Accelerating GLA with an M-Tree

Olli Luoma, Johannes Tuikkala, and Olli Nevalainen

Abstract—In this paper, we propose a novel improvement for the
generalized Lloyd Algorithm (GLA). Our algorithm makes use of an
M-tree index built on the codebook which makes it possible to reduce
the number of distance computations when the nearest code words
are searched. Our method does not impose the use of any specific
distance function, but works with any metric distance, making it
more general than many other fast GLA variants. Finally, we present
the positive results of our performance experiments.

Keywords—Clustering, GLA, M-Tree, Vector Quantization.

I. INTRODUCTION

Clustering - a process of classifying objects into groups
according to their similarity [1] - has been extensively
studied and put to use in many different application areas,
such as data mining [2] and pattern recognition [3]. The
importance of clustering has also passed over to the area of
data compression where clustering is used as a means for
codebook generation in vector quantization [4]. In this setting,
clustering is used to find a given number of code vectors, in
other words, a codebook, for a given set of training vectors
by minimizing the average pairwise distance between the training
vectors and their representative code vectors.

In this paper, we concentrate on the widely known
generalized Lloyd algorithm (GLA). The algorithm starts with
an initial, e.g., random, codebook which is iteratively
improved until some convergence condition is met. Each
iteration of GLA consists of two steps, the partition step
in which each training vector is assigned to its closest code
vector, and the codebook step in which the code vectors are
updated based on the partitioning found in the partition step. If
GLA is straightforwardly implemented, every partition step
requires \(n \cdot m\) distance computations, where \(n\) denotes the
size of the codebook and \(m\) the number of training vectors.

Our idea is to reduce the number of distance calculations by
building an M-tree [5] over the codebook and using this tree
to find the nearest code vector. Since the nearest code vector
for any given training vector can now be found with a
logarithmic number of distance computations, only
\(O(\log(n)) \cdot m\) distance computations are needed in the
partition step. Obviously, this method yields better results as
the size of the codebook increases, and thus our method is at
its best in clustering tasks involving relatively high number of
clusters.

Our method also has the additional advantage of being
more general than many other GLA variants. Many other
methods which force the use of Euclidean or otherwise
restricted distance function, but here, the distance function is
only assumed to be metric, i.e., to satisfy the non-negativity,
symmetry, and triangle inequality postulates.

The remainder of this paper is organized as follows. In
section II, we briefly review the related work, and in section
III, we present some basic concepts related to M-trees. Our
clustering method is presented in section IV and the results of
our performance evaluation in section V. Section VI
concludes this article and discusses our future work.

II. RELATED WORK

Index structures have previously been used to speed up GLA
by building an index on training vectors and using simple
geometric reasoning to reduce the number of distance
calculations [6, 7, 8]. The use of an kd-tree or other spatial
access methods, however, imposes the distance metric to be an
\(L_p\) norm ruling out the use of more sophisticated distance
functions, such as Levenshtein distance or Mahalanobis
distance. Furthermore, as observed in [8], the performance of
these methods is seriously degraded as the dimensionality of
the data increases.

The partial distortion search (PDS) [9] aims at reducing the
number of distance calculations by computing the distance
between a code vector candidate and a training vector cumulatively
by summing up the squared differences in each dimension. If the cumulative distance exceeds the distance
between the training vector and the closest code vector found
thus far, the code vector candidate is rejected. The mean-
distance-ordered partial search (MPS) [10] utilizes a less
expensive distance function to find a lower bound for the distance
between a code vector candidate and a training vector. If this value is greater than the current minimum
distance the candidate is rejected. Since most of
the information needed to calculate the lower bound can be
precalculated, this method can reduce the running time
substantially. However, MPS can only be applied if Euclidean
distance metric is used.

The triangular inequality elimination technique (TIE) [11],
on the contrary, does not force the use of any specific distance
function, and thus we regard TIE as the most relevant GLA variant to our paper. In TIE, it is assumed that the distance function is metric, and hence the distance calculation between a code vector \( C_i \) and a training vector \( T_j \) can be avoided if

\[
d(C_i, C_o) > 4 \cdot d(T_j, C_o)
\]

(1)

where \( d(O_i, O_j) \) denotes the distance between \( O_i \) and \( O_j \) and \( C_o \) denotes the nearest code vector found thus far. A practical implementation of TIE utilizes a matrix of the distances between all code vectors which is updated at the beginning of each partition step. Since updating the matrix requires \( n(n-1)/2 \) distance calculations, where \( n \) denotes the size of the codebook, the performance of TIE significantly degrades when the number of clusters increases.

The code vector activity detection proposed by Kaukoranta et al. [12] is based on the concepts of active and static code vectors. If a training vector is assigned to a static code vector, i.e., a code vector which was not changed in the last codebook step, only the distances to active code vectors, i.e., code vectors which were changed in the last codebook step, have to be computed. This method can be applied to a wide range of GLA variants, including the method proposed in this paper.

III. METRIC SPACES AND M-TREES

A metric space is defined as a pair \( M = (D, d) \), where \( D \) is a domain of feature values and \( d : D \times D \rightarrow R^+ \) is a distance function such that for all \( O_i, O_j, O_k \in D \):

\[
d(O_i, O_j) = 0 \iff O_i = O_j
\]

(2)

\[
d(O_i, O_j) = d(O_j, O_i)
\]

(3)

\[
d(O_i, O_j) \leq d(O_i, O_k) + d(O_k, O_j)
\]

(4)

Metric spaces can be indexed using so called metric trees [13] which only consider the relative distances between objects. One metric tree structure is the M-tree proposed by Ciaccia et al. [5]. In an M-tree, all indexed objects reside on the leaf level. Each inner node stores a routing object and its covering radius, i.e., the maximum distance between the routing object and the objects residing in the subtrees corresponding to the routing object.

For accessing the indexed objects, M-tree provides two search methods. In the range query, the query object and the maximum distance are specified, and in the \( k \) nearest neighbors query, the query object and the cardinality of the result set are the input parameters. Both types of queries start from the root and recursively traverse all the paths which cannot be excluded from the search. For our purposes, it is sufficient to say that in most cases, the number distance computations involved in both types of queries grows logarithmically with respect to the size of the tree. This, of course, is typical to tree structures, since the height of the tree also grows logarithmically with respect to tree size. For a detailed description of the M-tree, we refer the reader to [5].

IV. OUR ALGORITHM

The main intuition behind our algorithm is very simple. By building an M-tree over the codebook at the start of each partition step, we can expect to find the nearest code vector for each of the training vectors with a logarithmic number of distance computations. Of course, the building of the M-tree introduces some overhead which in typical clustering tasks, however, is negligible since the number of clusters compared to the number of training vectors is typically relatively small. This is evident in the results of our experimental evaluation presented in section VI.

Our algorithm is presented in Fig. 1. Operation clear empties the M-tree and operation knn\((O,k)\) returns \( k \) nearest neighbors of \( O \). Each partition step is preceded by building of a new M-tree over the codebook which takes approximately \( n \cdot \log(n) \) distance computations. We then use the M-tree to find the nearest code vector for each of the training vectors by issuing a nearest neighbor query. This can be done using \( O(\log(n)) \cdot m \) distance computations. After this, the codebook is updated and the improvement of the partition checked.

\[
\text{M-TREE-GLA}(T) \\
\text{IN: Training set } T \\
\text{Generate codebook } C \text{ by any algorithm;} \\
\text{REPEAT} \\
\text{tree.clear;} \\
\text{FOR EACH } C_i \in C \text{ DO} \\
\text{tree.insert( } C_i \text{);} \\
\text{FOR EACH } T_i \in T \text{ DO} \\
T_i.cluster \leftarrow \text{tree.knn( } T_i, 1\text{);} \\
\text{FOR EACH } C_i \in C \text{ DO} \\
C_i.update; \\
\text{UNTIL no improvement achieved}
\]

Figure 1. Pseudo-code of our algorithm.

V. EXPERIMENTAL RESULTS

We evaluated the performance of our method by performing codebook generation tasks on three standard CCIT test images presented in Fig. 2 - Fig. 4, with varying number of clusters. The size of all images was 256x256 pixels and the training vectors were 4x4 pixel blocks from the images. Thus, the number of training vectors in all cases is 4096. We also implemented the simple GLA and TIE as proposed in [11]. All algorithms were implemented using Java.

Fig. 3 illustrates the average number of distance computations per training vector per iteration for these three images using six different codebook sizes. The \( n \cdot (n-1)/2 \) distance computations needed to update the distance matrix in TIE and the \( n \cdot \log(n) \) distance computations needed to build the tree in our M-tree variant are included in the results.

Fig. 5 clearly shows that the performance of TIE degrades significantly as the number of clusters increases. In the case of 1024 clusters, for example, updating the distance matrix requires 524288 distance computations at the beginning of
each partition step, whereas building the M-tree requires approximately 10000 distance computations.

Fig. 6 illustrates the average running time per training vector per iteration. These results include not only the time needed for the distance computations, but also the time needed to sort the rows of the distance matrix in TIE, and thus TIE performs even worse. Overall, our M-tree variant clearly outperforms both TIE and simple GLA in clustering tasks involving a large number of clusters.

VI. CONCLUSION AND FUTURE WORK

We introduced an improvement to GLA which utilizes an M-tree built on the codebook. Unlike many other methods, our variant does not force the use of any specific distance function, which makes it more general than many other methods. We also presented the results of our performance evaluation which suggested that the M-tree variant can outperform the TIE method proposed by Chen and Tsieh [11].

![Figure 2. Bridge (256x256).](image1)

![Figure 3. Camera (256x256).](image2)

![Figure 4. Couple (256x256).](image3)

![Figure 5. The average number of distance calculations per training vector per iteration. The results are averages for Bridge, Camera, and Couple, 10 runs for each image.](image4)

![Figure 6. The average running time (in milliseconds) per training vector per iteration. The results are averages for Bridge, Camera, and Couple, 10 runs for each image.](image5)
obvious solution, building separate M-trees for active code vectors and for all code vectors, introduces some unnecessary overhead.

REFERENCES


