Quantize and Conquer: A dimensionality-recursive solution to clustering, vector quantization, and image retrieval

Yannis Avrithis
National Technical University of Athens

Abstract

Inspired by the close relation between nearest neighbor search and clustering in high-dimensional spaces as well as the success of one helping to solve the other, we introduce a new paradigm where both problems are solved simultaneously. Our solution is recursive, not in the size of input data but in the number of dimensions. One result is a clustering algorithm that is tuned to small codebooks but does not need all data in memory at the same time and is practically constant in the data size. As a by-product, a tree structure performs either exact or approximate quantization on trained centroids, the latter being not very precise but extremely fast. A lesser contribution is a new indexing scheme for image retrieval that exploits multiple small codebooks to provide an arbitrarily fine partition of the descriptor space. Large scale experiments on public datasets exhibit state of the art performance and remarkable generalization.

1. Introduction

We often visualize a clustering process in two dimensions as in Figure 1, where a number of centroids partition the underlying space into Voronoi cells. Even with k-means, which is arguably the fastest alternative at large scale, the cost is dominated by the assignment of data points to the nearest centroid. It is thus popular to solve this subproblem by approximate search [20]. In the 2D discrete space of Figure 1, one may envision solving first the inverse problem of computing a distance map on the entire 2D grid, which could then respond to assignment queries by lookup.

By analogy, one may envision image retrieval as a propagation process on this grid, where query descriptors serve as source points and a local distance map is generated around these points. Indexed images have their descriptors distributed on the grid and only those at a specific range from source points are retrieved. Weighting of points is possible based on the distance to nearest query point, as specified by the position on the grid where they are found.

But how about spaces of up to 128 dimensions as in the case of SIFT descriptors? Unfortunately, the number

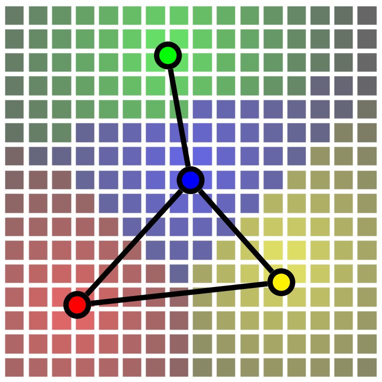

Figure 1. Clustering and space partitioning, visualized on 2D discrete space. Coloring of Voronoi cells follows that of the corresponding centroid; patch intensity follows the distance map.

of grid positions increases exponentially in the number of dimensions, which prevents us from visiting or even representing the entire space. This is exactly our contribution in this work: we use a 2D discrete grid not just as an analogy but to actually solve clustering or search problems in higher-dimensional spaces. The key idea is that the grid actually represents a $2d$-dimensional space S. The two “dimensions” that we see in fact capture the discrete topology of two subspaces S^L, S^R, each of d dimensions, that decompose S into a Cartesian product $S = S^L \times S^R$.

In a *clustering* setting, and assuming that we see centroids as point sources and do compute a distance map via propagation from the sources to the entire grid, it is possible to obtain a triangulation as a by-product, having the cluster centroids as vertices as in Figure 1. The graph representing this triangulation captures exactly the discrete topology of the space. Doing this for both S_L and S_R, we may apply the same idea to S, ending up with an algorithm that is recursive in the number of dimensions.

In a *retrieval* setting, we do not even need a single codebook for the entire descriptor space. We may start recursion after decomposing *e.g.* into two or four subspaces, of dimension 64 or 32 respectively for SIFT descriptors.

This corresponds to splitting each vector to *e.g.* two or four subvectors, each or which is assigned to a smaller codebook [12]. Searching is then possible by first finding nearest neighbors in each individual codebook [2]. We solve this problem recursively as well.

2. Related work and contribution

Following the success of bag-of-words image retrieval over fine codebooks [20], several attempts have been made to alleviate loss incurred by quantization, including *e.g.* multiple assignment [21], learning even finer codebooks [14], or embedding more descriptor information [10].

Progress in nearest neighbor search has helped in this direction: *product quantization* [12, 13] makes it possible to approximate descriptor distances with a rather small space overhead and the *inverted multi-index* [2] turns this idea into indexing, inducing a very fine partition over the descriptor space via multiple smaller codebooks. Multi-indices are not new: [18] studies multi-index hashing for fast search in Hamming space, and in this framework there are much earlier studies like [7].

At the other extreme of a single codebook, *scalar quantization* [4] offers a lightweight alternative that cannot be turned into indexing. Other approaches decomposing or structuring the underlying space exist, *e.g.* separable dictionaries in the form of a Kronecker product [8]. All such approaches decompose the input space only once and do not explore the potential benefit of a hierarchy.

What has captured our attention is that both [12] and [2] require a number of sub-quantizers that are assumed to be trained by an independent clustering algorithm, as well as a way to search into the small individual codebooks, possibly using an independent nearest neighbor search algorithm. On the other hand, it is known that clustering for large codebook construction benefits by approximate nearest neighbor (ANN) search, *e.g.* in the assignment step of approximate k-means, requiring an independent ANN algorithm *e.g.* randomized k-d trees [15].

Our work provides a deeper investigation into this connection. Using a second order index makes search appear much like local distance propagation on a 2D grid, except that raw/column ordering on the grid are local, *i.e.* they depend on the query. During clustering, one requires millions of queries to the very same points: the cluster centroids. If the codebooks and consequently the grid are small compared to the number of data points, it makes sense to **propagate from centroids to the entire grid rather than to make queries in the opposite direction**. Based on this idea, we make a number of contributions.

1. We introduce a new **clustering** paradigm where nearest neighbor search is handled by distance propagation on a grid. We devise a dimensionality-recursive variant of k-means that is *self-contained*, *i.e.* it does not need another algorithm for any sub-problem.

2. We exploit the recursive nature of the algorithm to formulate a tree structure that provides approximate or exact **vector quantization**. The former is used during training and the latter for indexing of new data. It is probably the first time to perform vector quantization *by lookup* in up to 64 dimensions.

3. We explore the use of a higher-order index in the context of **image retrieval** where we suggest a simple search alternative to existing solutions that scales to effective codebook size up to 2^{56} and multiple assignment neighborhoods of up to 2^{28} cells.

Cartesian k-means [17] is another k-means variant on a Cartesian product but involves no hierarchy and rather focuses on optimizing subspace selection and rotation, both represented by an orthogonal matrix. *Optimized product quantization* [6] develops the same idea independently, but also offers a parametric solution that optimizes prior to clustering, which could be combined with our work.

State of the art work on image retrieval focuses on post-processing and re-ranking methods to improve precision, for instance query expansion [5, 1], geometry [20, 19, 25], feature augmentation [27, 1], or k-NN re-ranking [25, 22]. Such studies are beyond the scope of this work, which focuses on potential improvements from codebook design and descriptor nearest neighbor search. More relevant are methods embedding descriptor information like [11] or the more recent [23, 26]. These are largely complementary, since our primary contribution concerns off-line processes.

3. Dimensionality-recursive clustering

The basic procedural part of our clustering algorithm is k-means. This includes random initialization for a predetermined number of centroids, as well as iterative assignment of points to centroids and centroid update to optimally represent the underlying points. We adopt a bottom-up description, from the one-dimensional base case to recursion in higher dimensions. While the former appears like a simple problem, it actually prepares the ground for the latter.

3.1. The base case: one dimension

Given a set X of N data points lying on a bounded interval $I = [a, b)$ of $\mathbb{R}$ and a target number $K > 1$ of centroids, we first construct a uniform partition $\{x_0, \ldots, x_B\}$ of I with $B \gg K$ and $x_i = a + \ell i$ for $i = 0, \ldots, B$, where $\ell = (b - a)/B$. We thus form a set $\mathcal{B} = \{b_0, \ldots, b_{B-1}\}$ of subintervals called bins, where each bin $b_i = [x_i, x_{i+1})$ is of constant length ℓ. By *scalar quantization*, we allocate each point $x \in X$ to bin $b_{s(x)} \in \mathcal{B}$ with

$$s(x) = \lfloor (x - a)/\ell \rfloor . \quad (1)$$

Let $Z = \{z_0, \dots, z_{B-1}\}$ denote the set of midpoints of subintervals in $\mathcal{B}$. Using Z as a discrete representation of interval I, the above is equivalent to approximating each $x \in X$ through a map $h : I \to Z$ with

$$x \mapsto h(x) = z_{s(x)} = a + \ell s(x) + \ell/2. \tag{2}$$

Denoting by $X_i = X \cap b_i = \{x \in X : s(x) = i\}$ the set of data points allocated to bin b_i, we empirically measure a finite probability distribution f, where

$$f_i = |X \cap b_i|/N = |X_i|/N \tag{3}$$

is the normalized histogram frequency of data in bin b_i, for $i = 0, \dots, B-1$. In fact, we may now discard the data; the distribution contains all the information we need for the remaining steps. This approximation may seem unnecessary, but is a critical component of the recursive case.

We **initialize** by sampling K points out of Z with replacement, according to distribution f. Let $C = \{c_0, \dots, c_{K-1}\}$ be the set of distinct sampled points; this is the initial set of centroids, which we represent as a sequence in ascending order. It is straightforward to see that it will remain so during course of the algorithm.

The set of centroids determines a *quantizer*, i.e. a function $q : I \to C$ that maps a point to the nearest centroid,

$$x \mapsto q(x) = \arg\min_{c \in C} \|x - c\|. \tag{4}$$

The **assignment step** involves only the restriction $q^* : Z \to C$ of q to Z, i.e., computing $q(z)$ and storing it as $q^*[z]$ for all $z \in Z$. This is achieved by partitioning Z into Voronoi cells, one for each centroid. The (discrete) Voronoi cell V_k of centroid $c_k \in C$ is the set of points z quantized to c_k,

$$V_k = \{z \in Z : q(z) = c_k\}. \tag{5}$$

Literally computing Voronoi cells is an easy task in one dimension: because centroids are ordered, we find all midpoints between successive centroids. If m_k is the midpoint of interval $[c_{k-1}, c_k)$ for $k = 1, \dots, K-1$ and we define $m_0 = a$, $m_K = b$, then for all $k = 0, \dots, K-1$,

$$V_k = Z \cap [m_k, m_{k+1}), \tag{6}$$

and for all $z \in V_k$, we assign $q^*[z] \leftarrow c_k$. This operation is clearly linear-time in $|Z| = B$.

The **update step** for centroid $c_k \in C$ simply requires weighted averaging of points over its Voronoi cell V_k,

$$c_k \leftarrow \sum_{i : z_i \in V_k} f_i z_i. \tag{7}$$

At termination, we approximate $q(x)$ for $x \in X$ (section 4.1) and construct a graph $G = \{C, E\}$ with edges $E = \{(c_k, c_{k+1})[1] : k = 0, \dots, K-2\}$ between successive centroids, representing a neighborhood system over I.

[1]Here (c_k, c_{k+1}) denotes a pair, not an open interval.

3.2. Recursion

Recursion assumes that a space S is decomposed into two subspaces that have been clustered, each producing a set of centroids, a set of labels for quantized data points, and a graph representing its topology. Based on this subspace information, we cluster S and produce exactly the same information for it. What we are actually doing is **learning a joint distribution from two marginal ones**.

More formally, assume a $2d$-dimensional space S decomposed into a product $S^L \times S^R$ of d-dimensional subspaces S^L, S^R, a set X of N data points lying on an interval $I = I^L \times I^R$ of S, and a target number $K > 1$ of centroids in I. Let x^L, x^R be the projections of x onto S^L, S^R respectively, that is, $x = (x^L, x^R)$[2] for $x \in X$. Also assume that the corresponding sets of projected points X^L, X^R have been clustered, giving rise to two sets of centroids C^L, C^R, each of cardinality J. Assume as well that each projected point x^L (respectively, x^R) has been quantized to $q^L(x^L) \in C^L$ (respectively, $q^R(x^R) \in C^R$). Finally, assume that two graphs $G^L = \{C^L, E^L\}$ and $G^R = \{C^R, E^R\}$ representing neighborhood systems over I^L, I^R respectively are available. Accordingly, a graph $G = \{C, E\}$ representing a neighborhood system over I is to be computed as a by-product of clustering X.

Let $Z = C^L \times C^R$ be a grid of $B = J \times J$ points in S. As in the one-dimensional case, we see Z as a discrete representation of I giving rise to a set of bins, and we approximate each $x \in X$ via a map $h : I \to Z$ as

$$x \mapsto h(x) = (q^L(x^L), q^R(x^R)). \tag{8}$$

If Z is written as $\{z_0, \dots, z_{B-1}\}$, let f denote the finite probability distribution as measured empirically by normalized frequencies of data points into bins, such that f_i is the probability of z_i. We then **initialize** centroids $C = \{c_0, \dots, c_{K-1}\}$ by sampling K points out of Z with replacement, according to distribution f.

The problem in the **assignment step** is to quantize each point $z \in Z$ to the nearest centroid $q(z) \in C$. This is certainly harder than (6) in the one-dimensional case and is in fact the bottleneck of the algorithm. We leave this discussion for section 3.3. The **update step** is identical to (7). At termination, we map centroids to the nearest points in Z (via (8)) and approximate $q(x)$ for all $x \in X$ (section 4.1). Graph G is computed once at the final assignment step, also discussed in section 3.3.

Discussion and analysis. This new k-means variant is called *dimensionality-recursive clustering* (DRC). It can be applied recursively on the number of dimensions to cluster points in $\mathbb{R}^D$ where D is a power of 2 or indeed any integer, with slight modifications. This takes $\log_2 D$ levels of recursion and its output is not merely a set of centroids

[2]x^L, x^R are d-tuples; (x^L, x^R) is their concatenation into a $2d$-tuple.

and a corresponding set of labels for the data points, but a tree structure that can perform approximate or exact nearest neighbor search over the centroids, as discussed in section 4. DRC may be generalized to any space that can be hierarchically decomposed into Cartesian products of subspaces where clustering can be solved more easily.

An interesting aspect of this new clustering paradigm is that not all data are required in memory at the same time: whenever probability distribution f is available over Z, we are free to discard labels $q^L(x^L), q^R(x^R)$, while data points x^L, x^R are never actually stored. Given N D-dimensional data points, the space needed for data is $O(N)$ instead of $O(ND)$. On the other hand, $O(K^2)$ space is needed to represent grid Z, and this limits the size of produced codebooks. K is assumed to increase with dimensionality d.

3.3. Propagation

The problem of the assignment step is to compute $q(z)$ (4) and store it as $q^*[z]$ for all $z \in Z = C^L \times C^R$. This is equivalent to computing a distance map over Z with C as source points, and can be solved efficiently by distance propagation using a fast marching method [24]. However, domain Z is not a 2D space here. The underlying space topology is represented by graphs G^L, G^R, which in fact describe triangulations over C^L, C^R respectively. Hence a fast marching method over a triangulated domain applies [3], except for the fact here we have a product of two such domains.

Our algorithm, called *product propagation* (PP), is reminiscent of Dijkstra's algorithm as any fast marching variant and is outlined in Algorithm 1. The key data structure is a min-priority queue Q that lets us visit each point $z \in Z$ exactly once in ascending order of the distance to the nearest centroid. This distance is maintained as property *dist*$[z]$ for all z and is the KEY value associated to z for Q.

To initialize, we map each centroid $c \in C$ to $h(c)$ as given by (8) and use all mapped points as sources, to enter Q first. At each iteration, it is the underlying graphs that guide exploration of the grid: given the current point $z = (z^L, z^R)$, we examine neighbors $E^L(z^L), E^R(z^R)$ in turn and propagate accordingly.

Centroids are sampled on the grid initially and mapped again to the grid at termination. But during k-means iterations, they are arbitrary points in space S as computed by (7). To measure the *exact distance* of a given point $x = (x^L, x^R) \in S$ to a point on $z = (z^L, z^R)$ the grid, we assume that the underlying codebooks can compute squared Euclidean distances $\delta^L(x^L, z^L)$, $\delta^R(x^R, z^R)$ for the projected points. Then, the required (squared) distance is

$$\delta(x, z) = \delta^L(x^L, z^L) + \delta^R(x^R, z^R). \tag{9}$$

Since centroids remain constant during propagation, required distances $\delta(c, z)$ are efficiently found via (9) by looking up precomputed values of δ^L, δ^R.

Algorithm 1: Product propagation

```
1  function $(q^*, E) \leftarrow \textsc{pp}(C, Z, h, \delta; E^L, E^R, \tau)$
2      $E \leftarrow \emptyset$; initialize queue $Q$
3      for $z \in Z$ do $state[z] \leftarrow$ ALIVE                ▷ initialize state
4      for $c \in C$ do PUSH$(c, h(c))$                       ▷ initialize sources
5      while $\neg Q$.EMPTY() do
6          $z \leftarrow Q$.EXTRACT-MIN()
7          $state[z] \leftarrow$ FAR; $c \leftarrow q^*[z]$
8          for $y \in E^L(z^L)$ do SCAN$(c, (y, z^R))$
9          for $y \in E^R(z^R)$ do SCAN$(c, (z^L, y))$
10     return $(q^*, E)$

11 function SCAN$(c, z)$
12     if $state[z] =$ ALIVE then PUSH$(c, z)$
13     if $state[z] =$ CLOSE then RELAX$(c, z)$
14     if $state[z] =$ FAR then JOIN$(c, z)$

15 function PUSH$(c, z)$
16     $dist[z] \leftarrow \delta(c, z)$; $q^*[z] \leftarrow c$
17     $Q$.INSERT$(z)$; $state[z] \leftarrow$ CLOSE

18 function RELAX$(c, z)$
19     $d \leftarrow \delta(c, z)$
20     if $d < dist[z]$ then
21         $dist[z] \leftarrow d$; $q^*[z] \leftarrow c$
22         $Q$.DECREASE-KEY$(z, d)$

23 function JOIN$(c, z)$
24     if $\delta(c, z) + dist[z] < \tau$ then
25         $E \leftarrow E \cup (c, q^*[z])$                   ▷ update edges $E$
```

As a by-product, edges E of graph G are generated wherever two propagating fronts meet. As shown in Algorithm 1, edges can be updated during propagation. However, this is in fact not repeated during every k-means iteration; it simply occurs once at termination.

The bottleneck of the entire clustering algorithm is distance propagation: with a *binary heap* for the priority queue, the time complexity of propagation on a $K \times K$ grid is $O(eK^2 \log K)$, where e is the maximal degree of the graph. A *Fibonacci heap* yields $O(eK^2)$, but is not any faster in practice. To limit the queue length, we *prune* edges as shown in line 24, where threshold τ is specified as a fraction of the average distance of all centroids to all bins. There is no guarantee that the entire grid will be explored at termination under pruning, but in practice we have verified that with $\tau = 0.35$, all grid positions are indeed visited.

4. Dimensionality-recursive quantization

The outcome of clustering as described so far is a set of centroids, a set of data labels, and a graph representing a neighborhood system. But there is more than that. Clustering of one space relies on clustering of two underlying subspaces, and this recursive implementation gives rise to a tree structure: each produced codebook is a *node* in the tree and an one-dimensional codebook is a *leaf*. This hierarchy refers to subspace structure or dimensionality and not to locality or data size as in typical hierarchical approaches [16].

Each codebook is equipped with appropriate information to recursively respond to approximate or exact nearest neighbor queries over its centroids, simply by delegating queries to its child nodes and aggregating. The two options are separately discussed below. We refer to both as *dimensionality-recursive quantization* (DRQ).

4.1. Approximate quantization

In one dimension, a given new point x in interval I can be mapped to $z = h(x) \in Z$, exactly as we did during training (2). In turn, z is mapped to a unique centroid $q(z) \in C$, and $q^*[z]$ is stored for all $z \in Z$. Hence a leaf codebook can approximate $q(x)$ by $q(z) = q^*[h(x)] \in C$ via scalar quantization followed by lookup.

In the general $2d$-dimensional case, given a new point $x = (x^L, x^R) \in I$, the child codebooks can generate approximations of $q^L(x^L), q^R(x^R)$ respectively. This gives rise to a point $z = h(x)$ on the grid (8). The node has again $q^*[z]$ stored for all $z \in Z$, so it can approximate $q(x)$ by $q(z) = q^*[h(x)] \in C$ via simple lookup.

Vector quantization via a sequence of scalar quantization and lookup operations achieves unprecedented speed as we shall see in section 6. For a space of dimensionality D that is a power of 2, only D scalar quantizations and $2D-1$ lookups are needed. The time complexity, $O(D)$, is then constant in K. Alas, its precision is not adequate *e.g.* for labeling new vectors for retrieval applications. Still, this kind of vector quantization is enough for training purposes. This is exactly how we implicitly treat input data during the assignment step of k-means, and it renders training virtually constant in the data size.

4.2. Exact quantization

In one dimension, each leaf codebook stores the original K scalar centroids, so given a new point $x \in I$ it can respond with a K-vector of squared distances $\delta(x, c) = (x - c)^2$ of x to all centroids $c \in C$.

In the recursive case, given a new point $x = (x^L, x^R) \in I$, the node first requests from its child codebooks the squared Euclidean distances $\delta^L(x^L, z^L)$, $\delta^R(x^R, z^R)$, for all $z^L \in C^L$ and all $z^R \in C^R$. It then computes $\delta(x, c)$ according to (9) for all $c \in C$. At the root of the tree, x can be quantized as

$$q(x) = \arg\min_{c \in C} \delta(x, c). \tag{10}$$

The idea is similar to product quantization [12] which employs only one level of decomposition. We rather decompose from the original space dimension D down to scalars. The computation is self-contained because it does not require another algorithm for the sub-quantizers. It is exact because node centroids are not arbitrary vectors but quantized and stored as coordinates on the grid.

5. Image indexing and retrieval

Applied to nearest neighbor search or image retrieval, our approach is tuned to rather small codebooks that can however quantize subspaces of the target descriptor space. We focus on image retrieval, applying DRC and DRQ but choosing to start recursing a number of levels below the target dimension D, yielding a forest of quantizers.

5.1. Multi-indexing

We assume n root codebooks, each of J centroids, inducing a partition of the D-dimensional domain I into $B = J^n$ *cells*[3]. An input vector $x \in S$ is now split into n sub-vectors, each quantized separately by one of the codebooks. As it stands, this representation is the same with product quantization [12]. However, instead of storing quantized labels per input data, it is possible to invert the representation when n is small, actually storing input data per label.

This leads to multi-indexing, that is, encoding index cells by n different codes and storing data appearing within each cell; n is called the *order* or *dimension* of the index. For instance, the inverted multi-index [2] focuses on the second-order case $n = 2$, performing full inversion. As explained in section 6, we attempt larger n where full inversion is not possible because the effective codebook size J^n becomes prohibitive. For instance, $J = 4096 = 2^{12}$ and $n = 4$ yields a partition of $B = 2^{48}$ cells.

For this reason we follow *partial inversion*, that is, we marginalize this fine partition along one or more dimensions. In our experiments for instance, we keep J^2 cells for inverted indexing in two dimensions, and embed the labels for the remaining two dimensions along with data. With 12 bits required for each label, this takes 24 bits per data point in addition to the image id, which is only stored once per point. Varying J and n may give more options in the design of a large scale retrieval system.

5.2. Retrieval

Searching in a higher-order index is certainly more demanding than in a plain inverted file. Because of the extremely fine partition, multiple cells need to be looked up, in the spirit of *multiple assignment* [21]. That is, choose the k nearest neighbors per codebook for each query vector by the exact search (section 4.2), and then search among the k^n possible neighbor cell combinations. The cost is exponential in the order of the index.

[3]In the context of clustering in section 3, these are referred to as bins, while (Voronoi) cells are collections of bins, one per centroid. The terminology here is aligned to related work.

One solution is the *multi-sequence* algorithm of [2], a simplified version of distance propagation in two dimensions, which visits cells in ascending order of distance from the query cell. However, it needs to explicitly store state per cell and this is prohibitive for large k^n, especially for image retrieval where thousands of queries are needed.

Another solution is to store n separate indices and search each independently for only a fraction of the k neighbors, generating n candidate lists to be verified against each other [18]. This is faster and avoids the k^n storage per query point, but multiplies the index size by a factor of n.

We rather follow a simple scheme that we call *rank sum*: we pre-compute all n-tuples $\alpha = (\alpha_0, \ldots, \alpha_{n-1})$ with sum

$$|\alpha| = \sum_{i=0}^{n-1} \alpha_i \leq k \tag{11}$$

and use them to visit all neighbor combinations having sum of ranks up to k. This choice approximates the neighboring cells visited by the multi-sequence algorithm but avoids the k^n storage and is much faster especially for large n, because neighbors are precomputed.

Entries found in all neighboring cells are weighted by an asymmetric distance between uncompressed query vectors and quantized database vectors. Formally, given query $x = (x^0, \ldots, x^{n-1})$, each of the k-nearest neighbors $z_j^i \in C^i$ in codebook i for $j = 0, \ldots, k-1$, is associated with an exact squared Euclidean distance $\delta^i(x^i, z_j^i)$ for $i = 0, \ldots, n-1$. Then, the squared distance to neighboring cell

$$z_\alpha = (z^0_{\alpha_0}, \ldots, z^{n-1}_{\alpha_{n-1}}) \tag{12}$$

is given by

$$\delta(x, z_\alpha) = \sum_{i=0}^{n-1} \delta^i(x^i, z^i_{\alpha_i}), \tag{13}$$

similarly to (9). Finally, entries found in a cell at squared distance δ are weighted by $w(\delta) = e^{-\delta/\sigma^2}$ where σ is a scale parameter, similarly to [21].

In the case of partial indexing, we only get a partial sum of (13) from the position of a cell. This partial sum is completed per entry (image id) using codes embedded per entry in the index. In other words, the cell contains a list of candidate neighbors and sum (13) is used to verify whether an entry belongs to a true neighbor or not.

6. Experiments

Our main contribution refers to *off-line* processes, i.e. dimensionality-recursive clustering (codebook training) and vector quantization. *On-line* applications like nearest neighbor search and image retrieval mainly serve as validation. We focus on the latter in this work.

K	$\log_2 K_d$ for dimension d						time (m)
	1	2	4	8	16	32	
16K	6	7	8	9	11	14	129.96
8K	6	7	8	9	11	13	119.43
4K	6	7	8	9	10	12	20.07
2K	5	6	7	8	9	11	2.792
1K	5	6	7	8	9	10	2.608
512	4	5	6	7	8	9	0.866
4K	Approximate k-means [20]						504.2

Table 1. Codebook setup and training times for varying codebook size K. Codebook size K_d for dimension d is given as a power of two. *E.g.*, for $K = 16\text{K}$, we get $2^{14} = 16\text{K}$ (target codebook size) for $d = 32$, which is trained on a $2^{11} \times 2^{11} = 2048 \times 2048$ grid, since codebooks at the previous level $d = 16$ are of size 2^{11}. Times refer to $n = 4$ codebooks on the $N = 12.5\text{M}$ 128-dimensional SIFT descriptors of Oxford 5K.

6.1. Datasets and evaluation protocol

We apply our methods to specific object retrieval and evaluate on two public datasets, namely Oxford buildings [20] and Paris [21], containing 5062 and 6412 images, as well as 55 queries each. We train codebooks on both datasets and evaluate retrieval performance on the same or on different datasets. At larger scale, we also use the additional 100K distractor images provided with Oxford buildings. We refer to datasets as Oxford 5K / Paris 6K without distractors, and Oxford 105K / Paris 106K with distractors.

We use features detected with the modified Hessian-affine detector and SIFT descriptors of [19], using the same settings as in [19] including the gravity vector assumption, producing *e.g.* a set of 12.5M features in total for Oxford 5K. We normalize SIFT descriptors as in RootSIFT [1], that is, ℓ_1-normalize and take square root element-wise.

For vector quantization, we measure performance by Recall@R, *i.e.*, the proportion of queries for which the nearest neighbor is ranked in the first R positions [12]. For retrieval, we use the protocol of [20], measuring performance by mean Average Precision (mAP), where *good* and *ok* images are treated as correct and *junk* as if they are not in the database. All times refer to single-threaded C++ implementations on a 3GHz Core i7 CPU with 24GB RAM.

6.2. Results

Training. As discussed in the retrieval experiments, we choose to focus on fourth order indices, that is, we decompose the 128-dimensional SIFT descriptor space into $n = 4$ 32-dimensional subspaces. Table 1 shows the setup of the training process for varying target codebook size K. The size K_d of each child codebook increases with the dimensionality d of the underlying subspace. It is clearly seen that the training time depends explicitly on the codebook size at $d = 16$, which determines the size of the grid where the root codebook is trained.

K	16K	8K	4K	2K	1K	512
Approximate (μs)	0.95	0.83	0.80	0.73	0.80	0.90
Exact (ms)	1.19	0.79	0.51	0.26	0.21	0.11

Table 2. Vector quantization times per point for varying codebook size, averaged over the $N = 75$K SIFT descriptors of the 55 cropped query images of Oxford 5K.

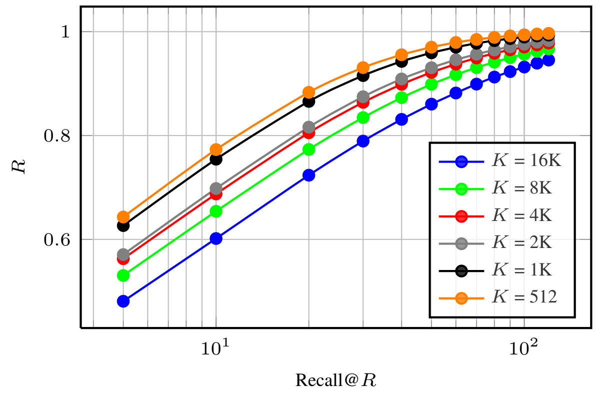

Figure 2. Recall@R performance of approximate vector quantization for varying codebook size, averaged over the query images of Oxford 5K.

The above reveals that the bottleneck of the algorithm is distance propagation on the grid. At the given sizes, training is much more efficient than *e.g.* approximate k-means [20] (25× faster); however at larger sizes it becomes impractical. It is interesting that training is independent of the data size, N. Averaging over 15 runs for $K = 4$K and N varying from 2.5M to 12.5M, we have found that training time does not increase with N.

Vector quantization. Table 2 shows average vector quantization times. Our *exact* quantization comes at a speed that offers a practical alternative over other approximate schemes, and this is exactly what we have used to label images for indexing. For instance, FLANN [15] takes 0.118ms per point on average at the same setup for a 4K codebook using 200 checks, corresponding roughly to a precision of 98%. Our *approximate*, lookup-based scheme offers unprecedented speed, but its performance is quite low as revealed in Figure 2. Although this is not adequate for labeling images, it is still appropriate for training.

Indexing. Our initial target has been a second order index. However, we have only achieved mAP performance up to 0.66 with codebooks of size up to 65K^2, where behavior is not much different than a standard inverted file and training times become an obstacle. Although there is still space for experiments, we have decided to move on to the unexplored area of a fourth order index with codebook size J^4.

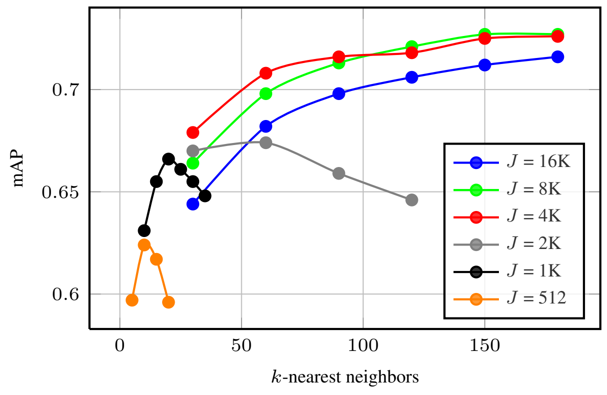

Figure 3. mAP performance versus k-nearest neighbors in our fourth order indexing scheme for varying codebook size on Oxford 5K, also trained on Oxford 5K.

Figure 3 shows that up to $J = 2$K, the behavior is similar to standard multiple assignment: mAP exhibits a peak and begins to drop due to additional distractor noise. The situation changes radically for larger codebooks, however. It appears that the space partition becomes so fine that mAP continues to increase for large number of neighbors k, and the only limit is the search time. We choose $J = 4$K because at $k = 90$ it outperforms the 8K codebook; and we choose $k = 90$ to keep query times below one second for all remaining experiments. At these settings, the average query time is 989ms on Oxford 5K.

Retrieval. We focus on performance evaluation relating codebooks, descriptor encoding and indexing. We compare to methods using multiple assignment or embedding additional descriptor information, but not other complementary re-ranking methods. We fix scale parameter σ^2 to 0.05, found to be optimal on all datasets. Table 3 compares our solution to the state of the art on different combinations of training and test sets or distractors.

We clearly outperform most methods. It is also remarkable that searching on the same or on a different dataset than the training one has little impact, unlike most known methods. [19] is superior at the Oxford5K/Oxford5K combination, but using a different test set is more important in general to avoid over-fitting behavior. [14] is superior when using alternative words, but this method is not really comparable as it employs large scale learning over a different training set of millions of images using geometry.

7. Discussion

We have investigated the relation between nearest neighbor search and clustering in high dimensional spaces, providing deeper insight and a new paradigm that may open new directions in numerous applications. We have shown that

Training set	Oxford 5K / other [*]				Paris 6K / other [*]		K	MA	Other
Test set	Ox5K	Ox105K	Pa6K	Pa106K	Ox5K	Ox105K			
This work	0.716	**0.657**	**0.696**	**0.584**	**0.703**	**0.640**	4K[4]	✓	
Perdoch *et al.* [19]	**0.717**	0.568	—	—	0.558	0.423	1M		
Arandjelovic *et al.* [1]	0.683	0.581	—	—	—	—	1M		
Shen *et al.* [25]	0.649	0.568	—	—	—	—	1M		
Philbin *et al.* [21]	0.614	0.498	—	—	0.403	0.290	1M		
Philbin *et al.* [21]	0.673	0.534	—	—	0.493	0.343	1M	✓	
Philbin *et al.* [20]	0.618	0.490	—	—	—	—	1M		
Jegou *et al.* [10]	—	—	—	—	0.615	0.516	200K	✓	HE, WGC
Jegou *et al.* [9]	—	—	—	—	0.647	—	20K	✓	HE, WGC
Mikulik *et al.* [14]	—	—	0.625*	0.533*	0.618*	0.554*	16M	✓	
Mikulik *et al.* [14]	—	—	**0.749***	**0.675***	**0.742***	**0.674***	16M	*	Learning

Table 3. mAP performance on different combinations of training and query / test sets, comparing our work to number of state of the art methods. K = codebook size. MA = multiple assignment. HE = Hamming embedding. WGC = weak geometric consistency.

a single recursive data structure is enough for all related problems, from codebook construction and database labeling, to indexing and search. In image retrieval, we have investigated higher order indices that offer remarkable generalization but do not scale well, hence mostly serve as a validation for our off-line solutions. A coarse/fine approach would be more practical as in [12]. Nearest neighbor search is another application we are currently investigating.

References

[1] R. Arandjelovic and A. Zisserman. Three things everyone should know to improve object retrieval. In *CVPR*, 2012. 2, 6, 8

[2] A. Babenko and V. Lempitsky. The inverted multi-index. In *CVPR*, 2012. 2, 5, 6

[3] T. J. Barth and J. A. Sethian. Numerical schemes for the Hamilton–Jacobi and level set equations on triangulated domains. *Journal of Computational Physics*, 145(1), 1998. 4

[4] J. Brandt. Transform coding for fast approximate nearest neighbor search in high dimensions. In *CVPR*, 2010. 2

[5] O. Chum, A. Mikulik, M. Perdoch, and J. Matas. Total recall II: Query expansion revisited. In *CVPR*, 2011. 2

[6] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. In *CVPR*, 2013. 2

[7] D. Greene, M. Parnas, and F. Yao. Multi-index hashing for information retrieval. In *FOCS*, 1994. 2

[8] S. Hawe, M. Seibert, and M. Kleinsteuber. Separable dictionary learning. Technical report, 2013. 2

[9] H. Jégou, M. Douze, and C. Schmid. On the burstiness of visual elements. In *CVPR*, 2009. 8

[10] H. Jégou, M. Douze, and C. Schmid. Improving bag-of-features for large scale image search. *IJCV*, 87(3), 2010. 2, 8

[11] H. Jégou, M. Douze, and C. Schmid. Exploiting descriptor distances for precise image search. Technical report, 2011. 2

[12] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *PAMI*, 33(1), 2011. 2, 5, 6, 8

[13] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *ICASSP*, 2011. 2

[14] A. Mikulik, M. Perdoch, O. Chum, and J. Matas. Learning vocabularies over a fine quantization. *IJCV*, 2012. 2, 7, 8

[15] M. Muja and D. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *ICCV*, 2009. 2, 7

[16] D. Nister and H. Stewenius. Scalable recognition with a vocabulary tree. In *CVPR*, 2006. 5

[17] M. Norouzi and D. Fleet. Cartesian k-means. In *CVPR*, 2013. 2

[18] M. Norouzi, A. Punjani, and D. J. Fleet. Fast search in Hamming space with multi-index hashing. In *CVPR*, 2012. 2, 6

[19] M. Perdoch, O. Chum, and J. Matas. Efficient representation of local geometry for large scale object retrieval. In *CVPR*, 2009. 2, 6, 7, 8

[20] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *CVPR*, 2007. 1, 2, 6, 7, 8

[21] J. Philbin, O. Chum, J. Sivic, M. Isard, and A. Zisserman. Lost in quantization: Improving particular object retrieval in large scale image databases. In *CVPR*, 2008. 2, 5, 6, 8

[22] D. Qin, S. Gammeter, L. Bossard, T. Quack, and L. Van Gool. Hello neighbor: Accurate object retrieval with k-reciprocal nearest neighbors. In *CVPR*, 2011. 2

[23] D. Qin, C. Wengert, and L. Van Gool. Query adaptive similarity for large scale object retrieval. In *CVPR*, 2013. 2

[24] J. Sethian. Fast marching methods. *SIAM Review*, 41(2), 1999. 4

[25] X. Shen, Z. Lin, J. Brandt, S. Avidan, and Y. Wu. Object retrieval and localization with spatially-constrained similarity measure and k-nn re-ranking. In *CVPR*, 2012. 2, 8

[26] G. Tolias, Y. Avrithis, and H. Jégou. To aggregate or not to aggregate: Selective match kernels for image search. In *ICCV*, 2013. 2

[27] P. Turcot and D. Lowe. Better matching with fewer features: the selection of useful features in large database recognition problems. In *ICCV*, 2009. 2