Vision par Ordinateur :
Indexation et recherche d'images

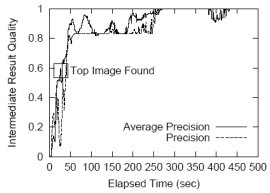
Approximate searches

--Exact Searches--
--Approximate Searches--

- Remember:
  - The cost of retrievals is particularly high when searching for the exact answer
    - (curse of dimensionality: all the data space has to be analyzed)

Where does the time goes?

- Disks: we know this from DB work
  - time spent for I/Os should be predictable
  - bounded and fixed.
- CPU: 99% of time goes to distances calculations
  - try to avoid distance calculations at search time
- Overall: analysis shows top result found very early



--Exact Searches--
--Approximate Searches--

- It is possible to answer faster if returning an approximate answer
  - what are the gains?
  - how good or bad is the approximation?
  - how can this be done?
- The key idea is to ignore some points
- True NNs might be missed…

Approximate Searches

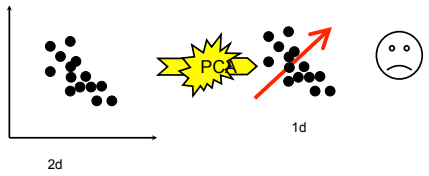
- Several Families of Approaches for Doing Approximate Searches:
  - PCA-based
  - Early aborts
  - Quantification-based
    - Scalar quantification: approximate the indexing of DB
    - Vectorial quantification: approximate the description

Idea: reduce the dim of data

- High dimensionality is problematic
- Let's reduce the dimensionality of data
- Hope to get back on track
- What dimensionality techniques are around?
  - mainly the Principal Component Analysis (PCA)

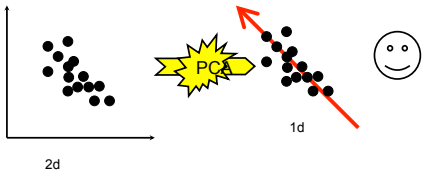
PCA

- Find correlation between dimensions
  - use only one instead of two
- How much information carries each dim?
  - eliminate information-less dimensions
  - New dimensions carefully determined!



PCA

- Find correlation between dimensions
  - use only one instead of two
- How much information carries each dim?
  - eliminate information-less dimensions



PCA

- Find new axes as linear combinations of existing axes
  - translation & rotation of the basis
- 5 canonical steps
  - center the matrix of vectors
  - compute co-variance matrix
  - compute eigenvectors and eigenvalues
  - choose what to keep
    - the d' most "energetic" components (d' largest eigenvalues)

PCA

- Less dimensions, so better behavior!
- Distances coarsely preserved
  - NN found in transformed space ≠ true NN
- Useful only when reduced enough
  - 20d -> 10d; 512d -> 256d
- Difficulties with updates
- Problems with outliers

Early Aborts

- List of candidate cells too long? Cut it!
- Very simple approach
  - find candidate cells
  - read some, then stop
  - stop: # of cells, time spent, distance, …
- Can be very efficient
- Hard to evaluate accuracy
- Unpredictable accuracy

Quantification-based Indexing

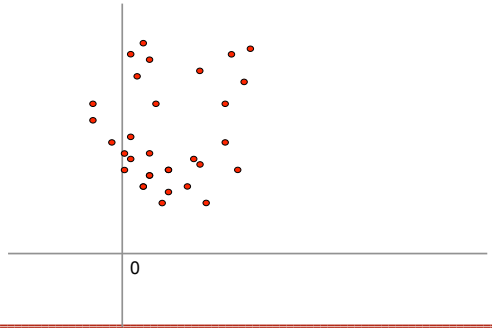
- Several families of quantification-based techniques
- Several families of quantification operations
  - scalar quantifiers
  - vectorial quantifiers
    - structured
    - unstructured
- Quantification Creates Cells
  - Each cell groups points defined as being similar enough
  - Each cell has a representative value
  - This is indexing!

Scalar vs. Vectorial

- Scalar Quantification
  - maybe the simplest form of quantification
  - the dictionary F is defined in $\mathbb{R}^1$
  - points from E exist in $\mathbb{R}^d$
  - it is typically an orthogonal projection on one segmented axis

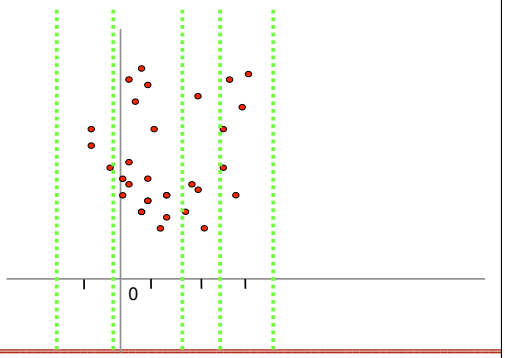
13

Scalar vs. Vectorial



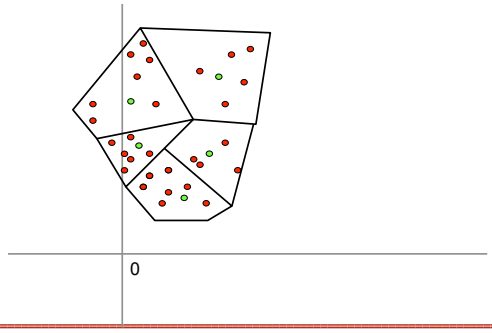
14

Scalar vs. Vectorial



15

Scalar vs. Vectorial



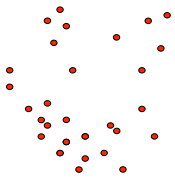
16

Scalar Quantification

- Several approaches try to turn an expensive d-dimensional search into one (or more) cheap 1-dimensional search(es)
  - Random Projections + Hashing
  - Random Projections + Rank Aggregation
  - Random Projections + Segmentations

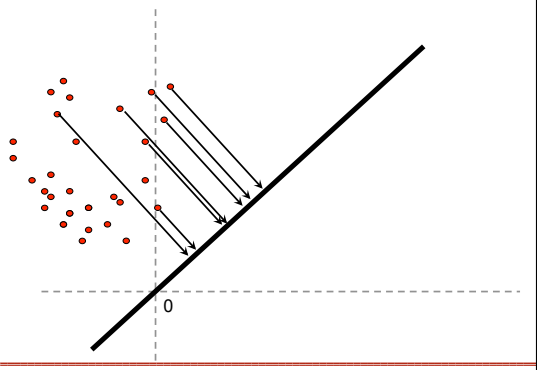
17

What are Random Projections?



18

What are Random Projections?



19

LSH: Locality Sensitive Hashing

- Random projection + hashing
- LSH is a very popular method
- Traditional hashing means
  - x and y are 2 values
  - if h(x)=h(y) ⇒x & y collide in the same bucket
- In high dimensions
  - near vectors should be hashed in the same bucket
  - distant vectors should go to different buckets

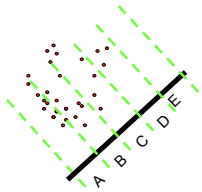
  ➔ "Locality sensitive" property of hash functions

20

LSH

- K random lines are generated
- Each line is partitioned in fixed sized intervals
  - this is called a radius *r*, typically 6—12 intervals
- Each interval is named by a symbol

a line (a,b) + quantization (step r) acts as a hash-function h



$$h(p) = \left\lfloor \frac{\langle p,a \rangle + b}{r} \right\rfloor$$

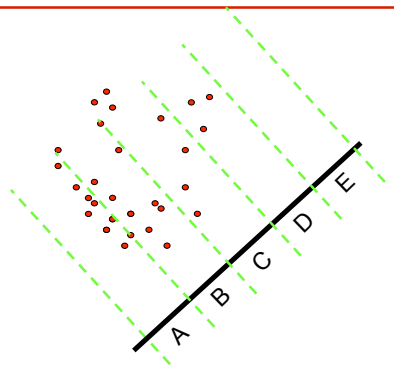
21

LSH

- 1rst level of hashing:
  - All descriptors are projected on one line
    - ~ they all get the symbol from where they lie
    - ~ this is hashing!
  - repeat this for all lines: h_1, h_2, h_3,....h_K
  - concatenate symbols
  - All vectors can thus be named with a word of length K
    - ~ $g^{(1)}(p)=(h_1(p), h_2(p), h_3(p),....h_K(p))$
    - ~ it is a "locality sensitive fingerprint"

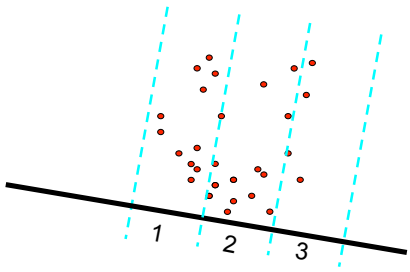
22

LSH



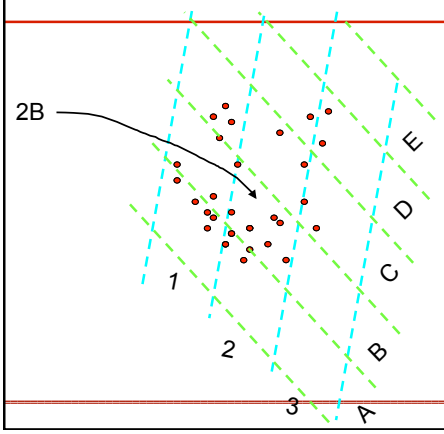
23

LSH



24

LSH



25

LSH

- Partitioning lines
  - smaller the intervals (r), larger the alphabet
  - probability of individual symbols varies a lot
    - ~ points tend to be normally distributed
- All points are turned into words
  - 2 points with the same word may be close
  - 2 points with different words may be far
  - **however: 2 close points might differ on one (few) symbol only**

26

LSH

- 2nd level of hashing: improving neighborhood
  - pick randomly m symbols from each word
  - Concatenate them
  - it makes a new word: hash-value $g^{(2)}$ (of length m)
  - points with identical 2nd-hash value $g^{(2)}$ stored together
  - do this many times (L=30)

$$g^{(2)}_j(p)=(h_{j1}(p), h_{j2}(p),\ldots,h_{jm}(p)), \qquad j \text{ in } \{1,L\}$$

27

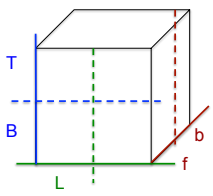
LSH

- Querying:
  - hash the query q according to the K random lines
    - ~ it gives a word: $g^{(1)}(q)$
  - rehash it L times
    - ~ it gives L new shorter words: $g_1^{(2)}(q)$, $g_2^{(2)}(q)$, … $g_L^{(2)}(q)$
  - load those L buckets
    - ~ give a list of candidate points: the data points that are hashed into the same bucket as *q*
  - compute distances with candidates

28

LSH – example

- Think about a cube
- 3 random lines partitioning the cube (K=3)
  - gives 3 hash functions h_1, h_2, h_3,
  - 3 symbols per DB point
- 2 partitions per line (r=2)
  - line 1 = left ; right L/R
  - line 2 = top ; bottom T/B
  - line 3 = front ; back f/b



29

LSH – example

DB vector	direct LSH word	2nd hash. Symb 1&3	2nd hash. Symb 2&3
v1	L.B.f.	L.f.	B.f.
v2	L.T.f.	L.f.	T.f.
v3	R.B.b.	R.b.	B.b.
v4	R.T.f.	R.f.	T.f.
v5	L.B.b.	L.b.	B.b.
v6	R.B.b.	R.b.	B.b.

- because of segmentation, close vectors can have different symbols
  - visible in direct LSH words
- 2nd round of hashing minimizes that effect
  - see here and here (with m=2, L=2)
- more 2nd-round functions shall be defined…

30

LSH – To keep in mind

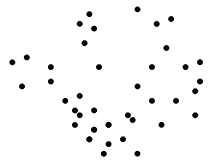
- Hashing is fast and nice
- But
  - normal distribution⇒most vectors fall into the central intervals, on all hash lines
  - depending on 2nd-round words, quality varies
- Distance computation

The NV-Tree

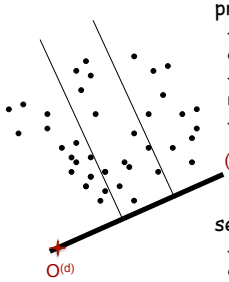
- Random Projections + Segmentation + Ranking
- Design objectives:
  - geared toward local descriptors
  - one I/O per query descriptor
  - no distance calculations during searches
  - copes with SCALE

NV-Tree: Projections & Segmentations

Here is a set of points



NV-Tree: Projections & Segmentations

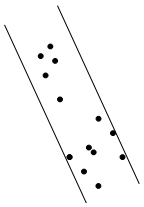


project points on a random line
- -this gives an order -a rank- for points on the line
- -points near in space tend to have close ranks
- -points far in space also :-(

segment line (in 3 here)
- -points far in space go to different chunks

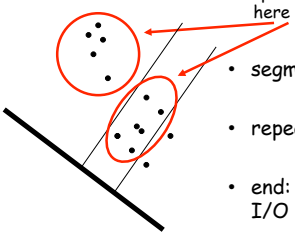
points near borders require specific treatment

NV-Tree: Projections & Segmentations



- pick a segment
- project again its points

NV-Tree: Projections & Segmentations



- project points on a **new** random line
  - -points far in space might get separated here
- segment again
- repeat
- end: #of points in a segment fill an I/O granule

small collection: few proj & seg
large collection: more proj & seg

Keeping Track of Segs and Projs

- Maintain a tree like structure:
  - Each node:
    - Random line equation (d)
    - Origin ($O^{(d)}$)
    - Distances from segment boundaries to origin
  - Each leaf:
    - All descriptors belonging to that leaf
    - Their distances to the origin of the last random line
    - Descriptors are **ranked** by increasing distance to the origin

Keeping Track of Segs and Projs

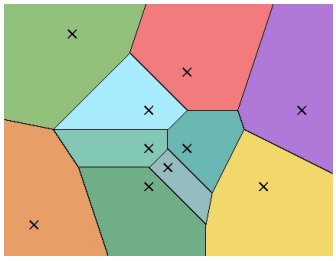
- At Query time:
  - project the query point, check which segment
  - re-project, until reach a leaf
  - fetch, use ranking to return close neighbors
- No distance computation between descriptors
- Key result :
  Query time independent of descriptor collection size

Vector Quantization

- General idea:
  - maps any *vector* from R^d to the closest *code-vector*
    - the code-book is defined elsewhere
    - code vectors $\in R^d$
  - *code-vector identifier* transmitted instead of vector
    - this is compression
  - errors are related to max dist between *vector* and *code-vector*

Compression – VQ

- This basically defines a Voronoï diagram



K-means – the algorithm

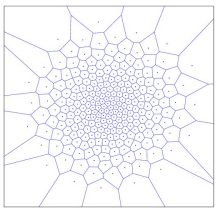
1. pick [randomly] k seed points
2. assign each point to its closest seed
3. gives an initial partitioning
4. compute the centroid of each partition
   - the centroid is the center of each partition, the mean point
5. assign each point to its nearest centroid
6. goto 4; stop if nothing moves

K-means – Comments

- Extremely popular, clear, nice
- Somehow efficient
  - complexity is O(#iteration * k * n)
  - #iterations << n
    - often quickly converges
    - intra class variance always decreases
  - k << n (not that true for CBIR)
- Need to set k (☹)
- Local optima (☹)

K-means – Comments

- Copes with weird data distributions
  - might create unbalanced cells
    - ~ cardinality
    - ~ size



- Defines Voronoi cells (and separating hyperplanes)
  - convex clusters only (☹)

K-means – Comments

- Very many variations
  - termination condition
    - ~ no move; intra-classΔ<ε; X-iterations; …
  - selection of initial seeds
  - when is it best to compute centroids?
    - ~ after each move; once all objects moved
  - escaping local optima
    - ~ do T different k-means on different samples
    - ~ seeds = centroids with min intra-class variance
    - ~ cluster the whole data set with these seeds

K-means – Outliers

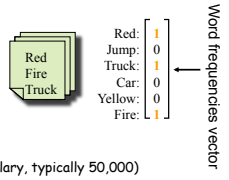
- Outlier= object with extremely
  - large values
  - weird values with respect to other values
  - severely impacts the value of means
  - distort clusters
- Poor behavior with noise and outliers

K-means at work: Video Google

- Key reference:
  - *Josef Sivic and Andrew Zisserman*
    « Video Google: A Text Retrieval Approach to Object Matching in Videos »
    International Conference on Computer Vision, 2003 (ICCV'2003)
- Idea: Google works very well for text Web pages. Use it for images!
  - difficulty: no words in images…

How does a Textual Search Engine Work?

- They (almost) all work the same way (Salton 1971)
- How is represented a document?
  - each document is represented by the list of words it contains (bag of words)
  - Notion of vocabulary



  - Properties of the descriptors
    - ~ Very high-dimensional (size of the vocabulary, typically 50,000)
    - ~ Sparse

How does a Textual Search Engine Work?

- Bof-of-word representation : Example

We have all witnessed the explosion of social networks and used them in one variant or the other. Yet, it seems that they have somehow been victims of their own success to the point that they are very much subject to information overaload. In addition, the use of social networks is not well understood in professional contexts. The opinion of young people is valued very much as they are the future users of any coming technology and at the same time have not been yet too much influenced by years of habits using long-standing technologies.

How does a Textual Search Engine Work?

- Terms in a doc are not equally important:
  - terms that are representative of the content are more important
  - terms that are discriminative wrt the collection are more important
- Practical use:
  - the score $w_{i,j}$ assess the importance of term t_j in doc d_i, usually based on statistical considerations

How does a Textual Search Engine Work?

- Term Frequency (TF)
  - The more frequent a term, the more representative: tf(t_j, d_i)
  - But not sufficient: some words are frequent in too many documents
    - eg *computer* in IT collection of docs
- Inverse Document Frequency (IDF) $idf(t_j) = \log\left(\frac{N}{n_j}\right)$
  - N: number of docs in collection, nj: number of docs containing t_j
  - The more frequent in the collection, the less discriminative and specific

$$w_{i,j} = tf(t_j, d_i).idf(t_j)$$

How does a Textual Search Engine Work?

- Similarity measure: cosine
  - Similarity measure: higher is better ≠ distance!
  - Cosine similarity between query $Q \in \mathbb{R}^n$ and document $D_i \in \mathbb{R}^n$:

$$\delta_{cos}(D_i, Q) = \frac{D_i \cdot Q}{\|D_i\|_{L2}\,\|Q\|_{L2}} = \frac{\sum_{j=1}^{n} w_{i,j}\, w_{q,j}}{\sqrt{\sum_{j=1}^{n} w_{i,j}^2 \sum_{j=1}^{n} w_{q,j}^2}}$$

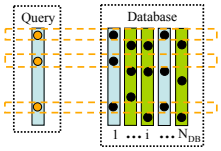
  - Term weights are usually already L_2-normalized:

$$\delta_{cos}(D_i, Q) = \sum_{j=1}^{n} w_{i,j}\, w_{q,j}$$

  - What is the consequence (computational cost) for sparse vectors?

How does a Textual Search Engine Work?

- Which indexing method makes the search fast?
  - inverted lists of words are built:
    - a word point to the list of documents where it appears (+ the number of occurrences)
    - Take advantage of the vector's sparsity and the distance computation (scalar product)
  - for a query, intersect lists



K-means at work: Video Google

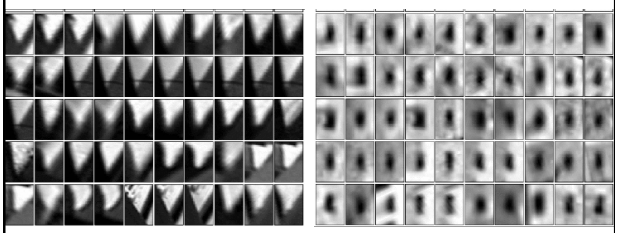
- Idea: Google works very well for text Web pages. Use it for images!
- difficulty: no words in images…

How to get words from images?

Defining Visual Words

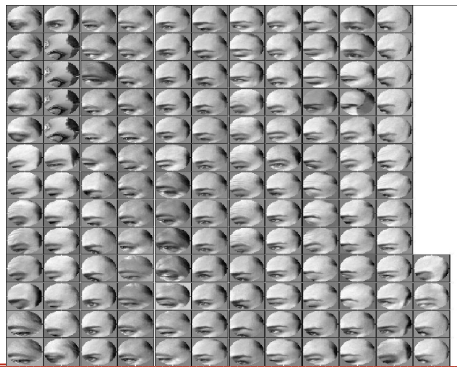
- Let's start with an image collection
  - compute the SIFT on a subset of images
- Cluster the descriptors in 20,000 cells
  - close descriptors are together
  - one possible method: uses k-means
- One cluster= one visual word
- Assign remaining vectors to clusters

Examples of Visual Words



55

Another example



56

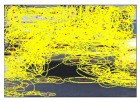
Video Google: Algorithm

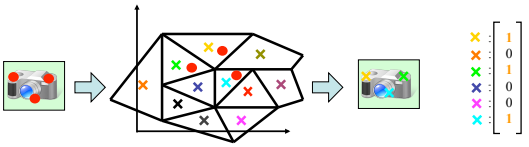
- « Preprocessing » step : learn/create the visual vocabulary (dictionary)
  - Run a k-means algorithm on a subset of descriptors or (better) on an unrelated set of descriptors extracted from some image database
  - Size of the dictionary = size of the further descriptors = k

57

Video Google: Algorithm

- For each image
  - compute interest points and SIFT descriptors
  - assign each descriptor to a visual word (cluster)
  - associate a occurrence vector to the image = number of occurrences of each word in the image





Quantization: each descriptor is represented by the closest visual word

58

Video Google: Algorithm

- Occurrence matrix (inverted file)

$$\begin{array}{c} \\ w_1 \\ w_2 \\ \vdots \\ w_k \end{array} \begin{array}{c} \begin{array}{cccc} im_1 & im_2 & \dots & im_m \end{array} \\ \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ & & & \\ a_{k1} & a_{k2} & \dots & a_{km} \end{pmatrix} \end{array}$$

- From there, a traditional search engine can be used

59

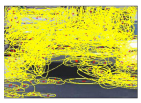
Video Google: Algorithm

- Good things about being back to text:
  - use stop-lists
    - remove words found in every document
  - use tf*idf approaches
    - tf: #occurrence/#words for each document
    - idf: #document/#document_with_that_word
    - higher weight for words appearing rarely
  - variations around list aggregation scheme

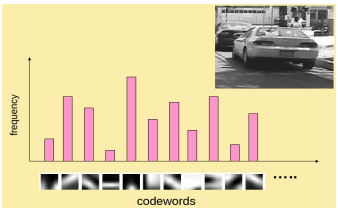
60

Video Google

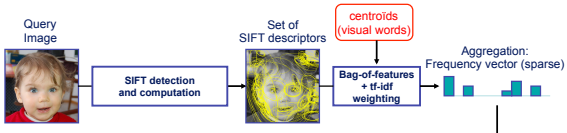
- Image description:
  - BOF: Bag Of Features
  - BOW: Bag of Word
  - BOVW: Bag of Visual Words



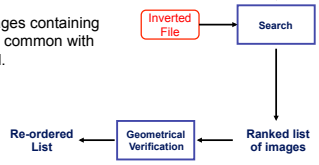
- Aggregation of local features



Video Google



- Inverted file:
  - Only distances with images containing at least one visual word common with the query are computed.



Video Google

- The off-line part is very costly
  - high cost for clustering
    - ~ why 20,000? why k-means?
  - high cost for vector assignment to clusters
    - ~ some strategies exists (pyramidal)
- The online part is cheap
  - but query are not of few words as in Google…
- It works pretty well
  - a lot of new approaches start from here…

Aggregation of local descriptors

- Problem: represent an image by a single fixed-size vector:

  set of *n* local descriptors → 1 vector

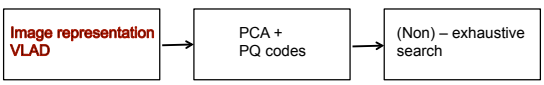
- Most popular idea: BoF representation [Sivic & Zisserman 03]
  - sparse vector
  - highly dimensional

→ high dimensionality reduction/compression introduces loss

- Alternative : vector of locally aggregated descriptors (VLAD)
  - non sparse vector
  - excellent results with a small vector dimensionality

Aggregating local descriptors

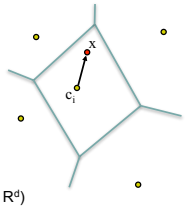
- **Aggregating local descriptors into a compact image representation**
- Aim: improving the tradeoff between
  - search speed
  - memory usage
  - search quality
- Approach: joint optimization of three stages
  - local descriptor aggregation
  - dimension reduction
  - indexing algorithm



VLAD : vector of locally aggregated descriptors

- Learning: a vector quantifier (*k*-means)
  - output: *k* centroids (visual words): $c_1,\ldots,c_i,\ldots c_k$
  - centroid c_i has dimension *d*
- For a given image
  - assign each descriptor x to the closest center c_i
  - accumulate (sum) descriptors per cell

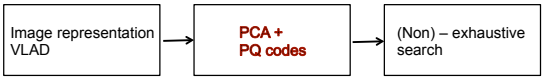
    $v_i := v_i + (x - c_i)$



- VLAD: concatenation of the k sums of residuals v_i (in R^d)
  - dimension $D = k \times d$
- The vector is L2-normalized

Compact image representation

- Approach: joint optimization of three stages
  - local descriptor aggregation
  - dimension reduction
  - indexing algorithm

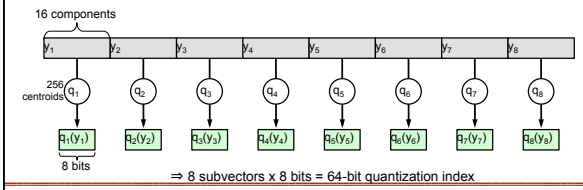


- Dimensionality reduction with
  - Principal component analysis (PCA)
  - Compact encoding: product quantizer

→ very compact descriptor, fast nearest neighbor search, little storage requirements

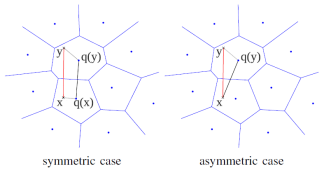
Product quantization

- Vector split into *m* subvectors: $y \rightarrow [y_1|\dots|y_m]$
- Subvectors are quantized separately by quantizers $q(y) = [q_1(y_1)|\dots|q_m(y_m)]$ where each q_i is learned by *k*-means with a limited number of centroids
- Example: y = 128-dim vector split in 8 subvectors of dimension 16
  - each subvector is quantized with 256 centroids -> 8 bit
  - very large codebook $256^8 \sim 1.8 \times 10^{19}$



⇒ 8 subvectors x 8 bits = 64-bit quantization index

Product quantizer: distance computation



- Symetric case:
  - Distances between centers are stored in tables
  - Query x = [x_1 | x_2 |…|x_m] -> q(x)=[$q_1(x_1)$ |$q_2(x_2)$ |…| $q_m(x_m)$]

$$d(x,y) = \sum_{i=1}^{m} d(q_i(x_i), q_i(y_i))$$

VLAD + PQ codes

- Excellent search accuracy and speed in 10 million of images
- Each image is represented by very few bytes (20 – 40 bytes)
- Tested on up to 220 million video frame
  - extrapolation for 1 billion images: 20GB RAM, query < 1s on 8 cores
- On-line available:
  - Matlab source code of ADC
- Alternative: using Fisher vectors instead of VLAD descriptors [Perronnin'10]