Principal Direction Divisive Partitioning*

Daniel Boley†

Department of Computer Science and Engineering

University of Minnesota

200 Union Street S.E., Rm 4-192

Minneapolis, MN 55455, USA

Abstract

We propose a new algorithm capable of partitioning a set of documents or other samples based on an embedding in a high dimensional Euclidean space (i.e. in which every document is a vector of real numbers). The method is unusual in that it is divisive, as opposed to agglomerative, and operates by repeatedly splitting clusters into smaller clusters. The splits are not based on any distance or similarity measure. The documents are assembled in to a matrix which is very sparse. It is this sparsity that permits the algorithm to be very efficient. The performance of the method is illustrated with a set of text documents obtained from the World Wide Web. Some possible extensions are proposed for further investigation.

1 Introduction

Unsupervised clustering of documents is a critical component for the exploration of large unstructured document sets. As part of a larger project, the WebACE Project [11, 8], we have developed a novel algorithm for unsupervised partitioning of a large data set which exhibits several useful features. These features include scalability to large data sets, competitive

*This research was partially supported by NSF grants CCR-9405380 and CCR-9628786.

†tel: 1-612-625-3887, fax: 1-612-625-0572, e-mail: `boley@cs.umn.edu`

performance in terms of the quality of the clusters generated, and capability of working on data sets where the number of attributes is much larger than the number of sample document. The overall WebACE project aims to create an agent that would help users explore the World Wide Web (WWW) by carrying out the following steps:

1. Starting with a seed set of user-retrieved documents, automatically classify documents retrieved.
2. Generate new WWW searches using classes discovered in step 1 or 3.
3. Classify newly received documents into existing classes, or update the classification structure.
4. Repeat from step 2.

A critical part of this task is a method to automatically discover structure present in a large set of documents and to classify a new set of documents according to a previously computed set of classes. The goal of this paper is to present one such algorithm which we have observed is capable of competitive performance in terms of speed, scalability, and quality of class structure found.

We make a remark on the origin of the name "Principal Direction Divisive Partitioning." The words "Principal Direction" are used because the algorithm is based on the computation of the leading principal direction at each stage in the partitioning. The key component in this algorithm that allows it to operate fast is a fast solver for this direction. This principal direction is used to cut a cluster of documents repeatedly. No reference to a distance or similarity measure is made, except for the very limited purpose of deciding which cluster should be split next, but the similarity measure is not used to do the actual splitting.

We use the word "Partitioning" to reflect the fact that we place all the documents in one cluster, so that at every stage the clusters are disjoint and their union equals the entire set of documents.

The word "Divisive" comes from the following taxonomy of clustering algorithms in [12, p298] (originally from [1]):

1. hierarchical agglomerative clustering
2. hierarchical divisive clustering
3. iterative partitioning
4. density search clustering

5. factor analytic clustering
6. clumping
7. graph-theoretic clustering

Our algorithm is a hierarchical algorithm, but the overwhelming majority of existing hierarchical algorithms which treat the documents as a point in Euclidean space work "bottom up" by agglomeration. Our PDDP algorithm is "top down" or "divisive" in the sense that it starts with all the documents in one single big cluster and proceeds to divide up the initial cluster into progressively smaller clusters.

As a hierarchical divisive algorithm, the PDDP algorithm would be appropriate for the scatter/gather task of [4]. The scatter/gather task consists of a "scatter" of a large collection of documents into a "small" number of clusters, and a "gather" task in which certain of those resulting clusters are combined. The scatter/gather process is then repeated on the resulting gathered cluster. In [4], the scatter task was carried out by a modified agglomeration algorithm to which some heuristics were applied to achieve running times faster than the $O(m^2)$ expected running time.

In their scenario, they proposed that the gather task would be carried out by a user who would be given information on the nature of the documents in each cluster, for example the most frequent words or sample document titles. They also proposed a "join" operator on clusters that could be used to automate the gather task. The "join" operator is defined in terms of the list of the k most common words in the first cluster, for some given k. If at least l words from this list (for some given $l \leq k$) also appear among the k most common words in the second cluster, then the two clusters are joined.

2 Notation and Mathematical Preliminaries

We will represent column vectors with lower case bold letters, matrices with upper case bold letters, scalars with italic letters, viz. $\mathbf{v} = (v_1, v_2, \ldots, v_n)^T$ where n is the dimension of $\mathbf{v}$. The transpose of a matrix $\mathbf{M}$ is the matrix $\mathbf{M}^T$ whose i,j-th element is m_{ji}, where m_{ij} denotes the i,j-th element of $\mathbf{M}$. A row vector will be denoted by $\mathbf{u}^T$. The notation $\mathbf{e} = (1, 1, \ldots, 1)^T$ will stand for a vector of all ones of appropriate dimension. In this paper we will use the Euclidean norm for vectors:

$$\|\mathbf{v}\|_2 = \sqrt{\sum_j v_j^2}, \tag{1}$$

and the Frobenius norm for matrices:

$$\|\mathbf{M}\|_F = \sqrt{\sum_{i,j} m_{ij}^2}, \tag{2}$$

The algorithm presented in this paper will be applied to document vectors. A document vector $\mathbf{d} = (d_1, d_2, \ldots, d_n)^T$ is a column vector whose i-th entry, d_i, is the relative frequency of the i-th word. In this particular application, we scale the document vectors to have Euclidean norm equal to 1, so that each entry has the numerical value

$$d_i = \frac{\mathrm{TF}_i}{\sqrt{\sum_j (\mathrm{TF}_j)^2}}, \tag{3}$$

where TF_i is the number of occurences of word i in the particular document $\mathbf{d}$. We refer to the scaling (3) as "norm scaling." An alternative scaling is the TFIDF scaling [13], defined as follows:

$$\text{let } \tilde{d}_i = \frac{1 + \frac{\mathrm{TF}_i}{\max_j(\mathrm{TF}_j)}}{\log\left(\frac{1}{\mathrm{DF}_i}\right)}, \quad \text{then } d_i = \frac{\tilde{d}_i}{\sqrt{\sum_j (\tilde{d}_j)^2}}, \tag{3.1}$$

where DF_i is the number of different documents in which the i word appears, among all documents in the entire document set. However, this scaling results in a nonzero value for every d_i, and hence destroys the sparsity structure present in the matrix. In addition it did not produce distinctly better clustering results, at least in our experiments, compared to (3). The sparsity structure is critical for the speed of our algorithm.

Given a collection of documents $\mathbf{d}_1, \ldots, \mathbf{d}_m$, the mean or centroid of the document set is

$$\mathbf{w} = \frac{\mathbf{d}_1 + \cdots + \mathbf{d}_m}{m} = \mathbf{M} \cdot \mathbf{e} \cdot \frac{1}{m}, \tag{4}$$

where $\mathbf{M} = (\mathbf{d}_1, \ldots, \mathbf{d}_m)$ is an $n \times m$ matrix of document vectors. If $\mathbf{w} = \mathbf{0}$, then the covariance matrix of this set would be $\mathbf{M} \cdot \mathbf{M}^T$, since each sample is a column vector, but in the general case the covariance matrix is

$$\mathbf{C} = (\mathbf{M} - \mathbf{w}\mathbf{e}^T) \cdot (\mathbf{M} - \mathbf{w}\mathbf{e}^T)^T = \mathbf{A} \cdot \mathbf{A}^T, \tag{5}$$

where $\mathbf{A} = (\mathbf{M} - \mathbf{w}\mathbf{e}^T)$. This matrix is symmetric and positive definite, so all its eigenvalues are real and non-negative. The eigenvectors corresponding to the k largest eigenvalues are called the *principal components* or *principal directions*. In our algorithm, we are interested only in the eigenvectors of $\mathbf{C}$, not the eigenvalues, so the specific scaling of the matrix $\mathbf{C}$ is not important to our algorithm.

The singular value decomposition (SVD) of an $n \times m$ matrix $\mathbf{A}$ is defined by the expression $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ where $\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}$ are, respectively, $n \times n$, $n \times m$, $m \times m$, $\mathbf{U}^T\mathbf{U} = \mathbf{I}$, $\mathbf{V}^T\mathbf{V} = \mathbf{I}$, and $\mathbf{\Sigma} = \mathit{diag}\{\sigma_1, \sigma_2, \ldots, \sigma_{\min\{m,n\}}\}$. The diagonal entries of $\mathbf{\Sigma}$ are called *singular values* and satisfy $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{\min\{m,n\}} \geq 0$. The columns of $\mathbf{U}\mathbf{V}$ are the *left* and *right singular vectors*, respectively. It is well known [7] that the eigendecomposition of $\mathbf{C} = \mathbf{A}\mathbf{A}^T$ is related to the SVD of $\mathbf{A}$ by

$$\mathbf{C} = \mathbf{A}\mathbf{A}^T = (\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T) \cdot (\mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T) = \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^T, \tag{6}$$

where we use $\mathbf{\Sigma}^2$ as a shorthand for the $n \times n$ diagonal matrix

$$\mathit{diag}\{\sigma_1^2, \sigma_2^2, \ldots, \sigma_{\min\{m,n\}}^2, 0, \ldots, 0\}$$

(extended with zeros if $m < n$).

3 Algorithm Description

The Principal Compoment Divisive Partitioning algorithm operates on a sample space of m samples in which each sample is an n-vector containing a numerical value. For clarity of presentation we will use the terms "document" to refer to a sample, but in fact this algorithm can in principle be applied to samples in other domains for which the appropriate scaling may differ. Each document is represented by a column vector (3) of attribute values, which in the case of actual text documents are word counts. In our experiments using web documents, we normalized each document vector to have a euclidean length of 1. For the purposes of our algorithm, the entire set of documents is represented by an $n \times m$ matrix $\mathbf{M} = (\mathbf{d}_1, \ldots, \mathbf{d}_m)$ whose i-th column, $\mathbf{d}_i$, is the column vector representing the i-th document.

The algorithm proceeds by separating the entire set of documents into two partitions by using principal directions in a way that we will describe. Each of the two partitions will be separated into two subpartitions using the same process recursively. The result is a hierarchical structure of partitions arranged into a binary tree (the "PDDP tree") in which each partition is either a leaf node (meaning it has not been separated) or has been separated into two subpartitions forming its two children in the PDDP tree. The details of the algorithm we must specify are (1) what method is used to split a partition into two subpartitions, and (2) in what order are the partitions selected to be split. The rest of this section is devoted to filling in this details.

3.1 Splitting a Partition

A partition of p documents is represented by an $n \times p$ matrix $\mathbf{M}_p = (\mathbf{d}_1 \quad \cdots \quad \mathbf{d}_p)$ where each $\mathbf{d}_i$ is an n-vector representing a document. The matrix $\mathbf{M}_p$ is a submatrix of the original matrix $\mathbf{M}$ consisting of some selection of p columns of $\mathbf{M}$, not necessarily the first p in the set, but we omit the extra subscripts for simplicity. The principal directions of the matrix $\mathbf{M}_p$ are the eigenvectors of its sample covariance matrix $\mathbf{C}$. Let $\mathbf{w} = \mathbf{M}_p\mathbf{e}/p$ be the sample mean of the documents $d_1, \ldots, d_p$, and the covariance matrix is $\mathbf{C} = (\mathbf{M}_p - \mathbf{w}\mathbf{e}^T)(\mathbf{M}_p - \mathbf{w}\mathbf{e}^T)^T$. The Karhunen-Loeve transformation [5] consists of projecting the columns of $\mathbf{M}_p$ onto the space spanned by the leading k eigenvectors (those corresponding to the k largest eigenvalues). The result is a representation of the original data in k degrees of freedom instead of the original n. Besides reducing the dimensionality, the transformation often has another beneficial effect of removing much noise present in the data, assuming an appropriate value of k can be chosen. In our case, we are interested in temporarily projecting each document onto the single leading eigenvector, which we will denote $\mathbf{u}$. This leading eigenvector is called the principal component or principal direction. This projection is needed only to determine the split and is not otherwise used for any purpose.

The projection of the i-th document $\mathbf{d}_i$ is given by the formula

$$\sigma v_i = \mathbf{u}^T(\mathbf{d}_i - \mathbf{w}), \tag{7}$$

where σ is a positive constant arising from the specific algorithm we use. In words, we translate all the documents so that their mean is at the origin, and then project the result onto the principal direction. The values $v_1, \ldots, v_k$ are used to determine the splitting for the cluster $\mathbf{M}_p$. In the simplest version of the algorithm, we split the documents strictly according to the sign of the corresponding v_i's. All the documents $\mathbf{d}_i$ for which $v_i \leq 0$ are partitioned into the left child, and all the documents $\mathbf{d}_i$ for which $v_i > 0$ are put into the right child. A document coinciding exactly with the mean vector $\mathbf{w}$ is more or less in the middle of the entire cloud of documents, and its projection (7) will be zero. In this case we make the arbitrary choice to put such documents into the left child.

3.2 Computational Considerations

In the computation of the splitting of the partition, the single most expensive part of the computation is the computation of the eigenvalues and eigenvectors of the covariance matrix $\mathbf{C}$. When $\mathbf{C}$ has large dimensionality this can be a significant expense. Standard off-the-shelf methods typically compute all the eigenvalues and eigenvectors. Even if special methods that compute only selected eigenvectors are used, many methods still require the computation of all the eigenvalues [7]. In addition, the covariance matrix is actually the product of a matrix and its transpose. This is exactly the situation where it is well known that better accuracy can be had by using the Singular Value Decomposition (SVD) [7]. The SVD of an $n \times m$ matrix $\mathbf{A}$ is the decomposition $\mathbf{U\Sigma V}^T = \mathbf{A}$, where $\mathbf{U}, \mathbf{V}$ are square orthogonal matrices of dimension $n \times n$, $m \times m$, respectively, and $\mathbf{\Sigma}$ $diag\{\sigma_1, \ldots, \sigma_{\min\{m,n\}}\}$, where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{\min m,n} \geq 0$. The σ's are called the singular values, the columns of $\mathbf{U}$ are called the left singular vectors, and the columns of $\mathbf{V}$ are called the right singular vectors. An orthogonal matrix $\mathbf{U}$ is a square matrix satisfying $\mathbf{U}^T\mathbf{U} = \mathbf{I}$.

Setting $\mathbf{A} = \mathbf{M}_p - \mathbf{we}^T$, we can show how to relate the SVD to the principal directions. We have

$$\mathbf{C} = \mathbf{AA}^T = (\mathbf{U\Sigma V}^T) \cdot (\mathbf{V\Sigma}^T\mathbf{U}^T) = \mathbf{U\Sigma}^2\mathbf{U}^T,$$

where we use $\mathbf{\Sigma}^2$ as a shorthand for the $n \times n$ diagonal matrix $diag\{\sigma_1^2, \sigma_2^2, \ldots\}$. From this formula, it is seen that the eigenvalues of $\mathbf{C}$ are the squares of the singular values of $\mathbf{A}$, and the eigenvectors of $\mathbf{C}$ are the left singular vectors of $\mathbf{A}$. The singular vectors also satisfy another interesting relation. Let $\mathbf{u}_j$,$\mathbf{v}_j$ denote the j-th column of $\mathbf{U}, \mathbf{V}$, respectively. We have that

$$\mathbf{u}_j^T\mathbf{A} = \mathbf{u}_j^T\mathbf{U\Sigma V}^T = \sigma_j\mathbf{v}_j^T, \tag{8}$$

and likewise $\mathbf{Av}_j = \sigma_j\mathbf{u}_j$. Using our definition for $\mathbf{A}$, we see that the projections (7) are exactly the entries in the vector $\sigma_1\mathbf{v}_1^T$. This means that once we have the SVD of the matrix $\mathbf{A}$, we already have the projections needed to split the cluster $\mathbf{M}_p$.

In order to compute the projections (7), it is clear that we do not need the entire SVD of $\mathbf{A}$. We need only the leading singular vectors $\mathbf{u}_1, \mathbf{v}_1$, which we will denote $\mathbf{u}, \mathbf{v}$ for short. A fast method for computing a partial SVD of a matrix can be constructed using the Lanczos algorithm [7]. This algorithm takes advantage of sparsity present in the matrix $\mathbf{M}_p$. In our implementation of this algorithm, we implicitly compute the eigenvalues of $\mathbf{A}\mathbf{A}^T$ or $\mathbf{A}^T\mathbf{A}$, whichever has a smaller dimension. The accuracy we obtain is quite sufficient for our purposes, since we need only to have the signs of (7). The Lanczos algorithm proceeds by repeatedly multiplying a vector $\mathbf{x}$ by the matrix $\mathbf{C}$ and then orthogonalizing the result against all previous vectors in the sequence. The algorithm also constructs a sequence of nested tridiagonal matrices of increasing dimensions, computing the eigenvalues for each one, until a stopping test is satisfied. For readers interested in the details of the Lanczos algorithm we implemented, we include a few details about the algorithm. We used no re-orthogonalization, and our stopping test was based on the observation that the eigenvalues of each tridiagonal matrix interlace those of the next one in the sequence. Hence the Lanczos algorithm proceeds until the largest eigenvalue stops growing, at which time we assume it has converged. Since only the largest eigenvalue is sought, a Sturm sequence method can be used to compute it fast from the largest eigenvalue of the preceeding tridiagonal matrix, and furthermore, spurious eigenvalues can be ignored. Once the eigenvalue is found, the generated Lanczos vectors are used to compute the approximate corresponding eigenvector. All of these properties mentioned here are based on the theory present in [7] and hence is not further discussed here.

3.3 Overall Algorithm

We summarize the overall algorithm used to carry out the partitioning. The basic algorithm constructs a binary tree, the PDDP tree. Each node in the tree is a data structure which holds the documents associated with that node, the various quantities computed from that set of documents, and pointers to the two children nodes, if any. Table 1 lists the specific information stored in each node in the tree in our implementation. The information stored in each node consists of the documents in the cluster associated with that node (to save space we store only the indices of the documents), the centroid (mean) vector, the leading singular value and associated singular vectors, and pointers to the children nodes, if any. We

field	contents
`indices`	indices of documents in this node's cluster (p integers)
`centroid`	the mean vector $\mathbf{w}$ of documents in cluster (n-vector)
`sval`	leading singular value σ for this cluster (a scalar)
`leftvec`	left singular vector $\mathbf{u}$ corresponding to σ (n-vector)
`rightvec`	right singular vector $\mathbf{v}$ corresponding to σ (p-vector)
`scat`	total scatter value for this cluster (a scalar)
`leftchild`	pointer to the left child node
`rightchild`	pointer to the right child node

Table 1: Definition of a node in the binary PDDP tree for a cluster with p documents. The root node contains all m documents, each of length n.

also store a "scatter" value, which we discuss next.

Having described the methods used to split a given node, the remaining question is to decide at each stage which node should be split next. One choice is to try to keep the binary tree balanced by splitting all the nodes at a given level (distance form root) before proceeding to the next level. But the resulting clusters are often imbalanced with a few large clusters and many small clusters, including many singletons. To avoid this situation, we use a "scatter" value as a measure of the noncohesiveness of a cluster. In this first version implemented for this paper, we used a norm-based scatter value, but the user is free to use any measure of cohesiveness that works on the particular data set being processed. Whatever method is used to obtain this "scatter" value, the algorithm will select the cluster with the largest scatter value to be split next.

For our experiments, the specific value we used was defined as follows. The `scat` field is a *total scatter value* defined to be the Frobenius norm of the corresponding matrix $\mathbf{A} = \mathbf{M}_p - \mathbf{w}\mathbf{e}^T$. The square of the Frobenius norm of $\mathbf{A} = (a_{ij})$ is given by

$$\|\mathbf{A}\|_F^2 = \sum_{i,j} |a_{ij}|^2,$$

and equals the Frobenius norm of the covariance matrix $\mathbf{C}$ as well as the sum of the eigenvalues σ_i^2 of $\mathbf{C}$ [7]:

$$\|\mathbf{A}\|_F^2 = \|\mathbf{C}\|_F = \sum_i \sigma_i^2.$$

0. **Start** with $n \times m$ matrix $\mathbf{M}$ of (scaled) document vectors,
   and a desired number of clusters $c_{\max}$.
1. **Initialize** Binary Tree with a single Root Node (Table 1).
2. **For** $c = 2, 3, \ldots, c_{\max}$ **do**
3. **Select** node `K` with largest `scat` value
4. **Create** nodes `L` := `leftchild(K)` and `R` := `rightchild(K)`.
5. **Set** `indices(L)` := indices of the non-positive entries in `rightvec(K)`
6. **Set** `indices(R)` := indices of the positive entries in `rightvec(K)`
7. **Compute** all the other fields for the nodes `L`, `R`.
8. **end.**
9. **Result:** A binary tree with $c_{\max}$ leaf nodes forming a partitioning
   of the entire data set.

Table 2: Principal Direction Divisive Partitioning Algorithm Summary

In our algorithm, the total scatter value is used to select the next cluster to split. We choose the cluster with the largest scatter value. The total scatter value reflects the distance between each document in the cluster and the overall mean of the cluster, which is a measure of the cohesiveness of the cluster. Using the total scatter value to choose the next cluster to be split usually results in clusters all having more or less similar numbers of documents. We remark that this scatter value is the only component of this algorithm that is based on a "distance" measure, and it would be just as easy to use other measures not based on a "distance" measure and appropriate for particular data sets.

Having discussed all the components of our algorithm, we now summarize the overall algorithm in Table 2. At each pass through the main loop, we select a node based on our measure of "cohesiveness," obtain the mean vector and principal direction for the documents associated with that node, and split the documents using the mean vector and principal direction into two children nodes. In geometric terms, we split the documents using the hyperplane normal to the principal direction passing through the mean vector.

The space required for the PDDP algorithm arises from three sources: the memory to store the initial raw data matrix $\mathbf{M}$, the memory needed to store the PDDP tree, and temporary storage for the SVD computation. The space to store $\mathbf{M}$ is fixed. Regarding the PDDP tree, we start with a single node, and in each step we add two new nodes to the tree. Therefore, when finding $c_{\max}$ clusters, we will have a total of $2 \cdot c_{\max} - 1$ nodes in the tree, of which $c_{\max}$ are leaf nodes. Within each node, we require storage for at least 2 n-vectors (`centroid` and `leftvec`), as well as space for the indices (`indices`) and some scalars. The remaining vector `rightvec` can be omitted to save space by recomputing it when needed using (8). In addition, `leftvec` and `rightvec` are not needed for the leaf nodes, unless one would like to be able to continue the splitting process from the current tree instead of starting over from scratch. Therefore the total storage needed to store the tree is $(2 \cdot c_{\max} - 1)$ `centroid`s plus $c_{\max}$ `leftvec`s plus $(2 \cdot c_{\max} - 1)$ `indices`, for a total of at most $(3 \cdot c_{\max} - 1) \cdot n + (2 \cdot c_{\max} - 1) \cdot m$ words, not counting the scalar values.

The temporary space required by the SVD computation depends on the number of iterations needed for the method to converge, and specific estimates require a detailed description of the algorithm in linear algebra, which is beyond the scope of this paper. However, a simple estimate can be had by observing that space is needed for one vector for each iteration, where the length of this vector is $\min\{n, m\}$. The number of iterations can vary from data set to data set, and even depend on the choice of starting vector, but in our experiments never exceeded 20 steps.

So the total space required can be summarized as follows:

item	memory words occupied
original matrix $\mathbf{M}$	$n \cdot m \cdot s_{\text{NZ}}$
PDDP tree	$(3 \cdot c_{\max} - 1) \cdot n + (2 \cdot c_{\max} - 1) \cdot m$
SVD temporary space	$\min\{n, m\} \cdot k_{\text{SVD}}$

(10)

where s_{NZ} is the fraction of entries in $\mathbf{M}$ that are nonzero, $c_{\max}$ is the number of clusters generated, k_{SVD} is the number of Lanczos iterations in the SVD computation. In our application, $n >> m$, so $\min\{n, m\} = m$. In addition, typical values for s_{NZ} in our examples range from $.04 = 4\%$ down to $.0068 = .68\%$.

The bulk of the cost in the algorithm of Table 2 is the SVD computation in step 7. We have already mentioned that the main memory required for this step is space for k_{SVD} m-vectors. The cost of generating each of those vectors is dominated by two matrix vector products involving the matrix $\mathbf{M}_p$. Taking advantage of the sparsity of $\mathbf{M}$, the cost of a single matrix vector product involving $\mathbf{M}$ is approximately $s_{\text{NZ}} \cdot m \cdot n$. Hence the total cost of the matrix vector products within the SVD step is approximately $k_{\text{SVD}} s_{\text{NZ}} \cdot m \cdot n$. This is, of course, an upper bound since the later clusters contain much fewer than m documents present initially. Beyond the SVD computation, the equivalent of a matrix vector product is required for the `centroid` vector and for the `scat` value. So the total cost of the matrix vector products is bounded above by

$$c_{\max} \cdot (2 + k_{\text{SVD}}) s_{\text{NZ}} \cdot m \cdot n. \tag{11}$$

The cost of all the rest of the computation is equivalent to lower order terms. We remark that this expected running time is linear in the number of documents (m), modulo the number of iterations within the SVD computation, whereas unmodified agglomeration algorithms typically have $O(m^2)$ running time [4].

4 Related Work

Existing approaches to document clustering are generally based on either probabilistic methods, or distance and similarity measures (see [6]). Distance-based methods such as k-means analysis, hierarchical clustering [9] and nearest-neighbor clustering [10] use a selected set of words (features) appearing in different documents as the dimensions. Each such feature vector, representing a document, can be viewed as a point in this multi-dimensional space.

AutoClass [3] is a method using Bayesian analysis based on the probabilistic mixture modeling [14]. Given a data set it finds maximum parameter values for a specific probability distribution functions of the clusters.

There are a number of problems with clustering in a multi-dimensional space using traditional distance- or probability-based methods. First, it is not trivial to define a distance measure in this space. Some words are more frequent in a document than other words. Simple frequency of the occurrence of words is not adequate, as some documents are larger than others. Furthermore, some words may occur frequently across documents. Techniques such as TFIDF [13] have been proposed precisely to deal with some of these problems.

Secondly, the number of all the words in all the documents can be very large. Distance-based schemes generally require the calculation of the mean of document clusters. If the dimensionality is high, then the calculated mean values may not differ significantly from one cluster to the next, especially if the initial set of clusters are poor. Hence the cluster means may not provide good separation. Similarly, probabilistic methods such as Bayesian classification used in AutoClass [3] do not perform well when the size of the feature space is much larger than the size of the sample set and the attributes are not statistically independent, as is typical of document categorization applications on the Web. It is possible to reduce the dimensionality by selecting only frequent words from each document, or to use some other method to extract the salient features of each document. However, the number of features collected using these methods still tends to be very large, and determining which features can be discarded without affecting the quality of clusters is difficult.

Latent Semantic Indexing (LSI) [2] was proposed as a method for query-based document retrieval in which the noise present in data sets of very high dimensionality is reduced by orthogonal projection. A low rank (say, rank $k << \min\{m, n\}$) approximation to the document matrix $\mathbf{M}$ is computed using the leading k singular values and vectors of the matrix. This has the effect of removing the noise, while representing each document by means of a set of k "generalized attributes." But unlike the PDDP algorithm, the SVD is applied to $\mathbf{M}$, not $\mathbf{A} = (\mathbf{M} - \mathbf{w}\mathbf{e}^T)$, and is used only to preprocess the data to reduce the dimensionality of its representation. In addition, computing k leading singular values and vectors is considerably more complicated and expensive than computing just one as in the PDDP Algorithm.

The formula (7) is an example of a linear discriminant function. Linear discriminant functions have been used extensively to partition samples in a test set into two classes. An example is the Fisher linear discriminant function (see e.g. [12]), typically used with a training set of samples with known "correct" classifications. As such it is usually used as a tool in "supervised learning," in which a training set with previously known class designations are used. An "optimal" direction is chosen on which to project all the samples, and a cut-point is selected using a one-dimensional Bayes rule based on the known mean and covariance matrices for the individual classes.

Previous algorithms for unsupervised clustering based on the use of one-dimensional Bayesian analysis or linear discriminants is very limited, at least to the knoweldge of this author. A hint in this direction appears in [12, p500], where the repeated use of a Fisher-style linear discriminant is suggested, resulting in a hierarchical classification. It is even suggested that the Karhunen Loeve transformation might lead to good directions. However no algorithm is given and no discussion of the specific structure that might result from such an algorithm is discussed.

5 Experimental Results

We present some experimental results that show that the PDDP algorithm is effective, at least as well as an agglomeration algorithm, but is much faster. We used a set of 185 documents downloaded from the World Wide Web, which were then processed to obtain word counts for each document. To obtain word counts, we removed the stop words (fixed in advance), stemmed the remaining words for plurals, verb tenses, etc., and counted up the occurences. The result was the matrix "J1" with 185 columns corresponding to the 185 documents, and 10536 rows each corresponding to a word. Then further heuristics were applied to obtain the matrices for data sets J2 through J11. These are summarized in Table 3 [8].

We applied two algorithms to this data set, the divisive PDDP algorithm (Table 2), and an agglomerative algorithm [5]. The agglomerative algorithm is shown only for comparative purposes, and is briefly summarized as follows. We start with m singleton clusters, one per individual document. We then repeatedly find the two clusters that are "closest together" according to a distance measure [5]. Those two clusters are merged into a single cluster, and the process is repeated. The "distance" is defined as the angle between the cluster centroids. This simple agglomerative algorithm was effective for some scalings, but slow.

In both cases we used either the TFIDF scaling (3.1) [13] or the norm scaling (3). For the agglomerative algorithm, the times were independent of the scaling. The TFIDF scaling destroys the sparsity, so the performance of the PDDP algorithm is substantially degraded and is not reported. The resulting times are shown in Table 5. It is seen that the divisive algorithm is faster by an order of magnitude. For the purpose of demonstrating the effectiveness of the algorithm on larger data sets, we did apply it to an unlabeled data set of 2340 documents and 21839 words on a Sun SuperSparc workstation, on which it took 198 seconds to obtain 16 clusters. However, until labels have been assigned it is not possible to derive a comparable entropy measure, so further analysis of the scalability and performance of this algorithm will be reserved for a future paper.

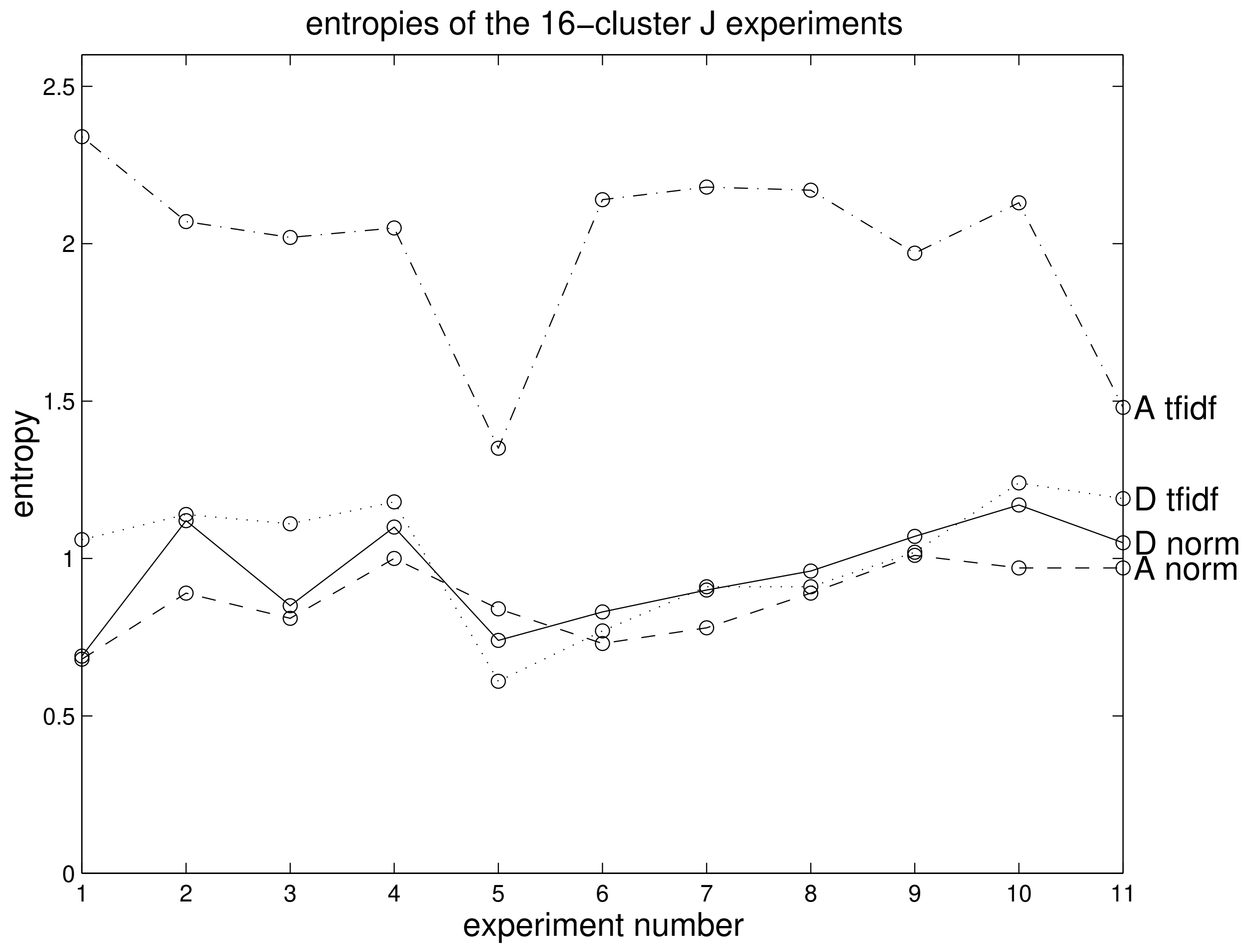

Figure 1: Entropies for the 16-cluster experiments. Lines with dots use TFIDF scaling, without dots norm scaling. Lines with dashes use agglomerative algorithm, without dashes use divisive algorithm.

expe-riment	word count	contents summary
J1	10536	all words
J6	5106	words with TF > 1
J4	2951	Top 5+ words + HTML-tagged words
J3	1763	Top 20+ words
J7	1328	Top 20+ with TF > 1
J8	1105	Top 15+ with TF > 1
J2	946	words accounting for 25% of document
J9	805	Top 10+ with TF > 1
J10	474	Top 5+ with TF > 1
J5	449	words appearing in a-priori word clusters
J11	183	hypergraph word clusters.

Table 3: Experimental data set summary, sorted by number of words. TF refers to Text Frequency (number of occurrences of a word).

In Table 6 we indicate the quality of the clusters generated. This quality was measured by first assigning labels to each document by hand, and using those labels to compute an entropy measure for each cluster. The labels used in our data set are given in Table 4.

The entropy of cluster j is defined by

$$e_j = -\sum_i \left(\frac{c(i,j)}{\sum_i c(i,j)} \right) \cdot \log \left(\frac{c(i,j)}{\sum_i c(i,j)} \right),$$

where $c(i,j)$ is the number of times label i occurs in cluster j. The entropy for a cluster is zero if the the labels of all the documents are the same, otherwise it is positive. The total entropy is the weighted average of the individual cluster entropies:

$$e_{\text{total}} = \frac{1}{m} \sum_j e_j \cdot (\text{number of documents in cluster } j).$$

As a consequence, the lower the entropy the better the quality. It is seen that the PDDP algorithm is competitive with the agglomeration algorithm with norm scaling. This can also be seen in Fig. 1. For the divisive algorithm, it turns out that one of the best results is obtained by using all the words (J1). Another data set that yielded low entropy was J5 in

label	topic	label	topic
A	affirmative action	L	employee rights
B	business capital	M	materials processing
C	information systems	P	personnel management
E	electronic commerce	S	manufacturing systems
I	intellectual property	Z	industrial partnership

Table 4: Topic labels assigned by hand to the documents.

which words had been selected based on their membership is multi-document word clusters. This selection was carried out by preprocessing step as part of a hypergraph algorithm which is also capable of clustering. Comparative results of this algorithm have been reported in [11, 8].

We illustrate in Table 7 a sample cluster from one of the cases of Table 6. Table 7 shows 16 clusters in which for each cluster we show the hand-assigned labels for documents in that cluster as well as the 5 most common words in the cluster. It is seen that the theme of each cluster can be deduced from the leading words. We remark that if we were to apply the "join" operator [4] described in Section 1 with $k = 5$ and $l = 3$, then clusters 3 and 4 would be joined. Further consolidation could occur using $l = 2$ or perhaps with larger values of k. This aspect deserves further investigation.

6 Extensions and Future Work

The binary tree constructed by the PDDP algorithm of Table 2 can be extended in several ways. In this section, we discuss several extensions that could be applied to this algorithm, including some preliminary experimental results validating some of them. We remark that we have already mentioned the possible application of the PDDP algorithm to the scatter/gather method of [4].

6.1 Classification of New Documents

The simplest extension to the PDDP algorithm of Table 2 is to classify a new set of documents according to the clusters from an original document set by using the original binary PDDP

data set	divisive algorithm 8 clusters	16 clusters	32 clusters	agglomerative algorithm 16 clusters	32 clusters
J1	1:40	1:54	2:10	–	–
J6	0:39	0:44	0:50	53:15	52:57
J4	0:24	0:27	0:30	29:23	29:14
J3	0:15	0:17	0:20	16:54	16:48
J7	0:13	0:14	0:17	11:35	11:31
J8	0:21	0:23	0:25	11:31	11:23
J2	0:11	0:12	0:14	7:49	7:45
J9	0:10	0:11	0:14	6:19	6:15
J10	0:08	0:09	0:11	3:51	3:49
J5	0:11	0:13	0:16	3:38	3:35
J11	0:07	0:08	0:11	1:45	1:43

Table 5: Comparative times (min:sec) for divisive and agglomerative algorithms, in Matlab 5 on an SGI Challenge 194 MHz processor.

data set	number of clusters:									
	8	16*	32	8	16*	32	16*	32	16*	32
	divisive algorithm						agglomerative algorithm			
	norm scaling			TFIDF scaling			norm scaling		TFIDF scaling	
J1	1.24	$\boxed{0.69}$	0.51	1.46	1.06	0.71	0.68	0.47	2.34	1.56
J6	1.33	0.83	0.56	1.17	0.77	0.65	0.73	0.55	2.14	1.32
J4	1.53	1.10	0.71	1.65	1.18	0.93	1.00	0.72	2.05	1.28
J3	1.33	0.85	0.61	1.57	1.11	0.93	0.81	0.62	2.02	1.20
J7	1.36	0.90	0.61	1.28	0.91	0.72	0.78	0.58	2.18	1.38
J8	1.47	0.96	0.69	1.32	0.91	0.77	0.89	0.65	2.17	1.47
J2	1.70	1.12	0.76	1.56	1.14	0.81	0.89	0.62	2.07	1.24
J9	1.65	1.07	0.76	1.42	1.02	0.76	1.01	0.73	1.97	1.15
J10	1.69	1.17	0.85	1.90	1.24	0.99	0.97	0.75	2.13	1.29
J5	1.31	0.74	0.51	1.07	0.61	0.46	0.84	0.55	1.35	0.84
J11	1.47	1.05	0.67	1.68	1.19	0.92	0.97	0.73	1.48	1.02

Table 6: Entropies. Starred columns are plotted in Fig. 1. Clusters for boxed result shown in Table 7.

1:	`A A A A A A A A A A A A A A`				
	affirm	action	people	discrimin	act
2:	`A A A A A A L L P`				
	employ	action	applic	minor	affirm
3:	`B B B B B B B B B B Z`				
	busi	capit	develop	fund	loan
4:	`B B B B B B B B B M P`				
	busi	capit	financ	fund	financi
5:	`C C C C C C C C C C C S`				
	inform	system	manag	servic	technologi
6:	`C C C M S S Z Z Z Z Z Z`				
	industri	research	inform	system	engin
7:	`C C M S S S S S S S S S S S`				
	system	manufactur	inform	integr	develop
8:	`C E E E E E E E E E E E E E E E`				
	electron	internet	commerc	busi	web
9:	`C E E E I I I I I I I L M`				
	copyright	right	web	internet	advertis
10:	`C M Z Z Z Z Z Z Z`				
	industri	partnership	inform	program	contact
11:	`E P P P P Z`				
	public	manag	personnel	hb	depart
12:	`I I I I I I I I I I I I`				
	patent	intellectu	properti	copyright	inform
13:	`L L L L L L L L L L L L L P`				
	employe	union	employ	court	right
14:	`M M M M M M M M M M M`				
	materi	process	engin	manufactur	develop
15:	`M S S S S Z Z Z Z`				
	technologi	manufactur	develop	industri	product
16:	`P P P P P P P P P P P P`				
	personnel	manag	servic	job	feder

Table 7: Labels and 5 most common words (after stemming) within each of 16 clusters computed by PDDP algorithm on the J1 data set.

tree. For simplicity, we discuss the task of classifying a single new document **d** using the PDDP tree constructed from an initial set of documents. The PDDP algorithm yields a collection of clusters, and the classification task is to select the cluster most closely associated with the new document. Since the initial set of documents have been separated by a sequence hyperplanes, the resulting clusters occupy regions in n-dimensional Euclidean space bounded by the hyperplanes (polytope regions), and each document from the initial set has been placed into one of these polytope regions. The classification task for the new document **d** can be viewed as determining in which polytope region it lies. One can then view **d** as most closely associated with the documents from the initial set occupying the same polytope region.

The classification task then proceeds as follows. From the root node of the tree we retrieve the mean vector, **w** and principal direction vector **u** corresponding to the entire initial set of documents. These two vectors define a hyperplane which was used to split the initial set of documents, but the same hyperplane actually cuts the entire space. Hence we can assign the new document **d** to the left child or the right child of the root depending on which side of this hyperplane it lies. Specifically, apply the formula (7) to the new document **d**

$$\sigma v = \mathbf{u}^T(\mathbf{d} - \mathbf{w}), \tag{12}$$

using the retrieved vectors $\mathbf{w}, \mathbf{u}$. If the resulting value v is positive, the **d** is assigned to the root's rightchild, otherwise it is assigned to the left child. This process is then repeated to assign **d** to either the left or right child at the next level, until ultimately **d** percolates down to a leaf node. The resulting algorithm is summarized in Table 8.

To test this classification method, we conducted a preliminary experiment of attempting to classify half the J1 document set using a PDDP tree constructed using the other half of the documents. We selected the documents so that both document sets had representatives of every topic label. We used the method of Table 2 on the first half of the document set to construct a new PDDP tree with 16 leaf nodes, then we used the method of Table 8 to classify the remaining documents, so that we end up with a set of 16 clusters, each containing documents from both halves of the original document set. The resulting entropy of the combined result (i.e. all the documents) was 0.756, only slightly degraded from the original entropy of 0.69 obtained when all 185 documents are processed together.

Of course, the purpose of this process is to classify documents from different sources, but we have not yet addressed issues of merging such documents into a common data set. In

0. **Start** with PDDP binary tree and a new document **d**.
1. **Initialize** `node` := root node of binary tree.
2. **While** `node` is not a leaf node **do**
3. **Retrieve** $\mathbf{w}$:= `centroid(node)` and $\mathbf{u}$:= `leftvec(node)`.
4. **Compute** $v := \mathbf{u}^T(\mathbf{d} - \mathbf{w})$.
5. **If** $v > 0$ **then Set** `node` := `rightchild(node)`
6. **else Set** `node` := `leftchild(node)`
8. **end.**
9. **Result:** `node` contains the leaf node to which **d** was assigned.

Table 8: A Classification Algorithm: classify a new document **d** by finding associated cluster of old documents.

particular, if the word dictionary used for the two sets of documents are different, then some paradigm must be used to select the words to be kept in the combined set. This is especially critical if the words used in each individual set have been pruned as in experiments J2-J11. By adding documents, the combined set of words may not be the same as what one would obtain if all the documents were processed together. This will all be reserved for future investigation.

6.2 Classification Updates

We address the problem of updating the PDDP tree when new documents have arrived. One can consider just the leaf nodes which represent the actual clusters and treat this simply as a problem of updating the existing clusters. One may use the Classification algorithm to assign all the new documents to existing clusters and then use one of the many existing updating algorithms on the set of clusters (see, e.g., [5, p201 or p225] or [4]) The drawback of such an approach is that clusters created by another updating method no longer has any connection with the PDDP tree.

A possible approach for rapidly constructing a new PDDP tree is outlined as follows. The obvious idea is simply combine the entire new set of documents with the initial set and run the PDDP algorithm from scratch. Of course this does not constitute updating, since the cost will be more than the cost of the PDDP algorithm on either data set alone. However we can save substantial costs by replacing the initial set of documents with the set of centroid vectors extracted from the tree. We then combine the centroid vectors with the set of new documents to form a set not much larger than the new set of documents. The success of this approach depends on the capability of the centroids alone to capture the structure of the original document set. So we show the results of an experiment that attempts to measure this capability.

To illustrate and support this idea, we considered the following experiment. We treated half the documents in the J1 data set as the "old" documents, and half as the "new" documents. We extracted the 31 centroid vectors obtained during the construction of 16 clusters for the "old" documents, combined them with the "new" documents, and constructed a new PDDP tree using this combined set. To test the quality of the result, we used the resulting tree to classify the entire J1 data set and found the entropy to be 0.728 compared to 0.69 (the boxed value in Table 6) when all the documents are clustered together. The result is sensitive to scaling of the centroid vectors, for example to account for the fact that the centroids are being used to represent a larger set of documents. For example, scaling the centroids by 1.1 without modifying the algorithm yields a an entropy for the entire J1 set of .572, but the best scaling appropriate in similar situations is not known. This indicates that the centroids do capture a significant part of the structure of the "old" documents.

By using "new" documents from the same source as the "old" documents, we automatically are using the same set of words for both sets. In general, however, documents from different sources will most likely be based on different word sets, so one is faced with the problem of merging two different word sets. This aspect, and many other aspects about this update strategy remain to be investigated.

7 Conclusions

We have proposed a new algorithm for the unsupervised partitioning of documents that is based on the use of principal directions (also known as principal components), and is "divisive" in nature, in contrast to almost all other algorithms based on the embedding of documents in high dimensional Euclidean space. No use of a distance function is made except for the very limited use of deciding which cluster to split next. We have illustrated its performance using a set of documents obtained from the World Wide Web and compared it to a traditional agglomerative algorithm. The use of this algorithm is not limited to text documents, but can be applied to any sample space embeddable in Euclidean space. We have also indicated some possible extensions and uses for this algorithm and pointed out directions for future work.

Acknowledgments

The author would like to acknowledge Robert Gross for carrying out some of the experiments reported in this paper, and all the other members of the WebACE team: M. Gini, S. Han, K. Hastings, G. Karypis, V. Kumar, B. Mobasher, and J. Moore, for bringing this problem to the attention of the author, for retrieving the documents, and for computing and saving the matrices of word counts for various experimental data sets used in this paper.

References

[1] M. R. Anderberg. *Cluster Analysis for Applications.* Academic Press, 1973.

[2] M. W. Berry, S. T. Dumais, and G. W. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Review*, 37:573–595, 1995.

[3] P. Cheeseman and J. Stutz. Baysian classification (autoclass): Theory and results. In U. Fayyad, G. Piatetsky-Shapiro, P. Smith, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 153–180. AAAI/MIT Press, 1996.

[4] D. Cutting, D. Karger, J. Pedersen, and J. Tukey. Scatter/gather: a cluster-based approach to browsing large document collections. In *15th Ann Int ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'92)*, pages 318–329, 1992.

[5] R. O. Duda and P. E. Hart. *Pattern Classification and scene analysis.* John Wiley & Sons, 1973.

[6] W. B. Frakes and R. Baeza-Yates. *Information Retrieval Data Structures and Algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[7] G. H. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins Univ. Press, 3rd edition, 1996.

[8] S. Han, D. Boley, M. Gini, R. Gross, K. Hastings, G. Karypis, V. Kumar, B. Mobasher, and J. Moore. WebACE: A web agent for document categorization and exploration. Univ. of Minn. Comp. Sci. report TR 97-049, 1997.

[9] A. Jain and R. C. Dubes. *Algorithms for Clustering Data.* Prentice Hall, 1988.

[10] S. Lu and K. Fu. A sentence-to-sentence clustering procedure for pattern analysis. *IEEE Transactions on Systems, Man and Cybernetics*, 8:381–389, 1978.

[11] J. Moore, S. Han, D. Boley, M. Gini, R. Gross, K. Hastings, G. Karypis, V. Kumar, and B. Mobasher. Web page categorization and feature selection using association rule and principal component clustering. In *7th Workshop on Information Technologies and Systems (WITS '97)*, Dec. 1997. Atlanta.

[12] M. Nadler and E. P. Smith. *Pattern Recognition Engineering.* Wiley, 1993.

[13] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

[14] D. Titterington, A. Smith, and U. Makov. *Statistical Analysis of Finite Mixture Distributions.* John Wiley & Sons, 1985.