INDEXING IMAGES IN HIGH-DIMENSIONAL AND DYNAMIC-WEIGHTED FEATURE SPACES

King-Shy Goh and Edward Chang
Electrical & Computer Engineering
University of California, Santa Barbara
kingshy@engineering.ucsb.edu, echang@ece.ucsb.edu

Abstract As information retrieval systems evolve to deal with multimedia content, we see the dimensions of content feature space increasing, and relevance feedback being employed to provide more accurate query results. In this paper, we propose using Tree-structured Vector Quantization (TSVQ) to index high-dimensional data for supporting efficient similarity searches and effective relevance feedback. To support efficient similarity searches, we first use TSVQ to cluster data and store each cluster in a sequential file. We then model a similarity search as a classification problem—similar objects are much more likely to be found in the clusters into which the query object is classified. When relevance feedback is considered, and thereby features are weighted differently, we show that our approach remains very effective. Our empirical study on both a 51K and a one-million-image dataset shows that tackling indexing as a classification problem and solving the problem with TSVQ is efficient, effective, and scalable with respect to both data dimensions and dataset size.

Keywords: Approximate search, high-dimensional index, similarity search, TSVQ.

1. INTRODUCTION

As information moves from text-based content towards multimedia content, existing infrastructure may be inadequate for organizing and indexing the new richer forms of data. In many applications, we are given a query object and we search for similar objects (also known as nearest neighbors) in the database. Similarity is usually measured by some distance function between feature vectors, and a good indexing structure is crucial for achieving efficient and accurate searches.

Very often, the multimedia content is characterized by very high-dimensional features (e.g., color, shape and texture features), which cause traditional indexing structures to succumb to the well-known "curse of dimensionality." The search space expands exponentially with the number of dimensions, and objects adjacent to each other in that space are not likely to occupy contiguous space on disk. In addition, many retrieval systems try to personalize search results by learning

X. Zhou et al. (eds.), *Visual and Multimedia Information Management*

the user's query concept with relevance feedback. This interaction allows the system to map the user's high-level query to a set of low level features of varying importance. That is, features may be weighted differently from one query to another, and from one user to another. Most indexing structures are built to work with features having a fixed set of weightings, not a dynamic set.

To deal with the dimensionality-curse problem and to support dynamic feature weightings, we propose an indexing scheme using clustering and classification methods for supporting *approximate similarity searches*. In many applications it is sufficient to perform an approximate search that returns many but not all nearest neighbors [2, 12, 10, 20, 22, 24]. (A feature vector is often an approximate characterization of an object, so we are already dealing with approximations.) For instance, in content-based image retrieval [7, 14, 30] and document copy detection [6, 9, 15], it is usually acceptable to miss a small fraction of the target objects. Thus it is not necessary to incur the high cost of an exact search.

Our indexing method is a statistical approach that works in two steps. It first performs non-supervised clustering using Tree-Structured Vector Quantization (TSVQ) to group similar objects together. To maximize IO efficiency, each cluster is stored in a sequential file. A similarity search is then treated as a classification problem. Our hypothesis is that if a query object's class prediction yields C probable classes, then the probability is high that its nearest neighbors can be found in these C classes. This hypothesis is analogous to looking for books in a library. If we want to look for a calculus book and we know calculus belongs in the math category, by visiting the math section we can find many calculus books. Similarly, by searching for the most probable clusters into which the query object might be classified, we can harvest most of the similar objects.

To achieve accurate class prediction, we experiment with cluster centroids and variance. We also study the effect of cluster size and different metrics for measuring search results. Furthermore, we assess the feasibility of supporting dynamic feature weighting during a similarity search on TSVQ clusters formed without feature weightings. Our empirical study on both a 51K and a one-million-image dataset shows that tackling indexing as a classification problem and solving the problem with TSVQ clustering is efficient, effective, and scalable with respect to both data dimensions and dataset size.

The rest of this paper is organized as follows. In Section 2 we discuss related work. In Section 3 we introduce the TSVQ clustering algorithm and the classification method. We also provide an overview of how a similarity search is conducted and how feature weighting is done. We present the results of our experiments in Section 4. Finally, we offer our conclusions in Section 5.

2. RELATED WORK

In this section we discuss related work in three categories:

1. Tree-like index structures for similarity search,
2. Approximate similarity search, and
3. Indexing for dynamic feature weighting.

2.1. Tree-like Index Structures

Many tree structures have been proposed to index high-dimensional data (e.g., R*-tree [3, 17], SS-tree [32], SR-tree [21], TV-tree [25], X-tree [5], M-tree [11], and K-D-B-tree [27]). A tree structure divides the high-dimensional space into a number of subregions, each containing a subset of objects that can be stored in a small number of disk blocks. Given a vector that represents an object, a similarity query takes the following three steps in most systems [16]:

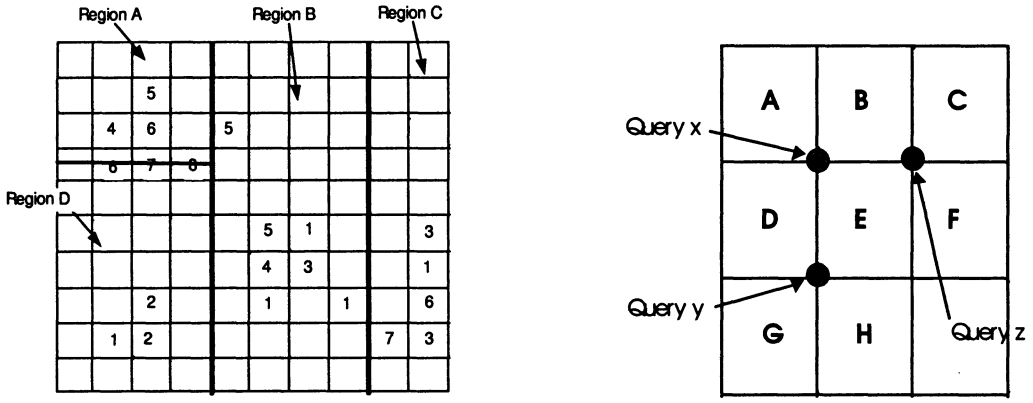
1 It performs a *where-am-I* search to find the subregion in which the given vector resides.

2 It then performs a *nearest-neighbor* search to locate the neighboring regions where similar vectors may reside. This search is often implemented using a *range* search, which locates all the regions that overlap with the search sphere, i.e., the sphere centered at the given vector with a diameter d.

3 Finally, it computes the distances (e.g., Euclidean, street-block, or L^∞ distances) between the vectors in the nearby regions (obtained from the previous step) and the given vector. The search result includes all the vectors that are within distance d from the given vector.

The performance bottleneck of similarity queries lies in the first two steps. In the first step, if the index structure does not fit in the main memory and the search algorithm is inefficient, a large portion of the index structure must be fetched from the disk. In the second step, the number of neighboring subregions can grow exponentially with respect to the dimension of the feature vectors. If D is the number of dimensions, the number of neighboring subregions can be on the order of $O(3^D)$ [16]. Roussopoulos et al. [28] propose the *branch-and-bound* algorithm and Hjaltason and Samet [18] propose the *priority queue* scheme to reduce the search space. But, when D is very large, even the reduced number of neighboring regions can still be quite large. Berchtold et al. [4] propose the pyramid technique that partitions a high dimensional space into $2D$ pyramids and then cuts each pyramid into slices that are parallel to the base of the pyramid. This scheme may not perform satisfactorily when the data distribution is skewed or when the search hypercube touches the boundary of the data space.

In addition to being copious, the IOs can be random and hence exceedingly expensive. An example can illustrate what we call the *random-placement* syndrome faced by traditional index structures. Figure 1(a) shows a 2-dimensional Cartesian space divided into 10 equal stripes in both the vertical and the horizontal dimensions, forming a 10×10 grid structure. The integer in a cell indicates how many points (objects) are in the cell. Most index structures divide the space into



(a) Clustering (b) Random Placement

Figure 1. The shortcomings of tree structures.

subregions of equal points in a top-down manner. Suppose each disk block holds 20 objects. One way to divide the space is to first separate it into three vertical compartments (i.e., left, middle, and right), and then to divide the left compartment horizontally. We end up with four subregions – A, B, C and D – containing about the same number of points. Given a query object residing near the border of A, B and D, the similarity query has to retrieve blocks A, B and D. The number of subregions to check for the neighboring points grows exponentially with respect to the data dimension.

Furthermore, since in high-dimensional spaces the neighboring subregions cannot be arranged sequentially for all possible query objects, the IOs must be random. Figure 1(b) shows a 2-dimensional example of this random phenomenon. Each grid in the figure, such as A and B, represents a subregion corresponding to a disk block. The figure shows three possible query objects x, y and z. Suppose that the neighboring blocks of each object are its four surrounding blocks. For instance, blocks A, B, D and E are the four neighboring blocks of object x. If the neighboring blocks of objects x and z are contiguous on disk, then the order must be $CFEBAD$, or $FCBEDA$, or their reverse orders. Then it is impossible to store neighboring blocks of object y contiguously on disk, and this query will suffer random IOs. This example suggests that in high-dimensional spaces, neighboring blocks of a given object are dispersed randomly on disk by tree structures.

Many theoretical papers (e.g., [2, 20, 22]) discuss the cost of an exact search, independent of the data structure used. In particular, these papers show that if N is the size of a dataset, D is the dimension, and $D >> log\ N$, then no nearest-neighbor algorithm performs significantly faster than a linear search.

2.2. Approximate Similarity Search

Many studies propose conducting an approximate similarity search for applications where trading a small percentage of recall for a faster search speed is

acceptable. For example, instead of searching in all the neighboring blocks of the query object, study [31] proposes performing only the where-am-I step of a similarity query, and returning only the objects in the disk block where the query object resides. However, this approach may miss some objects similar to the query object. Take Figure 1(a) as an example. Suppose the query object is in the circled cell in the figure, which is near the border of regions A, B and D. If we return only the objects in region D where the query object resides, we miss many nearest neighbors in A.

Arya and Mount [2] suggest doing only ε-approximate nearest-neighbor searches, for $\varepsilon > 0$. Let d denote the function computing the distance between two points. We say that $p \in P$ is an ε-approximate nearest neighbor of q if for all $p' \in P$ we have $d(p,q) \leq (1+\varepsilon)d(p',q)$. Many follow-up studies have attempted to devise better algorithms to reduce search time and storage requirements. For example, Indyk and Motwani [20] and Kushilevitz et al. [24] give algorithms with polynomial storage and query time polynomial in $logn$ and d. Indyk and Motwani [20] give another algorithm with smaller storage requirements and sublinear query time. Most of this work, however, is theoretical. The only practical scheme that has been implemented is the *locality-sensitive hashing* scheme proposed by Indyk and Motwani [20]. The key idea is to use hash functions such that the probability of collision is much higher for objects that are close to each other than for those that are far apart. Approximate search has also been applied to tree-like structures; [10] shows that if one can tolerate $\varepsilon > 0$ relative error with a δ confidence factor, one can improve the performance of M-tree by 1-2 orders of magnitude.

Although an ε-approximate nearest-neighbor search can reduce the search space significantly, its recall can be low. This is because the candidate space for sampling the nearest neighbor becomes exponentially larger than the optimal search space. To remedy this problem, a follow-up study of [20] builds multiple *locality-preserving* indexes on the same dataset [19]. This is analogous to building n indexes on the same dataset, with each index distributing the data differently. To answer a query, one retrieves one block following each of the indexes and combines the results. This approach achieves better recall than can be achieved by having only one index. But in addition to the n times pre-processing overhead, it has to replicate the data $n-1$ times to ensure that sequential IOs are possible via every index. Furthermore, a hash-based scheme like this cannot support dynamic feature weightings, a support which is critical for personalizing a query.

2.3. Indexing for Dynamic Feature Weighting

Our indexing approach can perform a similarity search on the clusters using a weighted distance function. Many researchers have shown that by assigning proper feature importance, a nearest-neighbor search can be personalized and hence improved [23]. Feature weighting has the effect of transforming the feature space. For a tree-like structure, supporting dynamic weightings may not cause too

severe a problem, since the splitting conditions can also be transformed. (However, tree-like structures are ill-suited for searches in high-dimensional spaces.) For a hash-based approach, such as the locality-preserving hashing scheme [19], a new set of hash functions is needed for every set of feature weights. Most feature weightings are learned through relevance feedback by a user in a multimedia information retrieval system. This means that a new structure must be built each time the user changes the weights assignment; this is clearly not acceptable.

3. CLUSTERING AND CLASSIFICATION METHODS

Since traditional approaches suffer from a large number of random IOs, our design objectives are to reduce the number of IOs, and to make the IOs sequential as much as possible. To accomplish these objectives, we propose a statistical clustering/classification approach. The design goals of this approach are:

1 Cluster similar data on disk to minimize disk latency,
2 Use classification to select clusters that can give k nearest objects, commonly called top$-k$ Nearest Neighbors ($k-$NNs), to query objects, and
3 Support relevance feedback, which adjusts feature weights dynamically during a search.

We first use TSVQ to perform non-supervised clustering on a set of training vectors. TSVQ is based on sound statistical theory and is adaptive to data distribution. More importantly, TSVQ is fast and is scalable to large and high-dimensional datasets[1].

Once the clusters are formed, we can process a similarity search by retrieving *selected* clusters into memory, then scanning the clusters for objects that are similar to the query. To minimize the number of IOs, we rank the clusters such that a highly-ranked cluster has high probability of providing the greatest number of k nearest objects. To achieve this goal, we model cluster ranking as a classification problem. We treat the query as an object to be classified. The cluster to which the query "belongs" is likely to contain objects that are similar to it.

In the remainder of this section, we describe the TSVQ clustering algorithm, followed by a description of two cluster ranking schemes. Finally, we outline how a similarity search and feature weighting are conducted.

3.1. Tree-structured Vector Quantization (TSVQ)

A $D-$dimensional tree-structured vector quantizer (TSVQ) is a fixed-rate (R) quantizer whose encoding requires the traversal of a binary tree of depth DR. The rate R gives the average number of bits per vector or codeword. A greedy method

[1] Clustering techniques have been studied in the statistics, machine learning, and database communities. Recent works include CLARANS [26], BIRCH [33], DBSCAN [13], CLIQUE [1], and WaveClusters [29]. The speed of these algorithms lags TSVQ substantially. For example, TSVQ clusters 1-million 144-dimensional image vectors on a Pentium-III PC in less than an hour, whereas a quadratic algorithm can take more than a week.

based on the generalized Lloyd algorithm (GLA) is applied on a training set to construct the tree.

Figure 2 illustrates the case for a dimension of two. In the first step (Figure 2(a)), the Lloyd algorithm produces the root node. The centroid of the entire training set will be the 0-bit codeword for the root node. In the next step, the Lloyd algorithm will split the root node into two as shown by the open circles in Figure 2(b). These seed nodes will then produce two children nodes that are 1-bit codewords. The training set is now separated into two with the centroids as the 1-bit codeword. In Figure 2(c), the two nodes are split again to generate four 2-bit codewords. From this step onwards, the Lloyd algorithm needs only be applied to the training set associated with the codeword that is generating more children nodes.

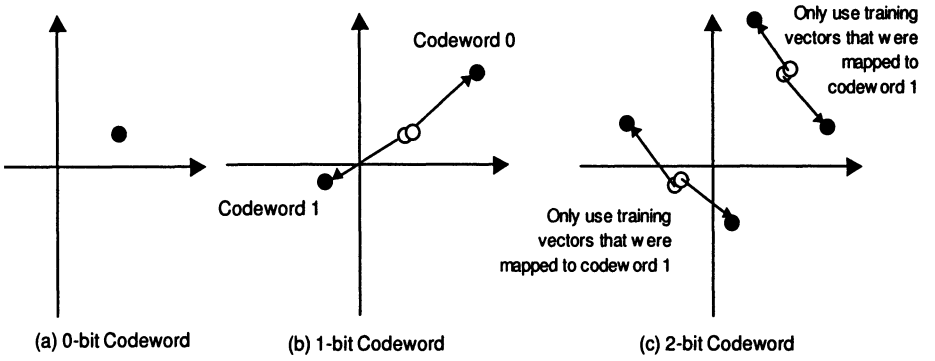


Figure 2. Example of Node Splitting in TSVQ

Two different techniques for choosing a node to split are employed in the Lloyd algorithm. The first method will find a node that has a high distortion (which is the average mean squared error of the training vectors). The split nodes are the current node and a slightly perturbed version of it. Upon failing to find a node with distortion, or when finding that one of the new children nodes has zero distortion, we use the second method. The split will be done by using the current node and the training vector that is farthest from it. The tree construction stops when a specified rate R is achieved, or when the number of training vectors associated with a node is less than the specified minimum vector parameter, or when the splitting of nodes is not possible.

There are two main advantages with using TSVQ for clustering:

1 To quantize a vector x, we begin by finding which of the two nodes originating from the root node has the closer testvector to x, then we find which of the two nodes stemming from this node has the closer testvector. We arrive at a terminal node after DR binary decisions. By increasing R, we have more levels of decisions leading to codewords with greater discriminating powers. This adaptive selection of bits allows more clusters and thus leads to better partitioning of densely populated space.

2 TSVQ is a very fast method for building clusters. It requires only $2DR$ distortion computations while constructing the quantizer, compared to 2^{DR} for an unstructured quantizer.

Each terminal node's codeword is the centroid of the training vectors associated with it. To translate the tree into clusters, only the terminal nodes are used. Each terminal node forms a cluster, the codeword gives the centroid of the cluster, and the node's training vectors are objects in the cluster. While the rate R controls the number of clusters formed by controlling the bits of the codeword, there are many instances when only one training vector is associated with a terminal node. A more effective way to control the number of objects in a cluster is to tune the minimum vector parameter, which controls the number of objects associated with a terminal node. The formal TSVQ algorithm is given in Figure 3.

- **Inputs**: *stopping_rate*, *training_vectors*;
- **Output**: *Cluster*;
- **Variables**:

```
float  rate;
struct node_type {
  *vect; /* Vector set associated with the node */
  codeword; /* centroid of vector set */
  distortion; /* average mean squared error of vector set */
} *Tree, *nodelist, root, next_node;
```

- **Algorithm**:

1: Init Tree:

```
root.vect ← training_vectors;
root.codeword ← centroid(root.vect);
root.distortion ← avg_mean_sq_error(root.vect);
Tree ← root;
```

2: *Lloyd*(*root*); /* Split node and do GLA */

3: *nodelist* ← *root*; /* Insert root into list of nodes to split */

4: while ($rate \leq stopping\_rate$)

- *next_node* ← *get_next_node*(); /* Choose node to split from nodelist. If the node's vector set is less than a specified number or no node is found, set distortion = 0 */
- if ($next\_node.distortion = 0.0$) break;
- $Tree \leftarrow (next\_node.left, next\_node.right)$; /* insert nodes into Tree */
- $rate \leftarrow update\_rate(next\_node)$;
- *Lloyd*(*next_node.left*); *nodelist* ← *next_node.left*;
- *Lloyd*(*next_node.right*); *nodelist* ← *next_node.right*;
- *remove_nodelist*(*next_node*);

5: for each $t \in (terminal\_nodes)$ /* Assign terminal nodes to clusters */

- *Cluster.id* ← *num* + +;
- *Cluster.centroid* ← (*t.codeword*);
- *Cluster.vect* ← (*t.vect*);

Figure 3. TSVQ Algorithm for Clustering.

3.2. Cluster Ranking Schemes

In this section, we describe two cluster ranking schemes. The first scheme is based on the commonly used cluster centroid; the second scheme makes use of both centroid and variance.

We store a cluster index containing the cluster ID and centroid in memory. To determine a query object q's proximity to a cluster, we measure the L_2 distance

from q to the cluster centroid c as follows:

$$d_c(q,c) = (\sum_{i=1}^{D}(q_i - c_i)^2)^{\frac{1}{2}} \quad \text{where } D \text{ is the dimension of features.} \qquad (1)$$

We call this the centroid scheme. We also evaluate the effect of incorporating cluster variance into our centroid ranking. Consider a cluster of objects in a two-dimensional space. The cluster will be more densely packed if the cluster variance is low. Thus it is likely to contain more nearest neighbors. The new distance with variance d_{cv} is computed as follows:

$$d_{cv}(q,c) = \frac{d_c(q,c)}{\sigma} \quad \text{where } \sigma \text{ is the cluster variance.} \qquad (2)$$

We rank a cluster highly when its distance (d_c or d_{cv}) from the query object is comparatively small. The top-1 cluster will be one with the shortest distance.

3.3. Similarity Search and Feature Weighting

Given a query object, a similarity search proceeds in two steps:

1 We compute the distance from the query object to the clusters. The definition of distance takes different forms depending on the type of cluster ranking scheme used. A highly-ranked cluster is closer to the query object and hence more likely to contain similar objects called nearest neighbors (NNs). A cluster list contains the ranked cluster sorted in descending order.

2 We read the top-ranked cluster from disk into memory and perform a sequential scan on the cluster to find the top-k NNs. If more NNs are desired, we can read additional clusters from the cluster list.

Theoretically, having more features should allow us to distinguish between objects more distinctly and hence ease the task of similarity search. However, finding meaningful features to encode an object is difficult. Very often, we end up with a sizable number of irrelevant features that only add noise during a similarity search. One popular solution is to use a weighted distance function as follows:

$$d(q,x) = (\sum_{i=1}^{D} w_i \times (q_i - c_i)^2)^{\frac{1}{2}}, \qquad (3)$$

where w_i is the weight for each feature i and $\sum_i^D w_i = 1$, D is the dimension of features, q is the query and x is any object. The weights w_i can be changed dynamically through relevance feedback. To deal with dynamic weightings, we adjust the cluster ranking schemes slightly. When comparing distance between the query point and clusters, instead of using Equation 1, we use Equation 3, which takes into account the weights w_i for each feature.

4. EMPIRICAL STUDY

Given a query, we perform a similarity search to return the k most similar objects, or k nearest neighbors ($k-$NN). We first establish a benchmark by scanning

the entire dataset to find the top-k NNs for each query; this constitutes the "golden" results set. The metric we use to measure the search result is *recall after X IOs*. In other words, we are interested in what fraction of top-k golden results are retrieved after X IOs are performed.

Our experiment consists of three parts:

1. We first evaluate the two cluster ranking schemes. Given a query, the objective is to achieve the highest recall with the minimum number of IOs.
2. We then investigate the effect of cluster size on recall and IO time.
3. We evaluate two feature weighting schemes and study their impact on recall.

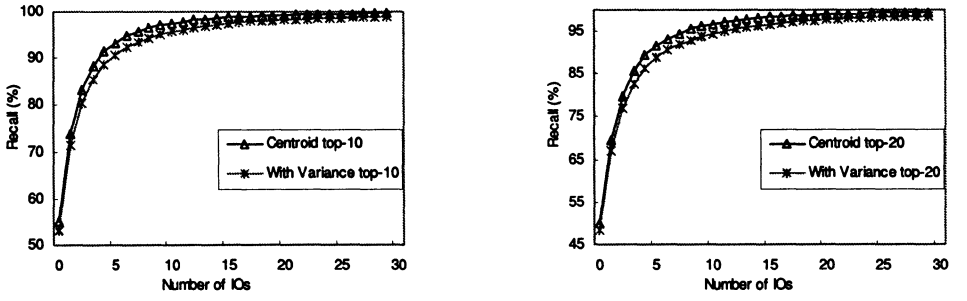
We perform our experiments on two sets of images:

- **51K-image-dataset**. From Corel Image CDs, we select 51,000 images to cluster. Images from this collection are widely used by the computer vision and image processing research communities. We then randomly select an additional 1830 separate images from the CDs to be used as the query set.
- **1-million-image dataset**. The second set is generated from a base of 70,000 images that includes images from Corel CDs and the Internet. We perform 24 transformations on these images. The transformations include rotation, cropping, scaling up and down, down-sampling and GIF-to-JPEG conversion. We then select one million images to cluster, and from the remaining ones, choose 1000 images to be used as queries. Due to space limitation, we do not present the results of this dataset. An extended version of this paper can be found at *mmdb.ece.ucsb.edu/~kingshy/vdb6-extended.pdf*.

Each image has 144 features: 108 for the 11 colors and 36 for textures [8].

4.1. Cluster Ranking Schemes

We use the 51,000−image set to evaluate each of the ranking schemes. We first apply TSVQ to the image set to obtain396 clusters with sizes ranging from 17 to 199. The clusters are then ranked by the schemes described in Section 3.2. We compute the accumulated recall, from the top−30 clusters, for 10−NN and 20−NN searches.



(a) 10−NN (b) 20−NN

Figure 4. Recalls of Centroid and Variance Ranking Scheme on 51K Image Set

The centroid scheme computes the L_2 distance from the query to the centroids of the clusters. The clusters are then sorted by distance in ascending order. Figure 4 compares the 10−NN and 20−NN recall of using d_c and d_{cv} to rank the clusters. The x-axis shows the number of IOs that are performed to achieve a particular recall. Given one IO, we retrieve the cluster with the smallest centroid to query distance d_c and achieve recall of 55% for 10−NN and 50% for 20−NN. After three IOs, the recall improves to 83% and 80% respectively. As more clusters are retrieved, the rate of improvement slows down.

In the same figure, we see that using variance degrades the recall results by about 2%. We believe that the variance estimate with a relatively small number of training instances in a high-dimensional space may not accurately characterize the density of objects in the cluster. (We use centroids in subsequent experiments.)

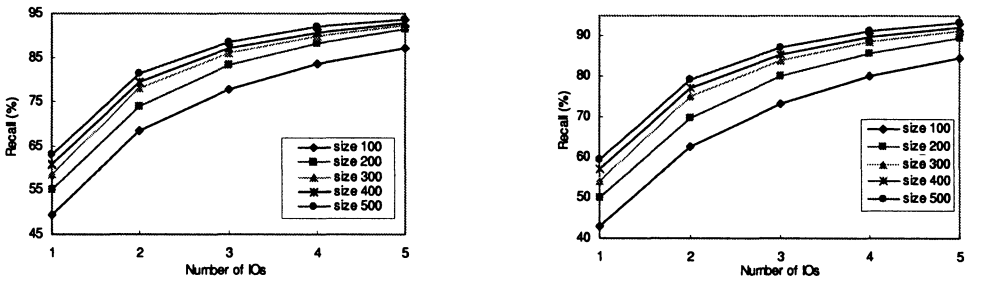
4.2. Effect of Cluster Size

In this section, we evaluate the importance of cluster size on recall using centroid and SVM schemes. We control the size of the clusters by varying the minimum vector parameter to produce cluster sizes shown in Table 1.

Minimum Vector Parameter	100	200	300	400	500
Number of Clusters	710	396	265	199	155
Average Size	69	128	191	255	327
Cluster Size (kBytes)	120	225	336	448	575

Table 1. Cluster Sizes for 51k-image Dataset

Figure 5 shows the effect of cluster size for 10−NN and 20−NN. For both the centroid and SVM schemes, we see that having a larger cluster size aids recall. By changing the cluster size from 69 to 327, the recall after one IO changes by 14% for the centroid scheme and 18% for the SVM. With larger clusters, the top−1 cluster containing the top−1 nearest neighbor is likely to enclose more of the additional nearest neighbors. The same is also true for subsequent retrieved clusters; hence the overall recall improves.



(a) Centroid 10−NN (b) Centroid 20−NN

Figure 5. Recalls for various Cluster Size

The penalty of larger cluster size is in IO time. We use a quantitative model to compute IO time. Let C be the cluster size in bytes, N be the number of IOs,

TR be the transfer rate, and T_{seek} be the average disk seek time. The IO time is estimated as

$$T = N \times T_{seek} + \frac{C \times N \times 8}{TR} \quad (4)$$

Assuming a transfer rate TR of 130Mbps, seek time T_{seek} of $14ms$, the IO times for each set of clusters are plotted in Figure 6. We observe that to achieve 60% recall, we need only to retrieve a small number of clusters, hence having a smaller cluster size translates to a faster retrieval. However, as the recall increases, we need to retrieve fewer clusters if the cluster size is large. This translates to a faster IO time. At recall of 95%, cluster sizes of 128 and 327 give the minimum IO time.

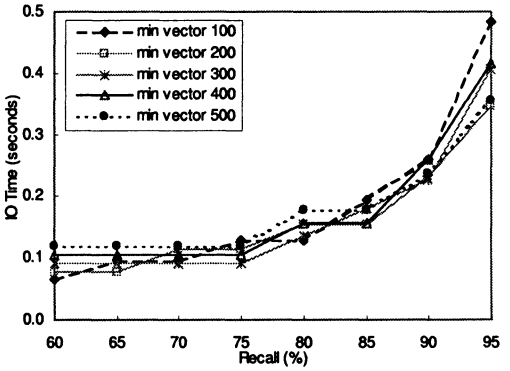


Figure 6. IO Time vs. Recall for 51K-image Dataset (10−NN)

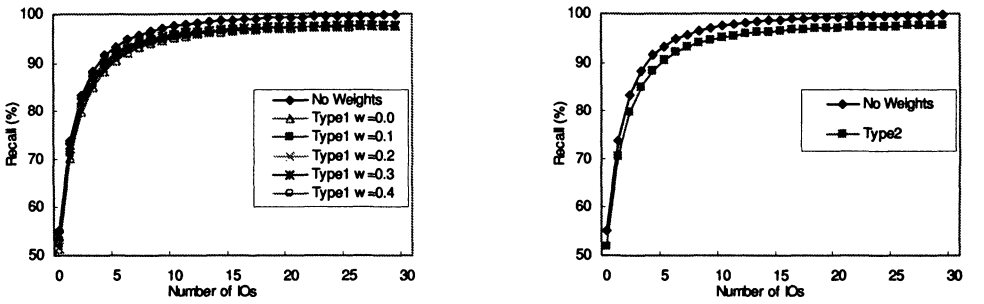
4.3. Feature Weighting

Our goal is to evaluate the feasibility of using feature weightings for a similarity search on clusters built without weightings. Using the centroid cluster ranking with L_2 distance computed as shown in equation 3, we assign the weights in two ways:

1. Binary Weights. Two values are used as weights. We choose N features randomly and assign W to them. The remaining $(D - N)$ features are given We evaluate the recall for $W = \{0.0, 0.1, 0.2, 0.3, 0.4\}$ and $N = 44$.

2. Linear Weights. Values between 0 to $(D - 1)$ are assigned randomly to D features.

After assigning the weights, we normalize them such that $\sum_i^D w_i = 1$. The weights are then accordingly reassigned to the features. We run the similarity search for each query object 10 times, then compute the average recall. Figure 7(a) shows the recall versus IO result when using binary weights with the centroid scheme. When we plot the recall using no weights, we find that using weights only reduces the recall by 1% to 4% depending on the value of W. The reduction for using linear weights is about 4% (Figure 7(b)). We get only a slight degradation of recall when features are weighted, despite the fact that the clusters are constructed on features without weights. This means that one indexing structure will suffice to support relevance feedback.



(a) Binary Weights (b) Linear Weights

Figure 7. 10−NN Recall Using Feature Weightings for 51K-image Dataset

5. CONCLUSIONS

We have presented a clustering approach using TSVQ to index data in high-dimensional spaces. We are able to support efficient similarity searches as well as relevance feedback via feature weightings. Using images as our testbed, we cluster similar images together and store them in a sequential file. A similarity search is modeled as a classification problem. We identify the cluster to which the query image belongs, and retrieve that cluster to search for similar images. We show that the use of a cluster centroid to classify the query image yields good recall. More importantly, we show that our approach is able to support relevance feedback, which is equivalent to modifying the features' weight. Using clusters constructed without feature weightings, our experiments show that the addition of weights during a similarity search lowers recall by only 1% to 4%. Lastly, we have shown that our approach is scalable to a 1−million-image dataset.

We are extending this work in several directions. First, we are evaluating additional cluster ranking schemes that use classification accuracy boosting. Next, we are extending our scheme to index data residing in a non-metric space.

REFERENCES

[1] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. *Proceedings of ACM SIGMOD*, 1998.

[2] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Proc. of the 5th SODA*, 1994.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *Proceedings of ACM SIGMOD*, May 1990.

[4] S. Berchtold, C. Bohm, and H.-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. *ACM Sigmod*, May 1998.

[5] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-Tree: An index structure for high-dimensional data. *Proceedings of the 22nd VLDB*, August 1996.

[6] S. Brin and H. Garcia-Molina. Copy detection mechanisms for digital documents. *Proceedings of ACM SIGMOD*, May 1995.

[7] E. Chang, B. Li, and C. Li. Toward perception-based image retrieval (extended version). *UCSB Technical Report*, February 2000.

[8] E. Chang, B. Li, and C. Li. Towards perception-based image retrieval. *IEEE Content-Based Access of Image and Video Libraries*, pages 101–105, June 2000.

[9] E. Chang, J. Wang, C. Li, and G. Wiederhold. RIME - a replicated image detector for the www. *Proc. of SPIE Symposium of Voice, Video, and Data Communications*, November 1998.

[10] P. Ciaccia and M. Patella. Pac nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. *Proceedings of ICDE*, pages 244–255, 2000.

[11] P. Ciaccia, M. Patella, and P. Zezula. M-Tree: An efficient access method for similarity search in metric spaces. *Proceedings of the 23rd VLDB*, August 1997.

[12] K. Clarkson. An algorithm for approximate closest-point queries. *Proceedings of the 10th SCG*, pages 160–64, 1994.

[13] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. *Proceedings of the 2nd International Conference on Knowledge Discovery in Databases and Data Mining*, August 1996.

[14] M. Flickner, H. Sawhney, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: the QBIC system. *IEEE Computer*, 28(9):23–32, 1995.

[15] H. Garcia-Molina, S. Ketchpel, and N. Shivakumar. Safeguarding and charging for information on the internet. *Proceedings of ICDE*, 1998.

[16] H. Garcia-Molina, J. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 1999.

[17] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Proc. of ACM SIGMOD*, June 1984.

[18] G. R. Hjaltason and H. Samet. Ranking in spatial databases. *Proceedings of the* 4^{th} *SSD*, pages 83–95, August 1995.

[19] P. Indyk, A. Gionis, and R. Motwani. Similarity search in high dimensions via hashing. *Proceedings of the 25th VLDB*, September 1999.

[20] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. *Proceedings of the 30th STOC*, pages 604–13, 1998.

[21] N. Katayama and S. Satoh. The SR-Tree: An index structure for high-dimensional nearest neighbor queries. *Proceedings of ACM SIGMOD*, May 1997.

[22] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. *Proceedings of the 29th STOC*, 1997.

[23] R. Kohavi, P. Langley, and Y. Yun. The utility of feature weighting in nearest-neighbor algorithms. *Proc. of the European Conference on Machine Learning*, 1997.

[24] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. *Proceedings of the 30th STOC*, pages 614–23, 1998.

[25] K.-L. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: an index structure for high-dimensional data. *VLDB Journal*, 3(4), 1994.

[26] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. *Proceedings of the 20th VLDB*, September 1994.

[27] J. T. Robinson. The K-D-B-Tree: A search structure for large multidimensional dynamic indexes. *Proceedings of ACM SIGMOD*, April 1981.

[28] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. *ACM Sigmod*, 1995.

[29] G. Sheikholeslami, S. Chatterjee, and A. Zhang. Wavecluster: A multi-resolution clustering approach for very large spatial databases. *Proceedings of the 34th VLDB Conference*, 1998.

[30] J. R. Smith and S.-F. Chang. Visualseek: A fully automated content-based image query system. *ACM Multimedia Conference*, 1996.

[31] D. A. White and R. Jain. Similarity indexing: Algorithms and performance. *Proc. SPIE Vol.2670, San Diego*, 1996.

[32] D. A. White and R. Jain. Similarity indexing with the SS-Tree. *Proc. of the 12th ICDE*, 1996.

[33] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. *Proceedings of ACM SIGMOD*, June 1996.