `classVector` to the new cluster and updates the number of occurrences of both clusters as well as all signatures that were changed by the reassignment of the word’s class. The updating of signatures requires looping through all occurrences of the word being moved.

**Experiments and Results**

In order to test my program I used the Brown Corpus. I preprocessed the corpus by converting all letters to lowercase, inserting spaces to separate punctuation from words, replaced clause ending punctuation (.;:!??) with “PUNCT”
and removed all other punctuation. I also formatted the corpus to have one word per line as this format is required by the program.

I ran the program to on the processed corpus to produce clusterings with different numbers of classes. The number of classes ranged from 20 to 80. I used the default values for the other parameters. That is a maximum of 20 iterations, no frequency cutoff, and deterministic initial clustering.

Finding an appropriate basis for evaluation of the results is somewhat problematic. An evaluation of how well the induced classes correspond to traditional parts of speech appears inconsistent with the aim of improving on the traditional parts of speech as well as being somewhat subjective. Using an objective statistical measure such as the perplexity of a class bigram model leaves the question of whether the statistical measure reflects the desired characteristics for the clustering. It seems reasonable to expect that methods designed based on a particular statistical measure would appear to be the best according to that measure. I will evaluate the results from a linguist’s perspective using the traditional ideas of parts of speech despite the problems with this approach noted above.

My results were harder to make sense of than I hoped. In some cases, more classes led to grouping words together that had been separated when fewer classes were used. For example, with 30 classes, most numbers written as numerals were assigned to one class and most numbers written with letters were assigned to another class. With 40 classes, numbers written in either way tended to be assigned the same class.

With 70 or 80 clusters, the words “a,” “an,” and “the” were assigned to three different clusters and the words “and” and “or” were assigned to different
clusters. From these observations I conclude that 70 clusters is too many clusters at least for this methodology and corpus.

I performed a more in depth analysis of the results with 20 clusters. Cluster 0 included various determiners including “the” and possessive pronouns among the oddities observed in this cluster were the names of many months.

Cluster 1 contains the punctuation stand in “PUNCT,” complementizers, and some verbs.

Cluster 2 contains the word “of,” verbs including many –ing forms, and the words “between, whose, and plus.

Cluster 3 includes the conjunctions “and,” “or,” and “nor” but is otherwise difficult to interpret.

Cluster 4 contains the words “to,” “not,” “also,” and “still” as well as assorted adverbs.

Cluster 5 contains the determiners “a” and “an,” the words “more” and “no,” and adverbs of degree.

Cluster 6 contains mostly prepositions.

Cluster 7 contains the words “that” and “all” as well as wh-words and names.

Cluster 8 contains verbs including most form of to be.

Cluster 9 is also primarily verbs. It includes the word “am” and various modals and contracted forms.

Cluster 10 contains pronouns and names. It includes the subject forms of personal pronouns.

Cluster 11 contains verbs and prepositions.

Cluster 12 contains pronouns, place names, and common nouns. It includes the object and reflexive forms of personal pronouns.
Cluster 13 contains prepositions and verbs including –ing forms.
Cluster 14 contains subordinating conjunctions and adverbs.
Cluster 15 contains adjectives and numbers written in letters.
Cluster 16 contains prepositions (many of which are also used as particles) and verbs in past tense/participle forms.
Cluster 17 contains verbs without morphological marking.
Cluster 18 contains numbers written as numerals, assorted individual letters, and abbreviations.
Cluster 19 contains common nouns.

The following lists the five most common words in each cluster for the results with 20 clusters.

0  the, his, this, her, their
1  PUNCT, before, where, until, ago
2  of, between, whose, including, include
3  and, or, t, than, nor
4  to, not, also, still, never
5  a, an, more, no, much
6  in, on, at, over, through
7  that, which, all, what, how
8  is, was, are, were, came
9  had, would, has, will, said
10  he, i, they, you, she
11  for, about, like, after, since
12  it, there, him, them, me
I will use a list of 10 parts of speech taken from (Quirk et al. 1972) to guide some additional examination of my results with 20 clusters. These parts of speech are: noun, adjective, adverb, verb, article, demonstrative, pronoun, preposition, conjunction, and interjection.

Nouns were separated into three clusters. One of these clusters, cluster 19, consists almost entirely of common nouns. In the other two clusters, clusters 10 and 12, contain pronouns as well as nouns. The nouns in these clusters were mainly proper nouns. The personal pronouns were divided into subject forms in cluster 10 and object and reflexive forms in cluster 12. The grouping of proper nouns with pronouns suggests the question of whether it is possible to distinguish them from each other by context alone.

Verbs were assigned to a variety of clusters. The clusters 2, 8, 9, 11, 13, 16, and 17 all had noticeable numbers of verbs. The analysis is further complicated by the prevalence of ambiguity between verbs and nouns. Most verbs in “base form” were assigned to cluster 17. Modal auxiliaries were assigned to cluster 9 along with the word am and forms of have and do. Most forms of the word be were assigned to cluster 8. Many of the clusters containing verbs also contained prepositions.
Adjectives were fairly consistently assigned to cluster 15. Numbers written in letters are also assigned to this cluster.

Adverbs were separated into the three clusters: 4, 5, and 14. Cluster 5 contained adverbs of degree as well as some determiners. The fact that the words “more” and “much” (which are in this cluster) can occur both as determiners and as adverbs of degree may help explain this grouping. Cluster 5 also contains the numbers “twenty,” “thirty,” “forty,” “fifty,” “sixty,” “seventy,” “eighty,” and “ninety.” The separation of these numbers from other numbers written in letters is most likely an undesired result of the preprocessing which treats all punctuation (including dashes) as separating consecutive words. Thus, a number such as “thirty-four” would be interpreted as two words. Under this interpretation, multiples of 10 are frequently followed by numbers less than 10 making their context different from that of other numbers. Besides adverbs, cluster 4 also includes the words “to” and “not.” Many adverbs dealing with frequency are included in cluster 4, but I have not found a broad pattern separating the adverbs in cluster 4 and those in cluster 14. Cluster 14 contains subordinating conjunctions in addition to adverbs.

Articles and other determiners are assigned to the clusters 0 and 5. Cluster 0 contains most of the determiners including the article “the.” It also contains a variety of other words including the names of most months. Cluster 5 contains the articles “a” and “an” as well as the words “more,” “no,” and “much” which can sometimes function as determiners. Cluster 5 consists mostly of adverbs.

The demonstratives except “that” were assigned to cluster 0 with other types of determiners. “That” was assigned to cluster 7 presumably due to being used often as a complementizer or relative pronoun rather than a demonstrative.
Most pronouns were assigned to clusters 10 and 12 along with proper nouns. Possessive pronouns were grouped with determiners in cluster 0.

Most prepositions were assigned to clusters 6, 11, 13, and 16. Notable exceptions are the words “to” and “of” which were assigned to cluster 4 and cluster 2, respectively. The separation of “to” from other prepositions is not particularly surprising as it is also used as the infinitive marker. It is less clear why “of” would be different. In general the division of prepositions into different clusters does not appear to reflect any normally recognized subcategories of prepositions. This is best illustrated by the separation of some opposing pairs such as “in” and “out” into different clusters.

The conjunctions “and,” “or,” and “nor” were assigned to cluster 3 while most other conjunctions were assigned to cluster 14.

There are too few interjections in the corpus for meaningful consideration of how they were clustered.
DISCUSSION AND CONCLUSIONS

Discussion

The theoretical desirability of automated part of speech induction stems from the need for an objective, data based basis for parts of speech. The induction method effectively works as an operational definition of the concept of parts of speech. While the appropriateness of any such definition is subject to question, use of clear operational definitions and examination of the consequences of those definitions could reasonably be expected to improve the understanding of the concept of parts of speech. For example, in my experiments I observed that proper nouns and pronouns were clustered together. Initial consideration supported the idea that they generally occur in the same contexts. This suggests the hypothesis that it is not possible to distinguish pronouns from proper nouns on the basis of context alone. This hypothesis could be tested using other context based part of speech induction algorithms. If the hypothesis were true, it would mean that either pronouns and proper nouns were the same part of speech or parts of speech cannot be determined by context alone.

The method presented in this paper has the advantage of being based on what seems to be a reasonable conceptual definition of the parts of speech. However, the method makes two problematic assumptions. These assumptions are that only a word’s immediately neighboring words provide relevant context, and that a word can have only one part of speech. The second assumption completely avoids the issue of ambiguity. These implicit assumptions are common in part of speech induction systems (Brown et al. 1992, Ney, Essen, and Kneser 1994, Martin, Liermann, and Ney 1998, Clark 2003).
Some work does address the issue of ambiguity including Clark (2000) and Schutze (1995). These two papers describe very different clustering procedures. The procedure described by Clark is somewhat similar to mine and the handling of ambiguous words could probably be adapted to work with my clustering procedure.

**CLAWS5 Tagset**

Some aspects of the CLAWS5 Tagset are less than ideal. The word “that” is given a single tag despite it having multiple distinctive usages including complementizer, relative pronoun, and determiner and pronoun uses paralleling “this”. This problem is not solved by the method of clustering developed in this article as this method of clustering cannot assign a word to multiple clusters.

Another related problem is that the word “that” is the only word given that tag thus ignores the fact that some usages of “that” exhibit the same syntactic behavior as other words. This is in some sense corrected by my clustering method as “that” tends to be clustered with other words. However, the resulting cluster is often difficult to interpret. It is doubtful that clustering “that” with other words is useful without addressing ambiguity.

Another ambiguity related problem is the failure to distinguish main verbs from auxiliaries. CLAWS5 separates forms of the verbs “be,” “do,” and “have” from other verbs and from each other but does not classify occurrences of these verbs according to whether they are the main verb of the sentence or an auxiliary verb. The clustering algorithm I have described does not solve this problem as it does not allow a word to belong to multiple clusters.
Conclusions

Automatic part of speech induction shows promise as a way of improving the understanding of the concept of parts of speech by providing a more empirical and objective determination of part of speech divisions. However, the method described in this paper is not satisfactory due to failing to account for ambiguous words and syntactic structures.

The main directions for future research appear to be introducing a method for accommodating ambiguous words and modifying the definition of context to account for syntactic structure. I have not found a clear basis for work on a more appropriate definition of context. The simplest step toward these goals is to adapt a method of handling ambiguous words such as the one described in (Clark 2000) for use with the clustering algorithm used here.
REFERENCES
REFERENCES


APPENDIX
This appendix contains an implementation of the algorithm discussed in this thesis. Much of the code originates from (Clark 2003). I extensively modified the file clusters.cpp in order to have it implement the new algorithm. I also made changes to the other files but these changes were generally superficial.
//clusters.cpp
//This implements the clustering algorithm.

#include "clusters.h"
#include <algorithm>
#include <utility>
#include <iostream>
#include <math.h>
#include <assert.h>

//note: these defaults can be overwritten from the command line
int FREQ_CUTOFF = -1;
int MAX_ITERATIONS = 20;

Clusters:::
Clusters(int numberClasses_,
         const SimpleCorpusOne & corpus_,
         bool randomised)
   :
   numberClasses(numberClasses_),
   numberTypes(corpus_.numberTypes),
   numberTokens(corpus_.numberTokens),
   data(corpus_.data),
   corpus(corpus_),
   clusterBigrams(numberClasses_+1,numberClasses_+1)
{
   classVector.resize(numberTypes);
   counts.resize(numberTypes);
   sortedWords.resize(numberTypes);
   first.resize(numberTypes);
   clusterUnigrams.resize(numberClasses);
   next.resize(numberTokens);

   wordSignatures.resize(numberTypes);
   for(int i = 0; i < numberTypes; i++)
      wordSignatures[i] = new MatrixInt(numberClasses+1, numberClasses+1);
   if(!wordSignatures[i])
      cerr << "could not allocate memory for word signature "
      << i << endl;
exit(1);
}
}

clusterSignatures.resize(numberClasses);
for(int i = 0; i < numberClasses; i++){
    clusterSignatures[i] = new MatrixInt(numberClasses+1, numberClasses+1);
    if(!clusterSignatures[i]){
        cerr << "could not allocate memory for cluster signature " << i << endl;
        exit(1);
    }
}

for (int i = 0; i < numberTokens; i++)
    next[i] = numberTokens;
for (int w = 0; w < numberTypes; w++){
    counts[w]=0;
    classVector[w] = numberClasses - 1;
}

// count occurrences of each word type
for (int i = 0; i < numberTokens; i++)
    counts[data[i]]++;  

// sort word types by decreasing frequency
vector< pair<int,int> > countsTable(numberTypes);
for (int i = 0; i < numberTypes; i++){
    countsTable[i] = pair<int,int>(counts[i],i);
}
cerr << "Sorting words" << endl;
sort(countsTable.begin(),countsTable.end());
//initialize elements of first to -1 and
//transfer sorted words from countsTable into sortedWords
for (int i = 0; i < numberTypes; i++){
    first[i] = -1;
    sortedWords[i] = countsTable[numberTypes - 1 - i].second;
}

//initialize clustering
if (randomised)
{
    for (int i = 0; i < numberTypes; i++)
    {
        first[i] = rand() % numberClasses;
        sortedWords[i] = countsTable[first[i]].second;
    }
}
if (counts[i] > FREQ_CUTOFF) {
    int rc = (int) (1.0 * numberClasses * rand() / (RAND_MAX + 1.0));
    classVector[i] = rc;
}
}
}
else {
    for (int i = 0; i < numberClasses - 1; i++) {
        classVector[sortedWords[i]] = i;
    }
}
}

vector<int> last(numberTypes, 0);
int leftEnv, rightEnv;

cerr << "Indexing data" << endl;
for (int i = 0; i < numberTokens; i++) {
    int w = data[i];
    assert(w >= 0 && w < numberTypes);
    if (i > 0)
        leftEnv = classVector[data[i - 1]];
    else
        leftEnv = numberClasses;
    if (i < numberTokens - 1)
        rightEnv = classVector[data[i + 1]];
    else
        rightEnv = numberClasses;
    if (first[w] == -1) {
        first[w] = i;
        last[w] = i;
    } else {
        next[last[w]] = i;
        last[w] = i;
    }
}
int c = classVector[w];
assert(c >= 0 && c < numberClasses);

(*wordSignatures[w])(leftEnv,rightEnv)++;
(*clusterSignatures[c])(leftEnv,rightEnv)++;
clusterUnigrams[c]++;
}
cerr << "Finished indexing " << endl;
}

// dump this clustering to cout as: word cluster p(word| cluster)
void Clusters::
dump() const
{
    vector<double> clusterCounts(numberClasses,0.0);
    for (int i = 0; i < numberTypes; i++)
        clusterCounts[classVector[i]] += counts[i];
    for (int i = 0; i < numberClasses; i++)
        cerr << i << " " << clusterCounts[i] << endl;
    for (int i = 0; i < numberTypes; i++){
        int w = sortedWords[i];
        int cl = classVector[w];
        if (clusterCounts[cl] <= 0)
        {
            cerr << "error: clusterCounts[" << cl << "] <= 0" << endl;
            exit(EXIT_FAILURE);
        }
        cout << *(corpus.wordArray[w]) << " "
             << cl << " "
             << double(counts[w])/clusterCounts[cl] << endl;
    }
}

void Clusters::
clusterNeyEssen()
{
    int i = 0;
int c = 0;
while ((c = reclusterNeyEssen()) > 0 && i < MAX_ITERATIONS){
i++;
cerr << "finished iter " << i << "," changed " << c << endl;
}

Clusters::
reclusterNeyEssen()
{
int changes = 0; //counts number of word types moved

//examine words in order of decreasing frequency,
//for each word with frequency greater than cutoff, find best clusters
for (int i = 0; i < numberTypes; i++)
{
int w = sortedWords[i];
if (counts[w] > FREQ_CUTOFF)
{
if (bestCluster(w)){
changes++;
}
}
}
return changes;
}

// move word w to the best cluster
// return true if w is moved or false if w was already in the best cluster
bool Clusters::
bestCluster(int w)
{
//cerr << "in function bestCluster: starting to work on word " << w << endl;
double score = infinity();

int best = classVector[w];

for (int i = 0; i < numberClasses; i++)
{
assert(counts[w] != 0);
assert(clusterUnigrams[i] != 0);
assert(classVector[w] != 0);
assert(clusterUnigrams[i] != 0);
double newDivergence = divergence(*wordSignatures[w],
*clusterSignatures[i], counts[w], clusterUnigrams[i]);
if (newDivergence < score) {
    score = newDivergence;
    best = i;
}
}

int old = classVector[w];
if (old != best) {
    moveWord(w, old, best);
    return true;
}
else {
    //cerr << "leaving " << w << " in class " << old << endl;
    return false;
}
}

//used in calculating divergence
double kl(double p, double q) {
    if (p > 0)
        return p * log(p / q);
    else
        return 0.0;
}

//calculates Jensen-Shannon divergence of signatures
//each signature is represented as a MatrixInt
//returns D(P||(P+Q)/2) + D(Q||(P+Q)/2)
//where D(X||Y) represents the Kulback-Leibler divergence
double Clusters::divergence(const MatrixInt & P, const MatrixInt & Q, int sumP, int sumQ) {
    assert(P.dim1() == Q.dim1());
    assert(P.dim2() == Q.dim2());
    int d1 = P.dim1();
    int d2 = P.dim2();
    double result = 0.0;
for(int i = 0; i < d1; i++){
    for(int j = 0; j < d2; j++){
        double p = (double)P(i,j) / sumP;
        double q = (double)Q(i,j) / sumQ;
        double m = (p + q) / 2.0;
        result += kl(p, m);
        result += kl(q, m);
    }
}

return result;

//move word w from oldCluster to newCluster
//and adjust all of the figures correctly
void Clusters::moveWord(int w, int oldCluster, int newCluster){
    int c = counts[w];
    classVector[w] = newCluster;
    clusterUnigrams[oldCluster] -= c;
    clusterUnigrams[newCluster] += c;

    //update cluster and word signatures
    int left1; //cluster immediately to the left
    int left2; //2nd cluster to the left
    int right1; //cluster immediately to the right
    int right2; //2nd cluster to the right
    int leftWord1, rightWord1; //words immediately to left and right
    int leftWord2, rightWord2; //words 2nd to left and right

    int currentPos = first[w];
    if(currentPos >= 0){
        while(currentPos < numberTokens){
            if(currentPos == 0){
                left1 = numberClasses;
                left2 = numberClasses;
                leftWord1 = -1;
            }
leftWord2 = -1;
}
else if(currentPos == 1){
    left1 = classVector[data[currentPos-1]];
    left2 = numberClasses;
    leftWord1 = data[currentPos-1];
    leftWord2 = -1;
}
else{
    left1 = classVector[data[currentPos-1]];
    left2 = classVector[data[currentPos-2]];
    leftWord1 = data[currentPos-1];
    leftWord2 = data[currentPos-2];
}

if(currentPos == numberTokens-1){
    right1 = numberClasses;
    right2 = numberClasses;
    rightWord1 = -1;
    rightWord2 = -1;
}
else if(currentPos == numberTokens-2){
    right1 = classVector[data[currentPos+1]];
    right2 = numberClasses;
    rightWord1 = data[currentPos+1];
    rightWord2 = -1;
}
else{
    right1 = classVector[data[currentPos+1]];
    right2 = classVector[data[currentPos+2]];
    rightWord1 = data[currentPos+1];
    rightWord2 = data[currentPos+2];
}

//occurrences of w to the right of the current position have not
//changed cluster yet
if(rightWord1 == w)
    right1 = oldCluster;
if(rightWord2 == w)
    right2 = oldCluster;
(*clusterSignatures[oldCluster])(left1, right1) -= 1;
(*clusterSignatures[newCluster])(left1, right1) += 1;

if(left1 < numberClasses){
    (*clusterSignatures[left1])(left2, oldCluster) -= 1;
    (*clusterSignatures[left1])(left2, newCluster) += 1;

    (*wordSignatures[leftWord1])(left2, oldCluster) -= 1;
    (*wordSignatures[leftWord1])(left2, newCluster) += 1;
}

if(right1 < numberClasses){
    (*clusterSignatures[right1])(oldCluster, right2) -= 1;
    (*clusterSignatures[right1])(newCluster, right2) += 1;

    (*wordSignatures[rightWord1])(oldCluster, right2) -= 1;
    (*wordSignatures[rightWord1])(newCluster, right2) += 1;
}

currentPos = next[currentPos];
}
}

//****************************
//clusters.h

#ifndef ASC_CLUSTERING_H
#define ASC_CLUSTERING_H 1

#include <vector>
#include "matrix.h"
#include "simplecorpus.h"

// All words that have frequency <= FREQ_CUTOFF are in
// a separate cluster

extern bool USE_TRUE_WEIGHT;
extern int MAX_ITERATIONS;
extern int FREQ_CUTOFF;
extern bool FULL_MORPHOLOGY_WEIGHT;

// this defines a clustering of a set of words

class Clusters {
    private:
        int numberClasses;
        int numberTypes;
        int numberTokens;
        vector<int> data; // copied from data vector in corpus
        vector<int> next;
        const SimpleCorpusOne & corpus;

        MatrixInt clusterBigrams;
        vector<int> clusterUnigrams;

        // store information about each type correspond to wordArray
        vector<int> first; // location of first occurrence of each type in the corpus
        vector<int> classVector; // class of each type
        vector<int> counts; // count of each type

        vector<int> sortedWords; // sorted from most to least frequent

        vector<MatrixInt*> wordSignatures;
        vector<MatrixInt*> clusterSignatures;

    public:
        Clusters(int numberClasses_,
            const SimpleCorpusOne & corpus,
            bool randomised);

        ~Clusters() {
            for(size_t i = 0; i < wordSignatures.size(); i++)
                delete wordSignatures[i];
            for(size_t i = 0; i < clusterSignatures.size(); i++)
                delete clusterSignatures[i];
void calcWordSignature(int w, MatrixInt* sig);
void calcClusterSignature(int c, MatrixInt* sig);
void calcSignatures();
void checkSignatures();

double divergence(const MatrixInt & P, const MatrixInt & Q, int sumP, int sumQ);
void dump() const;
void clusterNeyEssen();
int reclusterNeyEssen();

bool bestCluster(int w);
void moveWord(int w, int oldCluster, int newCluster);

#endif

//***************
//matrix.cpp

#include "matrix.h"
#include <assert.h>
#include <math.h>
#include <float.h>
#include <vector>
#include <iostream>

//
// Stuff for Matrix of doubles

//

// copy constructor

MatrixDouble::MatrixDouble(const MatrixDouble & other) {
    d1 = other.d1;
    d2 = other.d2;
    int size = d1 * d2;
    v = new double[size];
    for (int i = 0; i < size; i++)
        v[i] = other.v[i];
}

MatrixDouble::MatrixDouble(int x, int y) {
    // cout << "Allocating matrix (" << x << "," << y << ")" << endl;
    assert(x > 0);
    assert(y > 0);
    d1 = x;
    d2 = y;
    int size = x * y;
    v = new double[size];
    for (int i = 0; i < size; i++)
        v[i] = 0.0l;
}

MatrixDouble::~MatrixDouble() {
    delete[] v;
}

void MatrixDouble::dump() const {
    for (int i = 0; i < d1; i++){
        for (int j = 0; j < d2; j++){
            // Do something with v[i][j]
        }
    }
}
void MatrixDouble::dumpNonZero() const
{
    for (int i = 0; i < d1; i++){
        for (int j = 0; j < d2; j++){
            if (getV(i,j)>0)
                cout << "(" << i << "," << j << ")= " << getV(i,j) << " ;"
        
        }
            }
}

const MatrixDouble& MatrixDouble::operator=(const MatrixDouble& other) 
{
    assert(d1 == other.d1);
    assert(d2 == other.d2);
    int n = d1*d2;
    for (int i = 0;i< n;i++)
        v[i]= other.v[i];
    return *this;
}

const MatrixDouble& MatrixDouble::operator=(double newValue) 
{
    int n = d1*d2;
    for (int i = 0;i< n;i++)
        v[i]= newValue;
    return *this;
}

const MatrixDouble& MatrixDouble::operator*=(double newValue) 
{
    int n = d1*d2;
    for (int i = 0;i< n;i++)
        v[i] *= newValue;
}
return *this;
}

double MatrixDouble::operator ()(int x, int y) const
{
    assert(x >= 0 && x < d1);
    assert(y >= 0 && y < d2);
    return v[(x * d2) + y];
}

double& MatrixDouble::operator ()(int x, int y)
{
    assert(x >= 0 && x < d1);
    assert(y >= 0 && y < d2);
    return v[(x * d2) + y];
}

//
// Stuff for Matrix of integers
//
//

// copy constructor

MatrixInt::MatrixInt(const MatrixInt & other)
{
    d1 = other.d1;
    d2 = other.d2;
    int size = d1 * d2;
    v = new int[size];
    for (int i = 0; i < size; i++)
        v[i] = other.v[i];
}

MatrixInt::MatrixInt(int x, int y)
{
// cout << "Allocating MatrixInt (" << x << "," << y << ")" << endl;
assert(x > 0);
assert(y > 0);
d1 = x;
d2 = y;
int size = x * y;
v = new int[size];
for (int i = 0; i < size; i++)
v[i] = 0;
}

MatrixInt::~MatrixInt()
{
    delete[] v;
}

void MatrixInt::dump() const
{
    for (int i = 0; i < d1; i++)
        for (int j = 0; j < d2; j++)
            cout << "(" << i << "," << j << ")=" << getV(i,j) << " ";
    cout << endl;
}

void MatrixInt::dumpNonZero() const
{
    for (int i = 0; i < d1; i++)
        for (int j = 0; j < d2; j++)
            if (getV(i,j)>0)
                cout << "(" << i << "," << j << ")=" << getV(i,j) << " ";
    cout << endl;
}

const MatrixInt& MatrixInt::operator=(const MatrixInt& other)
{
assert(d1 == other.d1);
assert(d2 == other.d2);
int n = d1*d2;
for (int i = 0;i< n;i++)
    v[i]= other.v[i];
return *this;
}

const MatrixInt& MatrixInt::operator=(int newValue)
{
    int n = d1*d2;
    for (int i = 0;i< n;i++)
        v[i]= newValue;
    return *this;
}

const MatrixInt& MatrixInt::operator*=(int newValue)
{
    int n = d1*d2;
    for (int i = 0;i< n;i++)
        v[i] *= newValue;
    return *this;
}

int MatrixInt::operator()(int x, int y) const
{
    assert(x >= 0 && x < d1);
    assert(y >= 0 && y < d2);
    return v[(x * d2) + y];
}

int& MatrixInt::operator()(int x, int y)
{
    assert(x >= 0 && x < d1);
    assert(y >= 0 && y < d2);
    return v[(x * d2) + y];
}
//matrix.h

#ifndef ASC_MATRIX_H
#define ASC_MATRIX_H

// classes for 2d matrices of integers or doubles

#include <assert.h>

using namespace std;

class MatrixInt {
private:
    int d1;
    int d2;
    int * v;
public:
    MatrixInt(int x, int y);
    MatrixInt(const MatrixInt&);
    const MatrixInt& operator=(const MatrixInt&);
    const MatrixInt& operator=(int);
    const MatrixInt& operator*=(int);
    ~MatrixInt();

    int size() const { return d1 * d2; }
    int dim1() const { return d1;}
    int dim2() const { return d2;}

    int sum() const;
    int max() const;

    void dump() const; // dump it to cout
    void dumpNonZero() const; // same but only non-zero elements

    int& operator()(int x, int y);
    int operator()(int x, int y) const;

    int rowSum(int x) const;
    int columnSum(int y) const;

private:
int getV(int x, int y) const {
    return v[(x * d2) + y];
}

int& vv(int x, int y) {
    assert(x < d1 && y < d2);
    return v[(x * d2) + y];
}

void setV(int x, int y, int v_){
    (*this)(x,y)=v_;
}

};

class MatrixDouble {
private:
    int d1;
    int d2;
    double * v;
public:
    MatrixDouble(int x, int y);
    MatrixDouble(const MatrixDouble&);
    const MatrixDouble& operator=(const MatrixDouble&);
    const MatrixDouble& operator=(double);
    const MatrixDouble& operator*=(double);
    ~MatrixDouble();

    int size() const { return d1 * d2; }
    int dim1() const { return d1;}
    int dim2() const { return d2;}

double sum() const;
double max() const;

    void dump() const; // dump it to cout
    void dumpNonZero() const; // same but only non-zero elements

double& operator()(int x, int y);
    double operator()(int x, int y) const;
private:
    double getV(int x, int y) const {
        return v[(x * d2) + y];
    }
    double& vv(int x, int y) {
        assert(x < d1 && y < d2);
        return v[(x * d2) + y];
    }

    void setV(int x, int y, double v_){
        (*this)(x,y)=v_;
    }

};

#endif

// neyessen.cpp
// Do the Ney Essen clustering on some data in the
// simple corpus format.

#include <iostream>
#include "simplecorpus.h"
#include "clusters.h"
#include <unistd.h> /* for: getopt() */

using namespace std;

void printUsage()
{
    cerr << "Usage -- cluster_neyessen "
        << "[-p factor ] use prior probabilities [-i iterations] [-mMinCount] [-r](random initialisation) corpus additionalCorpus clusters" << endl;
    exit(-1);
}

int main(int argc, char* argv[])
{  
  char ch;                   /* to hold command line option */
  //  char *optstr = "sf:v:i:3@";
  char* optstr = "m:i:rhp:";
  
  /*
   * the option string will be passed to getopt(3), the format
   * of our string "sf:v:" will allow us to accept -s as a flag,
   * and -f or -v with an argument, the colon suffix tells getopt(3)
   * that we're expecting an argument. Eg: optest -s -f this -v8
   *
   * getopt(3) takes our argc, and argv, it also takes
   * the option string we set up earlier. It will assign
   * the switch character to ch, and -1 when there are no more
   * command line options to parse.
   *
   */
  bool randomInitialization = false;
  while( -1 != (ch=getopt(argc,argv,optstr))) {
    switch(ch) {
      case 'h':
        printUsage();
        break;
      case 'i':
        MAX_ITERATIONS = atoi(optarg);
        cerr << " iterations option " << MAX_ITERATIONS << endl;
        break;
      case 'r':           /* this is just the flag -- no argument expected */
        randomInitialization = true;
        cerr << "r option = random initialization" << endl;
        break;
      case 'm':
        cerr << "m option " << optarg << endl;
        FREQ_CUTOFF = atoi(optarg);
        break;
    }  
  }  
}
cerr << "Frequency cutoff is " << FREQ_CUTOFF << endl;
break;

case '?':
cerr << "unrecognized option: " << optopt << endl;
printUsage();
break;
}
}
cerr << "Frequency cutoff is " << FREQ_CUTOFF << endl;
if (optind + 3 != argc){
cerr << "Wrong number of arguments " << endl;
printUsage();
}
cerr << "Started ok." << endl;
SimpleCorpusOne corpus(argv[optind],argv[optind + 1]);
cerr << "Loaded corpus ok." << endl;

for(int i = 0; i < corpus.numberTypes; i++){
//cerr << "iteration " << i << " of " << corpus.numberTypes << endl;

if(*(corpus.wordArray[i]) == "TRUE")
cerr << "TRUE found in wordArray index " << i << endl;
}

int clusters = atoi(argv[optind + 2]);
assert(clusters > 0);
Clusters cne(clusters, corpus, randomInitialization);
cerr << "Created initial clustering" << endl;
cne.clusterNeyEssen();
cne.dump();
return 0;
}
```cpp
#include <iostream>
#include <fstream>
#include <algorithm>
#include "simplecorpus.h"

SimpleCorpusOne:::
~SimpleCorpusOne()
{
    for (size_t i = 0; i < wordArray.size(); i++)
        delete wordArray[i];
}

const int BUFLEN = 1000; //length of buffer for reading each line

// basic constructor
// loads corpus from filename
SimpleCorpusOne:::
SimpleCorpusOne(const char* filename, const char* additional)
:
    numberTypes(0),
    numberTokens(0)
{
    //open file to count tokens
    //assumes one token per line
    ifstream in(filename);
    if (!in)
    {
        cerr << "Couldn't open file." << endl;
        exit(-1);
    }
    char buffer[BUFLEN];
    while (!in.eof())
    {
        in.getline(buffer, BUFLEN);
        numberTokens++;
    }
    in.close();

    //reopen file to read corpus
    //closing and reopening starts reading from the begining of the corpus as desired
```
ifstream inn(filename);
data.resize(numberTokens);
int currentToken = 0;
int currentType = 0;
while (!inn.eof()){
  inn.getline(buffer, BUFLEN);
  string word(buffer);
  if (dictionary.find(word) == dictionary.end()){
    // new word
    // cerr << "New word " << word << endl;
    dictionary[word] = currentType;
    data[currentToken] = currentType;
    currentType++;
    wordArray.push_back(new string(word));
    countArray.push_back(1);
  }
  else {
    //cerr << "old word " << endl;
    int wordIndex = dictionary[word];
    countArray[wordIndex]++;
    data[currentToken] = wordIndex;
  }
  currentToken++;
}
numberTypes = currentType;

cerr << "read " << numberTypes << " types" << endl;
cerr << "read " << numberTokens << " tokens" << endl;
if (additional){
cerr << "Starting to read additional file for vocabulary\n"; ifstream in2(additional);
if (!in2)
  {cerr << "Couldn't open file." << endl;}
else {
cerr << "Opened file " << additional << endl;
  int newTokens = 0;
}
while (!in2.eof()){ 
    newTokens++; 
    in2.getline(buffer,BUFLEN); 
    string word(buffer); 
    if (dictionary.find(word) == dictionary.end()){ 
        // new word 
        // cerr << "New word " << word << endl; 
        dictionary[word] = currentType; 
        currentType++; 
        wordArray.push_back(new string(word)); 
        countArray.push_back(0); 
    } 
    cerr << "Read " << newTokens <<" new tokens\n"; 
}
}

numberTypes = currentType; 
cerr << "read " << numberTypes << " total types" << endl; 
}
/**
 * Return the index of word or -1 if it isn't in the dictionary 
 */

int SimpleCorpusOne::
lookUpWord(const string & word)
const {
    const 
    { 
        map<string,int>::const_iterator pos = dictionary.find(word); 
        if (pos != dictionary.end())
            return pos->second; 
        else 
            return -1; 
    }
/**
 * Count the number of types in the corpus 
 * whose frequency is greater than or equal to f 
 */

int
SimpleCorpusOne::
countWords(int f)
  const
  {
    int count = 0;
    for (int i = 0; i< numberTypes; i++){
      if (countArray[i] >= f)
        count++;
    }
    return count;
  }
/**
 * Return the position of the next occurrence of string in the corpus
 * that is AFTER start.
 * If there is none return the number of tokens in the corpus.
 */

int
SimpleCorpusOne::
findNextOccurrence(const string & target,
  int start)
  const
  {
    int w = lookUpWord(target);
    for (int index = start + 1; index < numberTokens; index++){
      if (data[index] == w)
        return index;
    }
    return numberTokens;
  }

//simplecorpus.h

#ifndef ASC_CORPUS_H
#define ASC_CORPUS_H 1

#include <string>
#include <vector>
#include <map>
using namespace std;

//
// load up the corpus
// this is in a simple one-line per word format
// keep a hash table
// of previously seen words
// linked to an index
//

class SimpleCorpusOne
{
public:

    SimpleCorpusOne(const char* filename, const char * additional);
    ~SimpleCorpusOne();

    int numberTypes;
    int numberTokens;
    //int boundaryType;  **unused?
    vector<int> data;  // stores sequence of words from the corpus as vector of integers
    vector<string *> wordArray;  // lists word types
    vector<int> countArray;  // number of occurances of coresponding word in wordArray
    map<string, int> dictionary; // dictionary of word types in the corpus and the integers used to represent them in this program

    const string & getNthWord(int i) const
    {
        return *(wordArray[i]);
    }

    const string & getNthDataPoint(int i) const
    {
        int w = data[i];
        return *(wordArray[w]);
    }

    int lookUpWord(const string &) const;
    // find next occurrence of the word
    int findNextOccurrence(const string &, int start) const;
    int countWords(int f) const;
};

#endif