# Similar images with Product Quantization

Mathias Chouilly, Lefty

February 24, 2016

# Goal

- Given an image, find similar image in our dataset

# Goal

- Given an image, find similar image in our dataset
- Mathy version: given a vector, find its nearest neighbors

# Constraints

- Handle very big dataset (possibly 1B elements)
- Fast (Ideally $< 1$s)
- Precise (Image should be similar to query)

# How to start ?

- Image to Vector: we already use GoogLeNet features (1024 floats) for image classification

## How to start ?

- Image to Vector: we already use GoogLeNet features (1024 floats) for image classification
- GoogLeNet is a Deep Neural Network that achieved state-of-the-art on ImageNet 2014 competition

# How to start ?

- Image to Vector: we already use GoogLeNet features (1024 floats) for image classification
- GoogLeNet is a Deep Neural Network that achieved state-of-the-art on ImageNet 2014 competition
- It is very fast to compute compared to similar performing models

## How to start ?

- Image to Vector: we already use GoogLeNet features (1024 floats) for image classification
- GoogLeNet is a Deep Neural Network that achieved state-of-the-art on ImageNet 2014 competition
- It is very fast to compute compared to similar performing models
- Let's use that

# Original Version (Jégou 2011)

- Decompose the space into a Cartesian product of low dimensional subspaces
- Quantize each subspace separately
- A vector is represented by a short code composed of its subspace quantization indices.

# Original Version (Jégou 2011)

- Decompose the space into a Cartesian product of low dimensional subspaces
- Quantize each subspace separately
- A vector is represented by a short code composed of its subspace quantization indices.
- The euclidean distance between two vectors can be efficiently estimated from their codes.
- An asymmetric distance calculation scheme increases the precision by computing the approximate distance between a vector and a code.

## Practical use

- Use a coarse quantizer (k-means) $q_c$ to build an inverted index in original space

# Practical use

- Use a coarse quantizer (k-means) $q_c$ to build an inverted index in original space
- Each vector $y$ is associated to its closest centroid $q_c(y)$, and we denote the residual vector: $r(y) = y - q_c(y)$

# Practical use

- Use a coarse quantizer (k-means) $q_c$ to build an inverted index in original space
- Each vector $y$ is associated to its closest centroid $q_c(y)$, and we denote the residual vector: $r(y) = y - q_c(y)$
- Residual vectors are product quantized by $q_p$, such that each vector is approximated by $y \approx q_c(y) + q_p(r(y))$

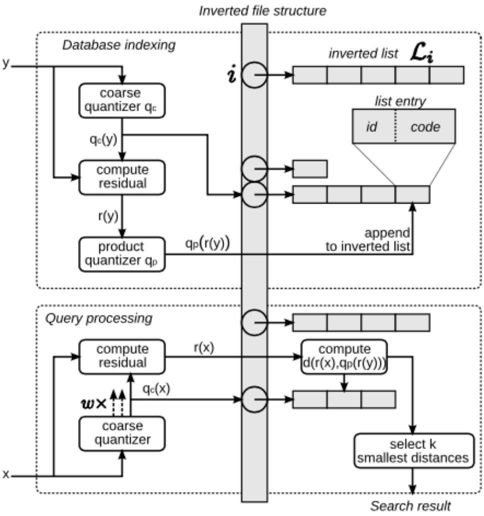
# Practical use

- Use a coarse quantizer (k-means) $q_c$ to build an inverted index in original space
- Each vector $y$ is associated to its closest centroid $q_c(y)$, and we denote the residual vector: $r(y) = y - q_c(y)$
- Residual vectors are product quantized by $q_p$, such that each vector is approximated by $y \approx q_c(y) + q_p(r(y))$
- Given a query vector $x$, find its nearest coarse centroids, compute residuals to each of these coarse centroids

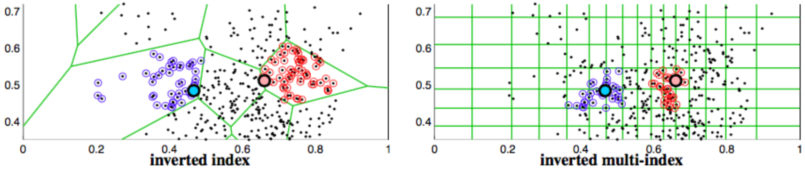
# Practical use

- Use a coarse quantizer (k-means) $q_c$ to build an inverted index in original space
- Each vector $y$ is associated to its closest centroid $q_c(y)$, and we denote the residual vector: $r(y) = y - q_c(y)$
- Residual vectors are product quantized by $q_p$, such that each vector is approximated by $y \approx q_c(y) + q_p(r(y))$
- Given a query vector $x$, find its nearest coarse centroids, compute residuals to each of these coarse centroids
- Use the quantization codes to compute distances and find the smallests

# Practical use



# Multi-Index (Yandex 2012)




- Build on Jégou work
- Replace the coarse quantizer by a product quantizer
- Use multi-sequence algorithm to sort coarse centroids
- Space can be divided into $K^2$ cells, just using 2K centroids
- The finer division of the space implies a higher retrieval accuracy, for the same list length as in the simple index case

# Optimized Product Quantization (Microsoft 2013)

- Replace Product Quantization by Optimized Product Quantization
- Find an optimal space decomposition when choosing low dimensional subspaces

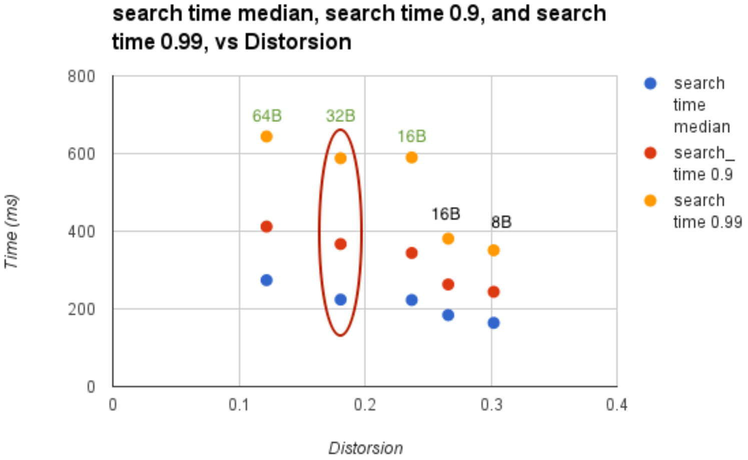
# Optimized Product Quantization (Microsoft 2013)

- Replace Product Quantization by Optimized Product Quantization
- Find an optimal space decomposition when choosing low dimensional subspaces
- Compute covariance matrix of the data
- Use eigen value allocation algorithm to build a rotation matrix
- Do Product Quantization on the rotated data

# Fast Bilayer Product Quantization (Yandex 2014)

- Use the assumption that all residuals are compressed in the same space
- Precompute the norms at launch
- Precompute scalar products at each query
- Use precomputed tables to compute distances very fast

# Lefty Product Quantization



# Lefty Product Quantization

- Use Optimized Product Quantization as an indexing-level quantizer with 16k centroids in each subspace of 512 dimensions. That is 256M possible centroids.
- Use Optimized Product Quantization as a compressing-level quantizer with 256 centroids in each subspace of 32 dimensions. That is $10^{77}$ possible centroids.
- Thus each vector is associated with its indexing ids (2 int) and its compressing ids (32 bytes)

# Lefty Product Quantization

- Use Optimized Product Quantization as an indexing-level quantizer with 16k centroids in each subspace of 512 dimensions. That is 256M possible centroids.
- Use Optimized Product Quantization as a compressing-level quantizer with 256 centroids in each subspace of 32 dimensions. That is $10^{77}$ possible centroids.
- Thus each vector is associated with its indexing ids (2 int) and its compressing ids (32 bytes)
- Precompute the norms and scalar products to speed up distances calculations (from 1024 multiplications and 1023 additions to 96 additions, for each candidate)

# Indexing process

- Rotate features, and quantize them with indexing-level quantizer -> 2 indexing ids

# Indexing process

- Rotate features, and quantize them with indexing-level quantizer -> 2 indexing ids
- Compute residuals, rotate them, and quantize them with compressing-level quantizer -> 32 compressing ids

# Indexing process

- Rotate features, and quantize them with indexing-level quantizer -> 2 indexing ids
- Compute residuals, rotate them, and quantize them with compressing-level quantizer -> 32 compressing ids
- Add the vector to inverted multi index database, with indexing ids as key, and list of compressed residuals as value.

# Indexing process

- Rotate features, and quantize them with indexing-level quantizer -> 2 indexing ids
- Compute residuals, rotate them, and quantize them with compressing-level quantizer -> 32 compressing ids
- Add the vector to inverted multi index database, with indexing ids as key, and list of compressed residuals as value.
- This takes around 10ms per vector

# Querying process

- Rotate features, find nearest centroids with multi-sequence algorithm

## Querying process

- Rotate features, find nearest centroids with multi-sequence algorithm
- Get compressed residuals from database, to build a list of candidates

# Querying process

- Rotate features, find nearest centroids with multi-sequence algorithm
- Get compressed residuals from database, to build a list of candidates
- Rotate features again, build a look-up table for fast distance computation

# Querying process

- Rotate features, find nearest centroids with multi-sequence algorithm
- Get compressed residuals from database, to build a list of candidates
- Rotate features again, build a look-up table for fast distance computation
- Rerank the list of candidates by increasing distance to query vector

# Querying process

- Rotate features, find nearest centroids with multi-sequence algorithm
- Get compressed residuals from database, to build a list of candidates
- Rotate features again, build a look-up table for fast distance computation
- Rerank the list of candidates by increasing distance to query vector
- This takes around 500ms-1s per query*

# Libraries used

- RocksDB (Facebook): embeddable persistent key-value store for fast storage. It stores the inverted multi index data.
- FlatBuffers (Google): serialization lib that provides access to data without parsing. Used as values in RocksDB.
- Eigen for matrix computation
- gRPC & Protocol Buffers as always

# Machines and Servers

- 1 similia-indexer machine (2 vCPUs, 1.8 GB RAM) with 3 similia_indexer_processor
- 2 similia-server machines (4 vCPUs, 15 GB RAM, local SSD)
- Each similia-server machine has an inverted_multi_index_server and a similia_server
- Librarian GetSimilarImages RPC wraps the call to similia_server with features retrieval/computation/storage

# Conclusion

- One can find similar images on instagram by providing to our API an url of a .jpg
- Currently 120M+ images indexed
- Fast (<1s) *
- Precise *
- *Mileage may vary

Thanks (everyone, with special thanks to Christian, Jean, and Mathieu)