# THE TMK+PDQF VIDEO-HASHING ALGORITHM AND THE PDQ IMAGE-HASHING ALGORITHM

We present a pair of technologies for the copy-detection problem space.

- PDQ, a photo-hashing algorithm for general-purpose use. Hashes are 256 bits with hamming distance, accompanied by a 0-100 quality metric which quantifies level of detail, e.g. for identifying blurry/featureless images. Hash-computation time is on the same order of magnitude as disk-read of image files.
- TMK+PDQF, a video-hashing algorithm for general-purpose use. It uses PDQ minus the final binary-quantization step (hence PDQF, for *floating-point)*, and then uses the TMK algorithm per se to collect timewise information about the frames. Hashes are 256KB but the first 1KB serve to differentiate almost all videos. Hash-computation time is a high multiple of video-playback time: perhaps 30x depending on storage density.
- The designs have been chosen to leverage large-scale lessons learned from abuse-detection patterns encountered in practice at Facebook.

## TABLE OF CONTENTS

## OVERVIEW

### DESIGN GOALS

TMK+PDQF and PDQ are **syntactic** rather than **semantic** hashers. Algorithms in the latter category detect features within images, e.g. determining that a given image is a picture of a tree. Such algorithms are powerful: they can detect different photos of the same individual, for example, by identifying facial features. The prices paid are model-training time and a-priori selection of the feature set to be recognized. For copy-detection use cases, by contrast, we simply want to see if two images are essentially the same, having available neither prior information about the images, nor their context.

Semantic hashers include various machine-learning techniques, such as those in use at Facebook; syntactic hashers alongside PDQ include pHash[1], aHash, dHash, and GIST, all of which are discussed in this document. They simply compare similarity of images without extracting meaning from them.

| Deeper image analysis, more robust, higher recall, higher CPU cost → | | | |
|---|---|---|---|
| ← Shallower image analysis, stricter matching, lower cost, faster processing | | | |
| Exact same bits: MD5, SHA-256 | Syntactic: PhotoDNA, dHash/aHash/pHash, PDQ | Deeper syntactic: GIST, SIFT | Semantic: Machine-learning algorithms |

Like other algorithms in its class, PDQ is **relatively fast**, running at approximately the same speed as disk reads. Likewise, TMK+PDQF runs at a very high multiple of video playback speeds -- perhaps 30x, depending on storage density.

Syntactic hashers excel in finding media which are shared with **minimum adversariality** – image quality is reduced, JPEG is converted to PNG, etc. They are unsuitable for detecting intentional attack/obfuscation, including deeper crops. Detection of more adversarial image transformations lies within the domain of semantic/machine-learning methods.

The context is that more powerful algorithms detect more highly manipulated variants of media; then lower-level syntactic hashers can be used to communicate information about detected variants.

In summary, TMK+PDQF and PDQ are suitable items in a service provider's content-matching toolbox. They are proposed as high-throughput, on-line hashing techniques. Facebook, as well as other tech companies, already uses multiple proprietary technologies appropriate to various use-cases. Internally none of the major tech companies needs another matching algorithm. What *is* lacking, though, is an unencumbered **algorithm for sharing hashes across companies**. This is the niche that TMK+PDQF and PDQ are designed to fill.

[1] PDQ owes much to pHash: in particular the former is an elaboration, extension, and improvement to the basic concepts of pHash – with a completely independent software implementation. See also (Zauner, 2010).

## FEATURES OF TMK

TMK (for *Temporal Match Kernel*) is a video-similarity-detection algorithm produced in conjunction with Facebook AI Research (FAIR). It produces fixed-length video hashes (on the order of 256KB), so that results are bounded in size.

### MATCHING PERFORMANCE

TMK detects whether two videos are visually identical, even if the frame rates, file format, and/or pixel resolution are changed. Moreover, it detects videos which differ in minor edits: light watermarking, insertion/deletion of small header/trailer sequences, etc.

It does not detect adversarially modified videos, e.g. cropping, border addition, heavier logos/overlays, etc. It belongs squarely in the copy-detection space.

As presently implemented, TMK+PDQF operates on whole videos, detecting small timewise edits: it might detect a two minutes and five seconds long variant of a two-minute long video where the additional 5 seconds are header or trailer material, but it would not detect those two minutes of footage embedded within, say, an hour-long video. It would be possible for a future version of TMK to detect more aggressive edits, including scene/subsequence detection. And as noted above, the purpose of TMK+PDQF as currently construed is to communicate the results of more powerful semantic-matching algorithms – TMK+PDQF is one component (a hash-sharing component) in an organization’s content-matching toolkit. Concretely, it may take more than one TMK hash to communicate the results of a complex video-matcher’s discovery of all the variants of an original video – we view this as an intended use of TMK+PDQF.

### HASH LENGTH

As detailed below, the video descriptor size for TMK+PDQF does not depend on the video length. Hash files are 256KB in size, although a 1KB vector suffices to differentiate almost all videos. (If a different framewise algorithm than PDQF were used, the hash size would be different – PDQF framewise features are 256 floating-point numbers so TMK+SomethingElse would have hash-file size of 512KB if SomethingElse produced 512 floating-point numbers per frame.)

### LICENSING

This is released under an open-source license at https://github.com/facebook/ThreatExchange/tree/master/hashing.

## FEATURES OF PDQ

### MATCHING PERFORMANCE

As will be seen in detail below, PDQ is effective at matching similar images (recall) as well as separating dissimilar images (precision). It is designed for lightly (non-adversarially) modified images. It does well with JPEG<->PNG conversions, JPEG-quality reduction, etc. And like many algorithms, it is possible for active adversaries (e.g. spammers) to modify images to evade detection. Lastly, PDQ has been found to tolerate light watermarking / logo placement.

PDQ, like many global-descriptor algorithms, does not do well at pairing photos one of which is a crop of another.

While producing hashes of images, PDQ can also compute what the hash of a rotated/flipped photo would have been, at very little additional computational expense.

### QUALITY METRIC

In addition to producing a hash of a photo, PDQ has a 0-100 quality metric which flags images that are relatively featureless.

### LICENSING

This is released under an open-source license at https://github.com/facebook/ThreatExchange/tree/master/hashing.

## ALGORITHMS

## TMK ALGORITHM

This section is shorter than the corresponding section for PDQ as significant algorithm detail for TMK is within the paper by Poullot et al., and significant implementation details are within the GitHub repo.

### TMK ALGORITHM SKETCH

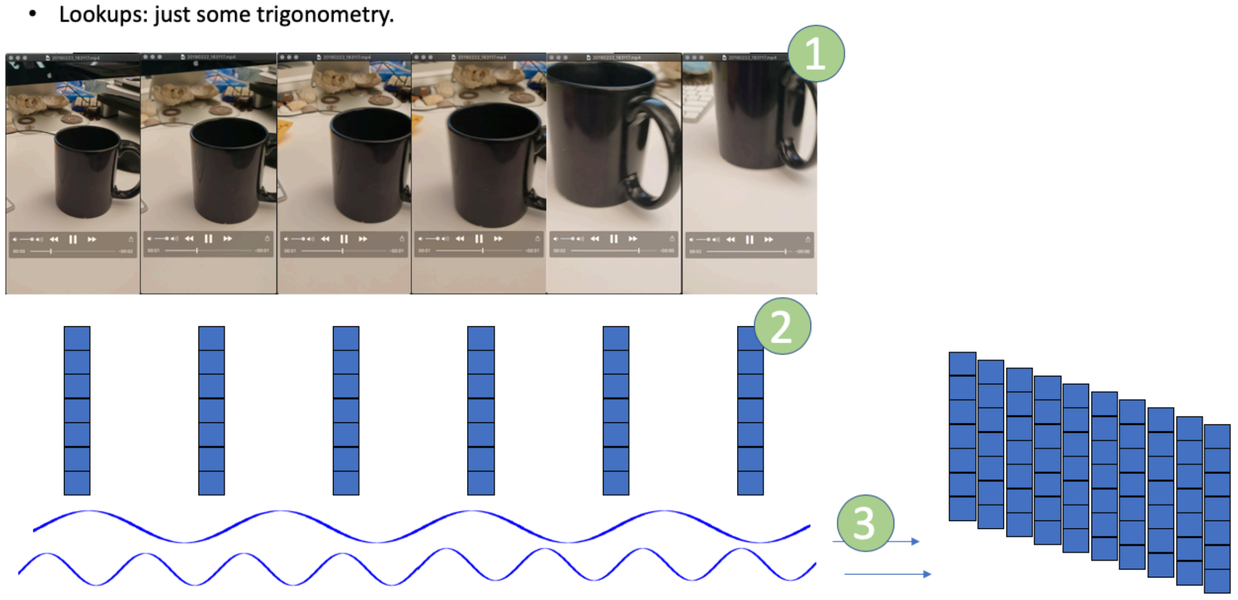
#### COMPUTING HASHES

Given a video:

- Time-resample to some common frame-rate: we use **15 frames per second**.
- Compute some "frame descriptor" or "**frame feature**" or "embedding": a mapping from image to a floating-point vector. For purposes of this document we use **PDQF**: simply PDQ as in the PDQ algorithm section of this document, minus the final binary-quantization step. (The *F* is for *floating-point*.)
    - Compute the elementwise average, cos/sin weighted, with various periods.
    - PDQF produces a 256-element vector for each frame so each temporalized feature is also a 256-element vector. For example, slot 37 is the time-weighted average of slot 37 over all frames of the video.
- Output:
    - The unweighted average over all frame features is called the **level-1 feature**.
    - The collection of weighted averages, using cosine and sine, various different periods, and various different fourier weights are collectively called the **level-2 feature**.
    - After all frames are processed, every feature is L2-normalized; then the level-2 features are weighted by the chosen fourier weights.
- File format:
    - IEEE floating-point numbers are 4 bytes each. PDQF uses 256-element vectors. We use cosine and sine weighting, four different periods, and 32 different fourier weights. In total this means the level-1 feature is 1KB per video, and the level-2 feature is 4x2x4x32x256 = 256KB per video.
    - Metadata including file-type-identifying magic numbers, fourier periods and fourier weights, along with the level-1 and level-2 features, are all collected into a .tmk file format. Explicit details are shown in the README-more.md file within the GitHub repo.

**TMK = Temporal Match Kernel**

- 1. Time-resample to some common frame-rate.
- 2. Compute some frame-descriptor: some chosen embedding from image to 256-element vector
- 3. Elementwise average, cos/sin weighed, with various periods. The hash is just all the trig-weighted averages.
- Lookups: just some trigonometry.



## COMPARING HASHES

- Given a pair of .tmk files, the cosine similarity is computed between the level-1 features. This is called the **level-1 pair score**. If these are less than the desired threshold of 0.7 – perfect match being 1.0 and smallest possible being -1.0 – then the videos are deemed to not match. It's key to note that this cosine similarity is quick and is suited to indexing systems such as FAISS (https://github.com/facebookresearch/faiss).
- Only if the level-1 pair score is within threshold do we compute a more detailed function of the level-2 features of the two .tmk files, as detailed in the paper by Poullot et al. and in the source code. This is called the **level-2 pair score**. Perfect match is 1.0 and minimum value is 0.0. Our recommended threshold is 0.7. The level-2 pair score isn't indexable by FAISS but it needs to be computed only on a level-1 match – which is FAISS-indexable.

## TMK ALGORITHM DETAILS

Please see the paper by Poullot et al. in the Bibliography (https://hal.inria.fr/hal-01842277/document).

## PDQ ALGORITHM

PDQ is:

- **P**: A **perceptual** hasher: it tries to match photos which people perceive as similar;
- **D**: It is a spectral hashing algorithm, namely, it uses a Discrete Cosine Transform (**DCT**);
- **Q**: One of its outputs is a **quality** metric, enabling systems to filter junky/featureless frames out of processing.

As well, it is pretty quick – as detailed below, while there are opportunities for more low-level optimization, the reference design can hash an image in the same time as, or less than, it takes to read the image file off a disk.

### PDQ ALGORITHM SKETCH

PDQ is inspired by, and related to, pHash (Zauner, 2010), as detailed below, while addressing some of concerns with pHash. PDQ's algorithm steps are:

- Convert from **RGB** to **luminance** rather than greyscale.
- Using **two-pass Jarosz filters** (i.e. tent convolutions), compute a weighted average of 64x64 subblocks of the luminance image. (*This is prohibitively time-consuming for megapixel input so we recommend using an off-the-shelf technique to first resize to 512x512 before converting from RGB to luminance.*) The Jarosz-filter pass has ensured that these pixels are not just cherry-picks out of the higher-resolution image, but rather are representative of the full subblocks.
- Within the 64x64 downsample, compute a sum of absolute values of horizontal and vertical gradients (i.e. the L1 norm of quantized gradients of the downsample matrix), then rescale this number such that images with (empirically determined) adequate amounts of features have a score at/near 100, while completely featureless images (such as solid-color images) have a score of 0. This 0-100 **quality metric** is useful to identify relatively featureless frames.
- Also within the 64x64 downsample, compute a two-dimensional discrete cosine transform (**DCT**), retaining the 1-16 slots in and X and Y directions.
- From the 16x16 DCT output, compute the **median value** in transform space.
- For each of the **16x16 bits** of the output hash, emit a **1** if the corresponding element of transform space is greater than the median, otherwise emit a **0**.
- The result is 256 bits rather than 64, which we feel is appropriate for web-scale.
- We efficiently compute all eight dihedral-transform hashes – the hashes one would have gotten for rotations and flips of the image – at fractional additional compute cost. These take as input the 16x16 floating-point DCT-output matrix, so they are quick to compute and independent of the size of the original image.

Out of all these steps, the key concept is the spectral-hashing property provided by the DCT: PDQ identifies which low-to-medium spatial frequency components contribute more or less to the image.

### PDQ HASH PROPERTIES AND DISTANCE THRESHOLDS

Given the facts of the algorithm sketch above, we have the following properties for PDQ hashes:

- **Two hashes are identical if their hamming distance is 0**; their maximum possible hamming distance is 256.
- Since each bit of the 16x16 output is a 1 if the corresponding DCT bin exceeds the median DCT value or not, necessarily each hash has 128 1-bits and 128 0-bits. The only exception is in highly degenerate images, e.g. solid-color images, where multiple DCT output bins have the exact same numerical value. (Phrased differently**, all non-trivial PDQ hashes have a hamming norm of 128**.) In particular, a perfectly solid-color image has all 256 bits 0, but essentially solid-color images won't: hence the reason for having a quality metric to surface these.
- Since hashes have half their bits 0s and half 1s, the maximum number of unique PDQ hashes is not $2^{256}$ but rather 256 choose 128 (still an astronomical number).
- As well, hamming distance between pairs of PDQ hashes is (again, almost) always an even number.
- Statistical distribution of distances will be discussed below. For the moment, let us say that the hamming distance between random pairs of PDQ hashes averages 128, supported on the range of 90 to 160 or so. Confident-match distances are up to the system designer, of course, but 30, 20, or less has been found to produce good results on evaluation data. (We will gain more context for matching-distance thresholds in the Evaluation section below.)

## PDQ ALGORITHM DETAILS

Each step of the above algorithm sketch is now discussed in detail.

### LUMINANCE CALCULATION

Luminance is computed as in (https://en.wikipedia.org/wiki/Luminance): namely if *R*, *G*, and *B* are 0-255 unsigned-byte values then

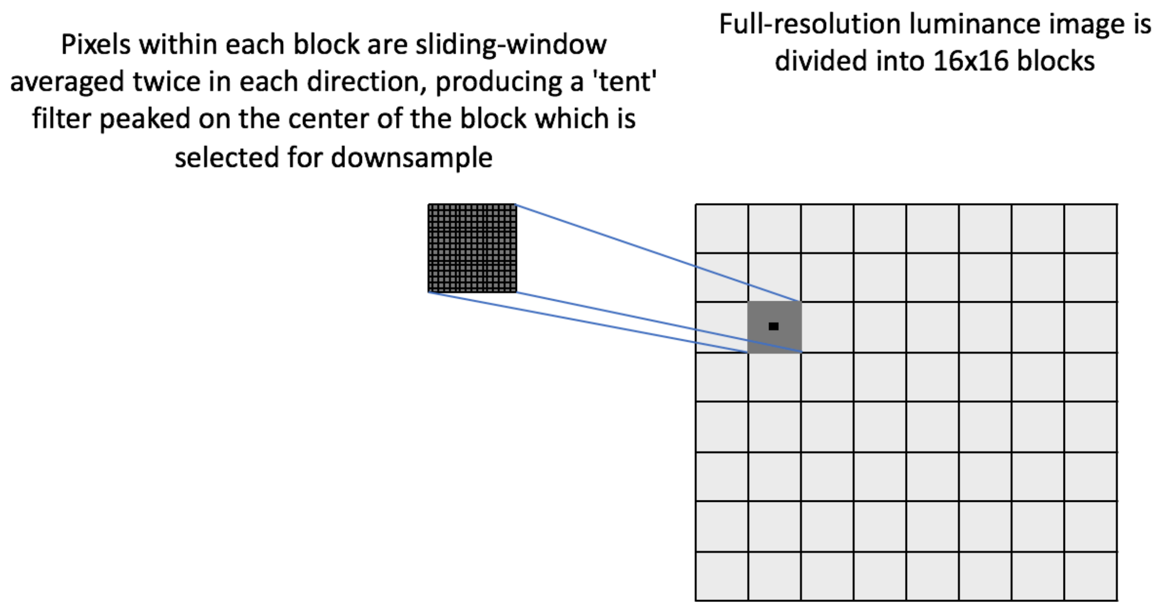
$$Y = 0.299\ R + 0.587\ G + 0.114\ B$$

which is also in the range 0-255. (If floating-point arithmetic is done, 0-255 is fine; for fixed-point arithmetic, say with 12 mantissa bits, then *Y* can be stored as integer with the 0.299 et al. coefficients replaced by $0.299 \cdot 2^{12}$, etc.) Note that this is distinct from greyscale (an unweighted average of *R*, *G*, and *B*); luminance is preferred since it is designed to align with visual perception.

### JAROSZ FILTERS AND DOWNSAMPLING

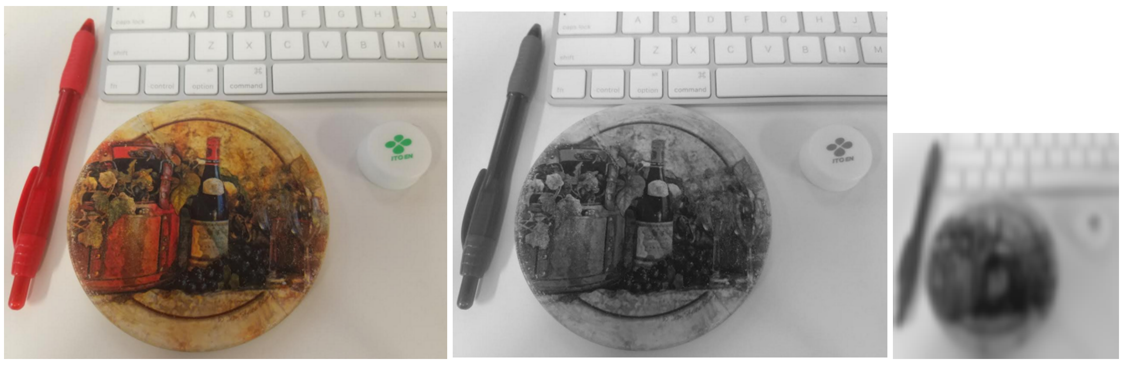
PDQ is intended to be robust with respect to light crops or shifts. For this reason, it is important that the downsample to 64x64 not be a cherry-pick of pixels out of the full-resolution luminance image, but rather an average over pixels within each image block. As is well-known in the image-processing field, a single 'box' filter (i.e. sliding-window sum) in the X and Y directions done twice produces a tent filter, and done three times produces a bell filter; eventually this approaches a Gaussian filter. So an implementation of smoothing/filtering decomposes into box passes, and a count of how many box passes are to be done. Furthermore, as detailed in Jarosz (Jarosz, 2001) the naïve two-dimensional filter can be much more efficiently be done using a pair of one-dimensional passes. Thus four 1-D passes – X direction, Y direction, X direction, Y direction – produces a 2-D tent filter.

**Window sizes:** The pHash algorithm, which inspired PDQ, uses 16x16 boxes and 64 output bits. Within those 256 boxes, single pixels are selected using convolution with a mean filter on 7x7-pixel patches. For example, if the input image is 1024x1024, then each block is 64x64 pixels, and only 49 of those 4096 are selected and averaged to compute the downsample point for the block. This is not robust with respect to light trims (for example, you screenshot and image and crop it out, then look it up in a photo-hashes database, but you were off by a few pixels in your crop). For this reason, PDQ has all pixels within each block of the full-size luminance image contribute to the corresponding downsample point. The window size for a single pass is image height/width, respectively, divided by 32 with round-up. This is to say, half the block size. In our 1024x1024 example, each of the 16x16 blocks has 64x64 pixels. The first pair of Jarosz passes averages over 32x32 patches; the second pair spreads those out to 64x64. Then the downsample pixel for the 64x64 block is selected from the center of that block.


Pixels within each block are sliding-window averaged twice in each direction, producing a 'tent' filter peaked on the center of the block which is selected for downsample
Full-resolution luminance image is divided into 16x16 blocks

**Fixed-point optimization** (not helpful): During the development of the reference implementation, the Jarosz-filter step was co-implemented in fixed-point: namely, take luminance byte values with 12 fractional bits as noted above, i.e. integers with a scale factor of 4096, then do the passes in integer arithmetic. The performance was on par with the floating-point version, within statistical-sampling error bars from run to run, and so (for simplicity) only the floating-point version was retained.

**Sample images:** At left is full-resolution color input; in the center is full-resolution luminance; at right is 64x64 downsample (shown magnified for convenience).



Note in particular that, like any image-hashing algorithm which starts with a downsample, small text which is legible in the full-resolution image is illegible within the PDQ downsample. This means we must be careful about banking content based on tiny amounts of text (even if that text says something particularly unpleasant).

## MORE ON DOWNSAMPLING

The two-pass tent filter described above is CPU-intensive for larger images. We have the following tradeoffs:

- If the two-pass tent filter is done from full-resolution image all the way down to 64x64 PDQ input, the output is identical across various implementations (C++/Python/etc). Yet it is far too slow.
- If an implementation-specific downsampler is used instead, the hamming distance between hashes produced by different implementations, given the same image, is too large.
- For this reason we have chosen a tradeoff: (a) Use implementation-specific downsampler to get to 512x512; (b) use the two-pass tent filter to go from 512x512 to 64x64. This is a balance between efficient computation, and a few bits (0/2/4/6) of hamming distance between different implementations given the same image.

## QUALITY METRIC

Given the 64x64 output, the 64x63 horizontal gradients (nearest-neighbor absolute-value differences) and 63x64 vertical gradients are integer-quantized to mask contributions of just a few pixels and retain the larger ones. These are summed, then divided by a heuristic constant and capped at 100 to obtain a number between 0 and 100.

The goal of the quality metric is to provide an indication of when an image is relatively featureless. The intuition of the quality metric is that if differences between adjacent pixels in the downsample matrix are small, the image is likely to have few features.

Sample images are shown below, in the section on the combined quality metric.

## DISCRETE COSINE TRANSFORM

**One-dimensional definition:** The discrete cosine transform (Wikipedia, 2017) of a length *n* real vector *x* is another length-*n* vector *y* given by

$$y_i = \sum_{j=0}^{n-1} D_{ij} x_j$$

where

$$D_{ij} = \sqrt{\frac{2}{n}} \cos\left(\frac{2\pi}{4n} i(2j+1)\right).$$

**Two-dimensional definition:** The two-dimensional downsampled luminance image may be written as a matrix $A_{ij}$ with *i*, *j* from 0 to 63 inclusive (*n*=64). Then transforms of the columns are simply *D* times each column, i.e. the matrix product *D A*. Left-multiplying this by the matrix transpose $D^t$ is the DCT of the columns of *D A*. Applying the row and column transforms (which associate since matrix multiplication is associative) is the 2-D DCT.

The intuition is that the 0,0 element of the 64x64 output $B = D\,A\,D^t$ indicates how much of the image luminance is contributed by the solid/flat frequency component; 0,*j* for higher *j* indicates how much of the luminance is contributed by higher and higher horizontal frequencies (think fenceposts); 0,*i* for higher frequencies (think ladder steps); and higher *i,j* for diagonal frequency components. PDQ hash, taking its principal inspiration from pHash, retains only the 16x16 submatrix consisting of the 1..16 slots of the DCT output along rows and columns. The intuition is that the very lowest frequency components represent uninteresting coarse-scale structure while the higher components represent fine-scale/noisy/jittery structure which is also uninteresting. The low-to-mid-range frequency contributions are found to better indicate features which the eye perceives (hence the term *perceptual hash*).

**Naïve implementation:** The next question is how to efficiently compute this. In general for the product $Z = X\,Y$ of $n$ x $n$ square matrices we have

$$Z_{ij} = \sum_{k=0}^{n-1} X_{ik} Y_{kj}$$

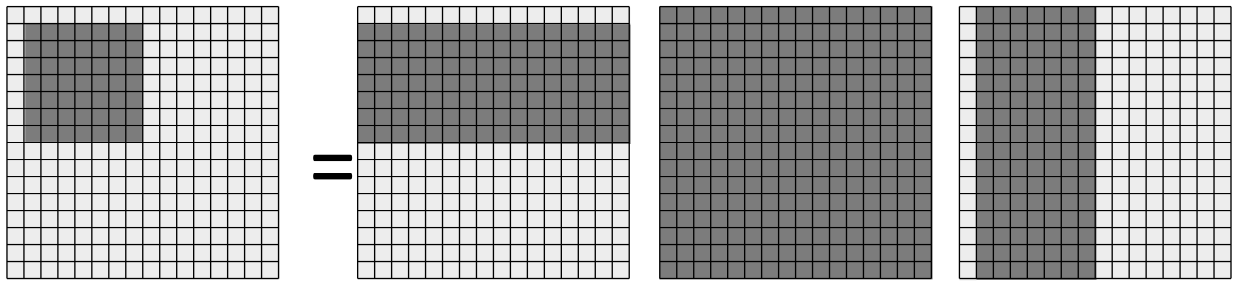
and for the product $Z = X\,Y^t$ (where the second multiplicand is transposed) we have

$$Z_{ij} = \sum_{k=0}^{n-1} X_{ik} Y_{jk}$$

After a little algebra, for the product $B = D\,A\,D^t$ we get

$$B_{ij} = \sum_{k=0}^{n-1} D_{ik} \sum_{l=0}^{n-1} A_{kl}\, D_{jl}$$

**Trimmed implementation:** In the above formula, full DCT output is obtained for *i*, *j* from 0..63. But since PDQ uses frequency slots 1-16 from the DCT on the 64x64 input, the *k* and *l* indices run from 0 to 63, inclusive, while *i* and *j* run only from 1 to 16 inclusive. This also means that the DCT matrix *D* elements used are *i* from 1..16 and all *j* from 0..63. Writing out $B = D\,A\,D^t$, the occupied parts of the output are 1..16 x 1..16 (shaded dark grey), the utilized parts of the DCT matrix are 1..16 x 0..63 (shaded dark grey), and all of the input (shaded dark grey).



Also note that in the reference implementation, the 1..16 indices are slipped down to 0..15.

**Factored implementation:** note that the arithmetic operation-count in the above double-sum equation is 16x16x64x64, where *i* and *j* take 16 values, and *k* and *l* 64. Operation-counts are reduced by first computing $T = D\,A$, then $B = T\,D^t$.

**Butterflied implementation, not helpful:** Note that due to symmetry of the cosine function there are not *n* x *n* distinct values of the cosine. The naïve matrix multiplication could be butterflied out for an additional performance optimization, using for example Lee's algorithm (Lee, 1984). However, these optimized algorithms rely on length *n* input as well as output to reduce the operation-count for a single-vector transformation from $n^2$ to *n* log *n*. Since PDQ retains only 16x16 output from 64x64 input, there is already a reduction in operation count. During the development of the reference implementation, the following was done: do full Lee transforms down columns (64 columns each of length 64), then do Lee transforms along desired output rows (16 rows each of length 64). Testing showed that this did not perform better than the factored-and-trimmed implementation described above; in fact, slightly worse.

**Performance note**: as will be discussed below, time in the DCT is secondary to the downsample for larger images; it becomes the primary time-consumer for smaller images – in particular, video frames. This difference is because the downsample step depends on the size of the original input; the DCT step is always on a 64x64.

### MEDIAN-VALUE CALCULATION

The Torben algorithm (Mogensen, 1998) is used for computing the median value of the 16x16 DCT outputs.

### HASHES FOR DIHEDRAL TRANSFORMATIONS

Given the PDQ hash of a photo, one cannot derive directly from that hash the other hashes that *would* have been produced if the photo were rotated/flipped and then hashed. However, backing up just one step in the processing algorithm, operating on the DCT output matrix, simple transforms allow us to produce the hashes of the transformed images, at very little additional computational cost – far less than the more-than-8x cost for transforming an image and then hashing its transforms.

## COMPARISON OF PDQ TO PHASH

PDQ is, as noted above, inspired by pHash (Zauner, 2010). The differences are at the algorithm as well as the reference-implementation level, as follows:

- *Algorithm*: PDQ uses 256 output bits in contrast to pHash's 64. This is because 64 bits is uncomfortably small for web-scale image-matching. (Recall that the median step means each hash has half its bits 0 and half 1: the maximum number of possible pHash values is not $2^{64}$ but rather 64 choose 32, and less in practice.)
- *Implementation*: PDQ's reference design uses the CImg library (http://cimg.eu) only for unpacking various image formats (JPEG, PNG, etc.) to matrices of RGB triples – and even then, only as a sample image-file-decoding library. Companies are encouraged to replace the decoder with whatever makes the most sense within their own installation. The reference design is a from-scratch implementation using performance optimizations dedicated to PDQ.
- *Implementation*: there is a single calculation of luminance (Y), rather than computing YUV and then keeping only the Y channel.
- *Implementation*: Explicit one-dimensional box filters are coded with highly optimized sliding-window sums.

- *Algorithm*: The two-pass Jarosz filtering, and the choice of window size, means that all the pixels in each image block contribute to the downsample pixel for the block. pHash’s 7x7 omits a large amount of input data.
- *Algorithm*: Window sizes for PDQ filters are scaled to image dimensions, rather than hard-coded at 7x7 regardless of input size as pHash does. This is aimed at scale-invariance (e.g. a 4MP and a 1MP image which are perceptually similar).
- *Algorithm*: Downsample pixels are taken from the centers – rather than the corners -- of the (tent-filtered) blocks. This is intended to avoid edge-crop effects. (For example, suppose you screenshot an image and look it up – we should not overdepend on the very edgemost pixels which in turn depend on the precise placement of your mouse pointer as you do the screenshot.)
- *Implementation*: The trimmed/factored DCT is faster than a pair of simple 64x64 matrix multiplies followed by retaining a 16x16 subsample of the DCT output.
- *Implementation*: Torben’s algorithm is used for computing medians.
- *Algorithm*: PDQ features a quality metric, as discussed above.
- *Algorithm*: We efficiently compute all eight dihedral-transform hashes – the hashes one would have gotten for rotations and flips of the image – at fractional additional compute cost.

## PAQ AND PGQ ALGORITHMS

Just as PDQ is an extension/improvement to pHash, PAQ and PGQ are extensions/improvements to aHash and dHash, respectively (Zauner, 2010). **PAQ and PGQ are discussed here solely to contrast with PDQ**, and to motivate the need for the DCT pass (which has nonzero computational cost). Below in this document we will compare/contrast algorithms, discovering why PDQ is proposed while PAQ and PGQ are not.

### PAQ/PGQ ALGORITHM DETAILS

The PAQ algorithm is simple:

- Luminance, Jarosz-filter (Jarosz, 2001), and decimate just as with PDQ – but downsample directly to 16x16.
- Find the median value in the downsampled image. (Here we advocate for the use of median rather than mean since the median statistic is robust with respect to outliers: if there is a localized flare/logo overlay/etc. affecting the brightest part of the image then the mean shifts but the median does not.)
- For each of the 256 bits of the output hash, emit a 1 if the corresponding downsample pixel is greater than the median (**above**, hence the A in PAQ), else 0.
- The quality metric is similar to PDQ's.

Likewise, PGQ is simple:

- Luminance, Jarosz-filter, and decimate just as with PDQ – but downsample directly to 8x8.
- For each 7x8 horizontal-neighbor pair, emit a 1 if the left is greater than the right, else 0. For each 8x7 vertical-neighbor pairs, emit a 1 if the top is greater than the bottom, else 0. These are horizontal and vertical **gradients**; hence the G in PGQ.
- The total number of output bits is 112, although within the PDQ hash reference implementation this is stored in a 256-bit value (mostly zeroes) for tooling convenience.
- The quality metric is similar to PDQ's.

Benefits of PAQ and PGQ over PDQ:

- Both are faster to compute than PDQ, since they don't require a DCT. However, all three have the same downsample logic, which is the most time-consuming step for larger images – so this performance improvement is not always a solid win for PAQ/PGQ.
- For both it is trivial to obtain from the hash itself the hash of rotated/flipped images.

### DIFFERENCES BETWEEN PAQ/PGQ AND AHASH/DHASH

- Better Jarosz-filtering is used for both (as with PDQ vs. pHash).
- PAQ uses median, not mean as aHash does, as discussed above.
- PGQ uses vertical as well as horizontal neighbors while dHash uses horizontal only.
- PAQ and PGQ produce more than 64 bits of output (256 and 112 bits, respectively).

## EVALUATION

### SUMMARY OF PRECISION AND RECALL CHARACTERISTICS

First note what precision and recall are:

- **Precision** is the ratio of true positives to true positives plus false positives: **TP/(TP+FP)**. It measures **how often our detection is wrong**.
- **Recall** is the ratio of true positives to false negatives: **TP/(TP+FN)**. It measures **how often we miss detecting** what we want to detect.

| | Algorithm says they match | Algorithm says they do not match |
|---|---|---|
| Should match | True positive | False negative |
| Should not match | False positive | True negative |

| **Precision** | Algorithm says they match | Algorithm says they do not match |
|---|---|---|
| Should match | True positive | False negative |
| Should not match | False positive | True negative |

| **Recall** | Algorithm says they match | Algorithm says they do not match |
|---|---|---|
| Should match | True positive | False negative |
| Should not match | False positive | True negative |

Measuring precision and recall **is – and must be – an iterative** process here. Like many image-hashing algorithms, TMK+PDQF and PDQ have tunable matching-distance thresholds.

- **If the threshold is set high** – for example, well above the average distance between random pairs of images – then everything will match everything. **Recall** will be **good** (vacuously – we found the match! And everything else too!) but **precision** will be **abysmal** (almost all matches will be undesirable).
- **If the threshold is set low** – for example, at or near zero – then even minor differences will cause a mismatch. **Recall** will be **low** (we didn't find much!) but **precision** will be **great** (what we did find, we're very sure of!).

Our goals for selecting distance thresholds are twofold:

- First, it is up to the user to define thresholds which are appropriate for their use-case – for example, whether precision or recall is more important. Here in this document we need to establish landmarks and **guidelines** to help users tune their system.
- Second, it is important to see how **consistent** these thresholds are. If – at various thresholds – an algorithm produces false positives in some category of image pairs but false negatives in another category, then the user will have no good options.

As a disclaimer, and to set expectations, let's first reiterate the class of matching algorithms we're working with here: TMK+PDQF and PDQ, like many algorithms, balance performance against recall. More CPU-intensive, training-intensive, semantic-matching algorithms will of course be able to equate less and less similar media pairs, such as deep image-crops, rotations, shears, different photos of the same person, and so on.

## TMK EVALUATION

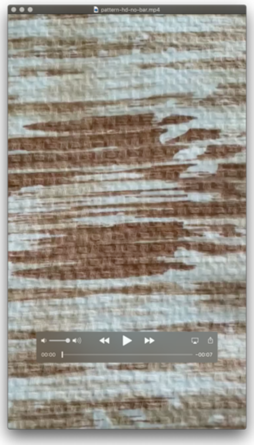
- We start with a small-scale set of 15 videos delivered along with the open-source GitHub repository.
- We then look at pair-score distributions on a larger dataset of 2800 videos.
- Finally, we describe the logic used on an even larger dataset of 40,000 videos which was used to determine the suggested level-1 and level-2 thresholds of **0.7**.)

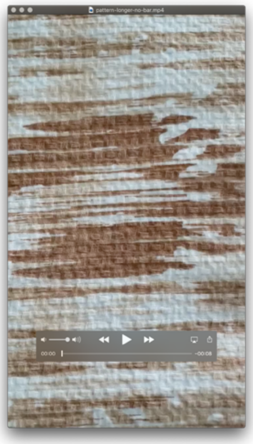
### MODIFIED SAMPLE VIDEOS

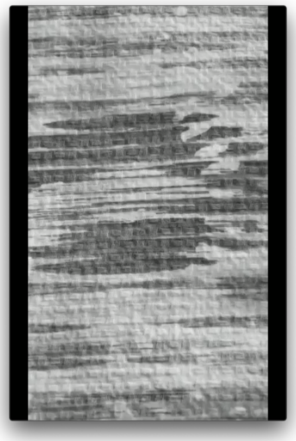
As found in the GitHub repo there are 15 videos with color modifications, minor time-trimming, small and large logo overlays, and horizontal sidebars. There are three originals: one of chair, another of a doorknob, and a third of a fabric pattern. Initial frames are as follows:

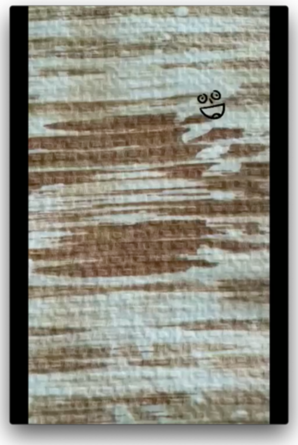
- chair-orig-22-fhd-no-bar.mp4 – Original 22 seconds, FHD 1080x1920
- chair-orig-22-hd-no-bar.mp4 – Same as previous but HD 720x1280
- chair-orig-22-sd-bar.mp4 – Same as previous but SD 480x720 and with black sidebars
- chair-22-sd-grey-bar.mp4 – Same as previous but greyscale
- chair-22-sd-sepia-bar.mp4 – Same as chair-orig-22-sd-bar.mp4 but sepia
- chair-20-sd-bar.mp4 -- Same as chair-orig-22-sd-bar.mp4 but with the first two seconds removed
- chair-19-sd-bar.mp4 -- Same as chair-orig-22-sd-bar.mp4 but with the first three seconds removed
- chair-22-with-small-logo-bar.mp4 -- Same as chair-orig-22-sd-bar.mp4 but with small logo added
- chair-22-with-large-logo-bar.mp4 -- Same as chair-orig-22-sd-bar.mp4 but with large logo added
- doorknob-hd-no-bar.mp4 – No other variants shown
- pattern-hd-no-bar.mp4 – 7 seconds 720x1280
- pattern-longer-no-bar.mp4 – Same as pattern-hd-no-bar.mp4 but one second longer
- pattern-sd-grey-bar.mp4 – Same as pattern-hd-no-bar.mp4 but SD 480x720 and with black sidebars
- pattern-sd-with-small-logo-bar.mp4 – Same as pattern-hd-no-bar.mp4 but SD 480x720 and with small logo
- patttern-sd-with-large-logo-bar.mp4 – Same as pattern-hd-no-bar.mp4 but SD 480x720 and with large logo

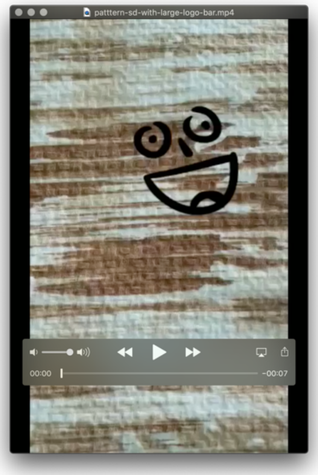
Fabric pattern initial frames:



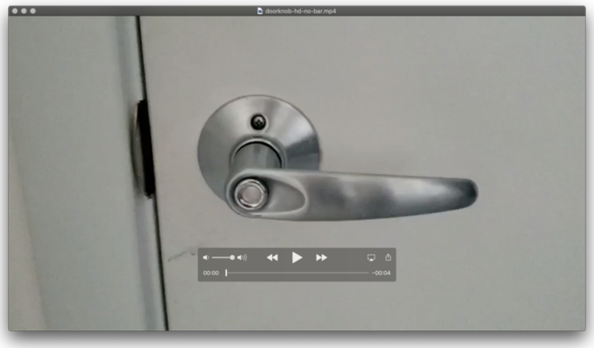




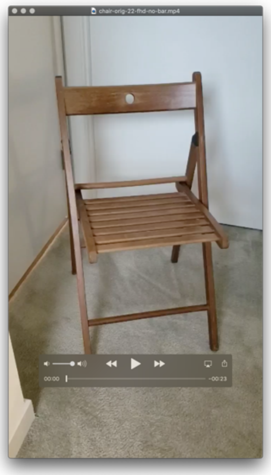


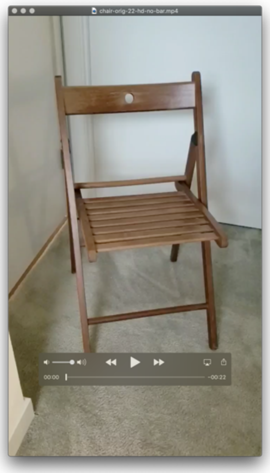


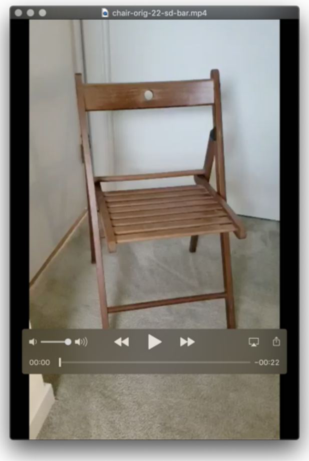
Doorknob initial frame:

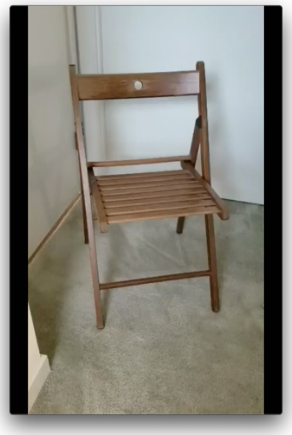


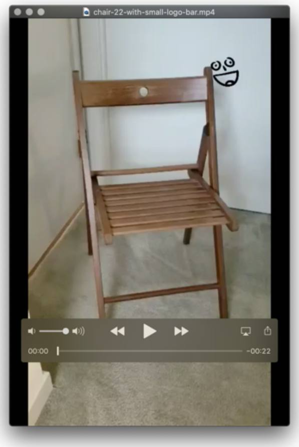
Chair initial frames:

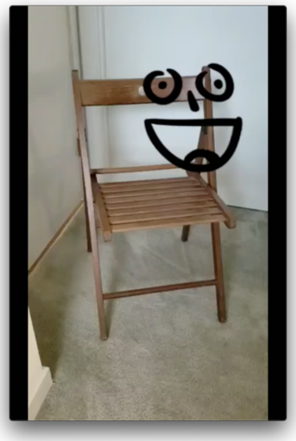


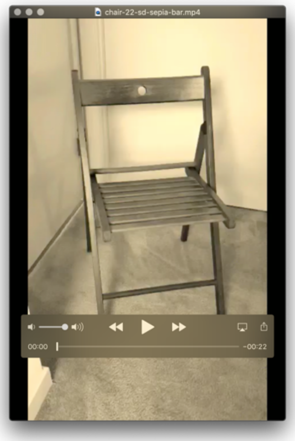


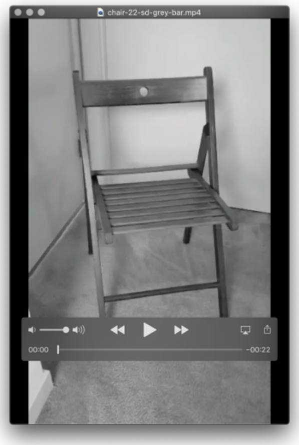


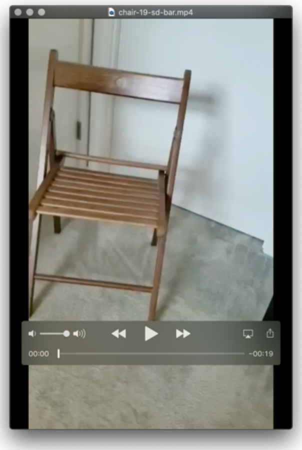












## CLUSTERING

Using default level-1 and level-2 thresholds of 0.7 we see the following (note the Miller data-manipulation tool, mlr, is standard on most modern Linux distributions, as well as MacOS via brew install miller):

```
$ tmk-clusterize --c1 0.7 --c2 0.7 *.tmk | mlr --opprint cat

clidx clusz filename
1   8   chair-19-sd-bar.tmk
1   8   chair-20-sd-bar.tmk
1   8   chair-22-sd-grey-bar.tmk
1   8   chair-22-sd-sepia-bar.tmk
1   8   chair-22-with-small-logo-bar.tmk
1   8   chair-orig-22-fhd-no-bar.tmk
1   8   chair-orig-22-hd-no-bar.tmk
1   8   chair-orig-22-sd-bar.tmk

2   1   chair-22-with-large-logo-bar.tmk

3   1   doorknob-hd-no-bar.tmk

4   4   pattern-hd-no-bar.tmk
4   4   pattern-longer-no-bar.tmk
4   4   pattern-sd-grey-bar.tmk
4   4   pattern-sd-with-small-logo-bar.tmk

5   1   pattern-sd-with-large-logo-bar.tmk
```

Using conservative level-1 and level-2 thresholds of 0.9 we see the following:

```
$ tmk-clusterize --c1 .9 --c2 .9 *.tmk | mlr --opprint cat

clidx clusz filename

1   6   chair-19-sd-bar.tmk
1   6   chair-20-sd-bar.tmk
1   6   chair-22-sd-grey-bar.tmk
1   6   chair-22-sd-sepia-bar.tmk
1   6   chair-22-with-small-logo-bar.tmk
1   6   chair-orig-22-sd-bar.tmk

2   1   chair-22-with-large-logo-bar.tmk

3   2   chair-orig-22-fhd-no-bar.tmk
3   2   chair-orig-22-hd-no-bar.tmk

4   1   doorknob-hd-no-bar.tmk
```

| | | |
|---|---|---|
| 5 | 2 | pattern-hd-no-bar.tmk |
| 5 | 2 | pattern-longer-no-bar.tmk |
| 6 | 2 | pattern-sd-grey-bar.tmk |
| 6 | 2 | pattern-sd-with-small-logo-bar.tmk |
| 7 | 1 | pattern-sd-with-large-logo-bar.tmk |

**Analysis:**

- The doorknob singleton is equated with nothing else, of course.
- At looser or tighter tolerance, all chair variants are equated with one another, and likewise with the pattern variants, with the exception of the large-logo variants which have too much manipulation.
- At looser tolerance, chair and pattern variants with and without black bar are equated; at tighter tolerance, the barred and unbarred chair and pattern variants are split. Note however that these black bars are very small; in general, we do not expect TMK+PDQF to equate variants with/without thick borders, and likewise with non-trivial crops.

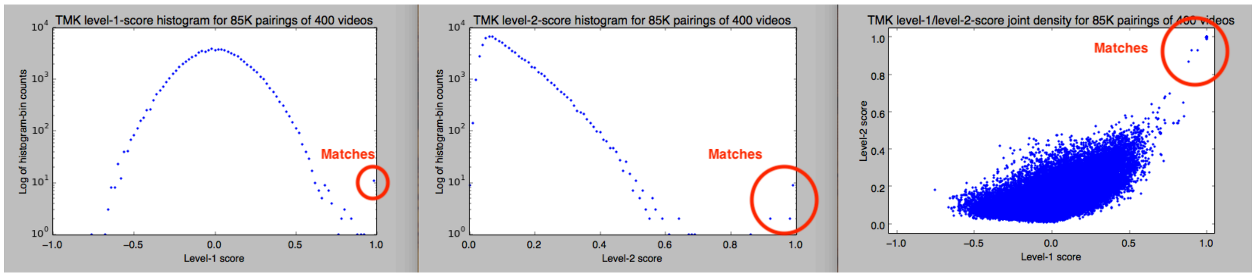
## PAIR-SCORE DISTRIBUTIONS

We took a sample of 2,679 videos of objectionable content, mostly distinct but with several similarity variants in the database. Out of *n* videos there are $n*(n-1)/2$ pair -- for all 2,679 video hashes within this particular dataset, that's over 3 million pair-scores to compute. So we took a random sample of about 400, resulting in about 85,000 possible distinct pairs. Here we are plotting histograms of the level-1 scores, the level-2 scores, and the joint density.

Remember from above we use the level-2 score to decide if two videos match or not, with the level-1 score as the initial gate. There should be a gap between the matches and the non-matches. For this sample dataset, we can see in the left and middle plots that there clearly is.

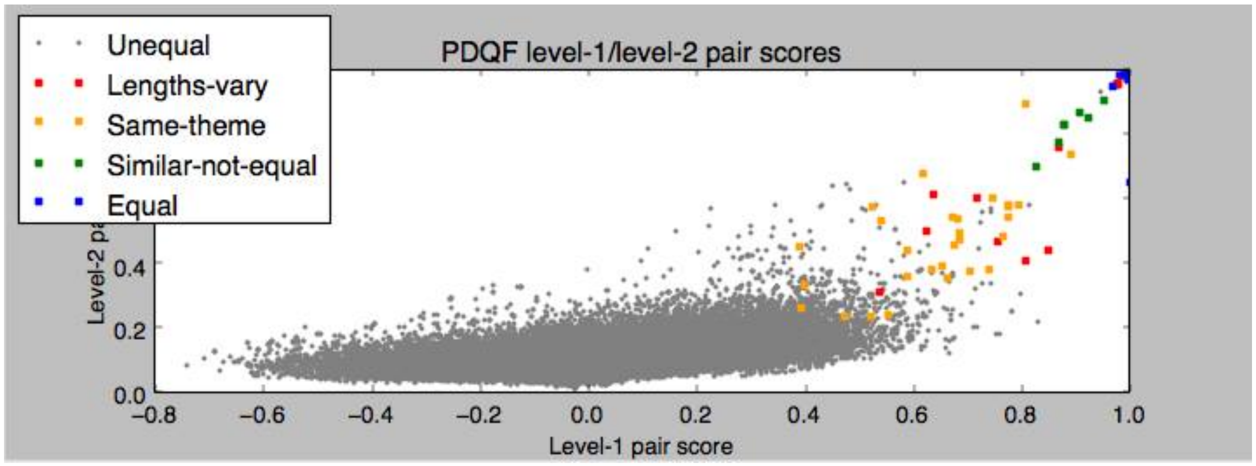
Since level-1 and level-2 scores of 1 are best, we can see dots for matching videos in the upper-right-hand corner of the right-hand plot, whereas the big blue blob in that plot is the distribution of the pair-scores of unrelated videos.

Also remember from above that the level-2 pair-score is a bit expensive to compute so we use the level-1 pair-score to see if we should bother – so in particular the level-2 pair-score is not computed if the level-1 pair-score is under threshold. It would be bad if there were pairs of videos with high level-2 pair-score but low level-1 pair-score -- those would be false negatives. For this plot, we're computing the level-2 score regardless of the level-1 score and we can see from the right-hand plot that for this sample dataset we aren't 'missing' any matches this way.



## CLUSTERING LIMITS

To look at the limits of TMK+PDQF matching, here's another dataset of several hundred videos, containing more aggressive variants of several originals. They were hand-labeled by whether variants were visually equal, visually similar but distinguishable, visually similar but of varying lengths (some timewise editing had been done on some variants), same theme (different studio edits of the same material), or completely unequal.



Here we see several things:

- The grey blob is the distribution of pair-scores for unrelated videos.
- We see empirical confirmation of **0.7** as level-1 and level-2 thresholds for similarity.
- We see evidence that TMK+PDQF as currently construed is truly a copy-detection algorithm: strong timewise edits have a detected level of dissimilarity approaching that of unrelated videos. (There are yellow dots among the grey.)

## DETERMINATION OF SCORING THRESHOLDS

- We took at set of 36,000 videos
- Again using $n(n-1)/2$ the number of distinct pairs is prohibitively large for exhaustive search so we took a random sample of 40,000 pairs. (Scoring thresholds and indexing systems can be used to effectively search datasets of this size and larger, of course, but in this section we are discussing the kinds of brute-force queries needed to *determine* the thresholds necessary for a sublinear-time indexing algorithm.)

- From among all the 40,000 pairings we bucketed the level-1 and level-2 pair-scores by steps of 0.1 – e.g. for each video on the left-hand side of a pair, look at matches with level-1 or level-2 score in the buckets …, 0.1 to 0.2, …, 0.9 to 1.0
- Look for false positives at too-low threshold and false negatives at too-high threshold.
- We conclude that 0.7 is an acceptable threshold for both level-1 and level-2 scores.
- At these thresholds, from among the 36,000 sample videos, we found 12,000 videos participating in non-trivial clusters (i.e. of size greater than 1), with 2,800 non-trivial similarity clusters.

## PDQ EVALUATION

Summary of scope:

- PDQ does not handle deep crops; this is outside its purview as a fast syntactic hasher.
- PDQ has a quality metric available for low-feature-image detection.
- PDQ is robust to some light image modifications such as small logos or lens flares.
- PDQ is designed to prefer precision over recall: when it declares a match at low threshold, we trust the match; when they do not, it is either because the images are perceptually different, or because they are perceptually similar beyond the range of the algorithm: for example, one image is a deep crop of another.

Summary of evaluation plan:

- Select image datasets; from those, retain in-scope image clusters (e.g. exclude deep crops).
- We will first place obvious upper and lower bounds of the matching-distance threshold for PDQ, then refine that.
- Only then can we use a matching-distance threshold to draw a line, and test for consistency of precision as well as recall across that line.

### EVALUATION DATASETS

- CopyDays (INRIA, 2006) is specifically designed for copy-detection: images are re-encoded for varying JPEG quality, downsized, and cropped. There are 157 original images, each with about a dozen systematic modifications.
- LabelMe (Laboratory, MIT Computer Science and Artificial Intelligence, 2005) is a dataset intended primarily for higher-level object-recognition algorithms. (There are subdirectories `train` and `test`, specifically for ML algorithms.) Nonetheless, the dataset offers opportunities for duplicate-detection. In particular, similar images here are similar for different reasons than in CopyDays: for example, different people walking through the same doorway, or images of the same person taken just a second apart. These play a similar role to video-frame taps. This is a much larger dataset than CopyDays: 50,000 images total.
- Facebook abuse-detection data, primarily frame taps at 2.5FPS from published and detected extremist-propaganda videos. This is a total of about 5,000 images.
- Frame-taps from sample videos at Pexels (http://www.pexels.com) at 4FPS. This is a total of about a thousand images.

### CATEGORIES OF IMAGES IN EVALUATION DATASETS, AND EXPECTATIONS

The above datasets offer multiple opportunities:

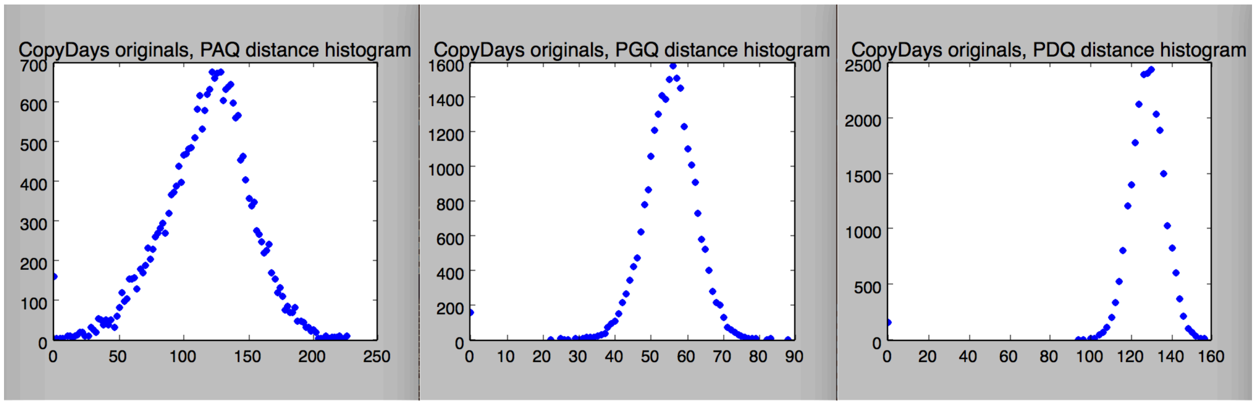
- The CopyDays dataset includes a subdirectory named `original`. These photos are all **perceptually distinct images**. As a very minimum requirement, **any matching algorithm should clearly separate these**.
- CopyDays also includes images modified by **various JPEG qualities**.
  - For **minor** image-quality differences, syntactic matchers in the category of PDQ **should solidly detect matches**.

    - For the very **most** degraded quality-reducing modifications, **some mismatches will not be surprising**.
- CopyDays includes a subdirectory named `strong`. These are adversarially modified images – outside of the scope of PDQ.
- LabelMe includes a large number of **related images**: not the same image being modified but different images which happen to be similar. For example, different people walking through the same doorway, or images of the same person taken just a second apart. **Perceptually similar images should be matched.**
- Likewise, **video-frame taps** include a large number of similar images, **which should be clustered together.** The degree of matching should correspond to the match-distance threshold.
- CopyDays includes **deep-crop images** which **PDQ doesn't detect as similar**.
- Lastly, **logo-overlay images should ideally be detected as similar**.

## UPPER BOUND ON MATCHING-DISTANCE THRESHOLD

Given the above image categories, we start with the most unambiguous: original images from CopyDays, all of which are **perceptually quite distinct**. There are only 157 such so it's quite feasible to compute distances between all pairs of images.

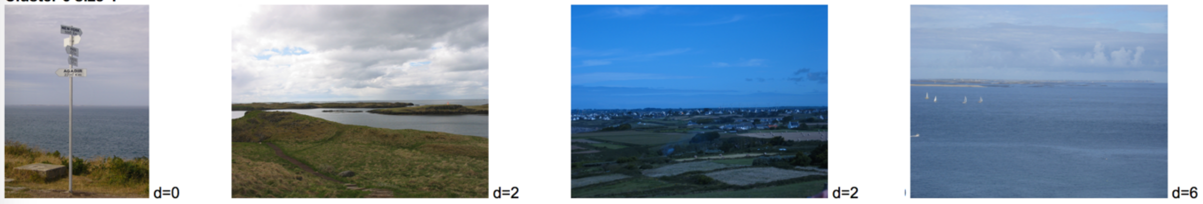
Computing histograms we get the following:



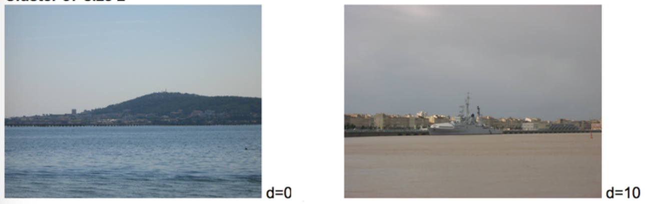
Since we are computing distances between all pairs of (CopyDays original) images, including each image compared to itself, we see a histogram dot at 0. Then we expect a clear gap to all the distances between pairs of distinct images.

Already it's clear that PAQ will not do. There is no clear separation between images which should be made distinct. Looking at some examples reveals why:

Cluster 6 size 4



Cluster 57 size 2



Looking at the hashes gives more insight:

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1    0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1    0 0 0 0 0 0 0 0 0 1 1 1 0 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Since PAQ tracks only a single bit for luminance (in the downsampled image) above the median or not, it isn't able to track how *much* above or not. Further, information about each downsample pixel is independent of its neighbors, except inasmuch as they are all connected to the median luminance.

Using PGQ and PDQ on these six images, we get the following matrices of distances for all pairs of the six images. (The diagonal is all zeroes, of course, since each image is identical to itself.)

```
PGQ                             PDQ
   0  51  46  41  53  51          0 122 140 134 128 136
  51   0  35  50  46  46        122   0 130 128 130 130
  46  35   0  45  33  43        140 130   0 136 122 116
  41  50  45   0  42  22        134 128 136   0 150 124
  53  46  33  42   0  34        128 130 122 150   0 124
  51  46  43  22  34   0        136 130 116 124 124   0
```

With reference to the above histograms for distances of distinct images, we see that the images are all successfully separated, by significant distances. Looking at the bits we see the following:

```
1 1 1 1 1 0 0 1 0 0 0 1 0 1 1 1   0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 1   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1   1 1 1 1 1 1 1 1 0 1 1 0 0 1 1 1   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1   1 0 0 0 0 1 0 0 0 0 1 1 1 1 1 1   1 1 0 0 0 1 1 1 0 1 0 1 1 1 1 1   0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
1 1 1 1 0 1 0 0 0 0 1 0 0 0 0 1   0 0 1 1 1 1 1 1 0 1 1 0 1 0 0 1   0 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0   0 1 1 1 1 1 0 0 1 1 1 1 1 0 0 1
1 1 1 1 1 1 0 1 1 0 1 1 0 1 0 0   1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 1   0 0 0 1 1 1 0 1 0 1 1 1 1 1 1 0   0 1 0 1 1 1 0 1 1 0 1 1 1 0 0 0
0 1 0 1 1 1 1 1 1 0 1 1 0 0 0 0   0 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1   0 1 0 0 0 1 1 1 1 0 0 1 1 1 1 1   1 0 0 0 0 1 1 0 0 0 0 0 1 1 1 1
0 1 1 1 0 0 0 0 1 0 1 1 0 1 1 0   0 1 1 0 1 0 0 1 0 0 0 0 1 1 0 0   0 1 0 0 1 1 1 1 1 1 0 0 0 1 0 1   1 1 1 1 0 0 0 1 1 0 0 0 0 0 1 1

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1   PGQ
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1   0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1   1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1
0 0 0 1 1 1 0 0 0 0 1 1 1 1 0 0   0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0
1 1 0 0 0 1 1 1 1 1 1 1 1 0 1 1   1 0 0 0 0 1 1 0 1 0 0 0 1 1 1 1
1 1 1 1 0 1 0 1 0 1 1 1 0 0 1 1   1 1 1 1 1 0 0 1 1 1 1 1 1 0 1 1
```

```
1 1 0 0 1 1 1 0 0 1 1 0 0 1 1 0   0 0 1 0 1 0 1 0 0 1 1 1 0 0 1 1   1 1 0 0 1 1 0 1 1 1 0 0 0 0 0 1   0 0 1 0 0 1 1 0 0 1 0 0 1 1 1 1
0 0 1 1 0 0 0 1 1 1 1 0 0 1 0 1   0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1   0 0 0 0 1 1 1 0 0 0 1 0 0 1 0 0   1 1 0 1 1 1 0 1 0 0 0 0 0 0 1 1
1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1   1 0 1 1 1 1 0 1 1 1 1 1 1 1 0 0   1 1 1 1 0 1 0 1 1 1 1 1 1 0 1 0   1 1 0 1 1 1 1 0 1 1 1 0 1 0 0 0
1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1   1 1 1 1 0 1 0 1 1 1 1 0 1 0 0 1   1 0 0 1 1 0 1 0 1 1 1 0 0 0 1 1   0 1 0 0 0 0 1 1 0 0 1 0 0 0 0 0
1 0 1 1 1 0 0 1 1 0 1 0 0 0 0 1   1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1   0 1 0 0 0 1 1 0 0 1 0 0 0 1 0 1   0 1 0 0 0 0 1 0 0 0 1 0 1 0 1 0
0 1 0 0 0 1 1 1 0 0 0 1 1 0 0 0   1 0 1 0 0 0 1 0 0 1 1 0 0 1 0 0   1 0 1 0 1 0 0 1 1 0 1 1 0 0 1 1   1 1 1 1 1 1 1 1 0 1 1 1 0 0 1 1
0 0 1 1 0 0 1 1 1 0 1 0 0 1 0 1   0 1 1 1 1 1 1 0 1 1 1 1 0 1 0 0   0 1 1 1 0 0 0 0 0 0 1 1 0 1 0 1   0 0 1 0 1 1 0 0 1 1 0 1 1 1 0 1
0 1 1 0 0 1 1 1 0 1 0 1 1 0 0 0   0 0 1 0 1 0 0 0 1 1 1 1 1 0 1 1   1 0 0 0 0 1 1 1 1 0 0 0 1 0 1 1   0 0 1 0 0 0 0 0 1 1 0 1 0 1 0 0
1 1 0 1 0 0 1 1 1 0 0 1 1 1 0 1   0 0 1 0 0 0 0 0 0 1 0 0 0 1 1 0   1 0 1 1 0 1 1 1 0 0 1 1 0 0 1 0   1 0 1 0 1 0 0 0 1 1 0 1 1 1 0 1
0 0 1 0 0 0 0 1 0 0 1 1 0 0 0 0   0 1 1 1 1 0 1 1 0 1 1 1 1 1 0 0   1 1 0 1 1 0 0 0 1 1 0 0 1 1 0 1   1 0 1 1 1 0 1 1 1 0 0 1 0 0 0 0
0 1 1 1 1 0 1 1 1 0 0 1 1 1 0 0   0 1 0 1 1 0 0 1 1 1 1 0 0 0 0 1   0 0 1 1 0 1 1 0 0 1 1 1 0 0 1 1   1 1 0 1 0 1 1 1 0 0 1 0 1 0 0 1
0 0 0 1 0 0 0 1 0 0 1 0 0 0 1 1   0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1   1 0 1 0 0 1 1 1 1 0 0 1 1 0 0 0   1 1 0 1 1 0 1 1 1 1 1 1 1 0 0 1
0 1 1 0 1 1 1 0 0 1 0 1 1 1 0 0   1 0 1 1 0 0 0 1 0 1 0 1 1 1 0 0   0 1 0 0 1 0 0 0 1 1 1 0 0 1 1 0   0 0 0 0 0 0 1 0 0 0 1 0 0 1 1 0
0 0 0 1 0 1 1 0 0 0 1 0 0 0 1 1   1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1   0 1 1 0 0 1 0 0 0 0 0 1 1 0 1 0   0 0 1 0 1 1 1 0 0 1 1 0 0 1 1 0
1 1 0 0 1 1 0 0 0 1 1 0 1 1 1 1   1 0 0 1 1 1 0 1 0 0 1 0 0 0 1 0   1 0 0 1 0 0 0 1 1 1 1 0 0 1 1 1   0 1 1 1 1 1 1 0 1 0 0 1 1 1 1 0
1 1 1 0 1 1 1 0 1 1 1 0 0 1 1 0   0 1 0 0 0 0 1 0 1 0 0 0 0 1 1 0   0 1 0 0 1 1 1 0 0 1 0 1 0 1 0 0   0 0 0 0 1 0 0 0 0 1 1 0 1 0 1 0

0 0 1 1 1 1 0 0 1 0 1 1 1 0 0 0   0 1 0 1 1 0 0 1 1 1 0 1 1 1 1 1
1 0 1 0 0 0 0 1 0 1 0 0 1 1 0 0   0 0 0 1 1 0 0 1 1 1 0 0 0 0 1 1   PDQ
1 0 1 1 1 1 0 0 0 0 1 0 0 1 1 1   1 0 1 0 0 1 0 0 1 0 1 0 0 0 0 0
1 1 0 1 1 0 1 1 1 0 0 1 0 0 1 1   0 1 1 0 0 1 1 0 0 1 1 0 0 0 1 0
0 0 1 1 0 1 0 0 1 1 0 1 0 0 0 0   0 1 0 1 1 0 1 1 1 1 0 0 0 1 1 1
1 1 1 0 1 1 0 1 1 0 0 0 1 1 0 0   1 0 1 0 1 1 0 0 1 0 1 1 0 0 0 0
0 1 0 0 0 0 1 1 0 1 1 0 0 1 1 1   1 1 1 0 0 0 0 0 0 0 1 1 0 0 0 1
0 0 0 1 1 1 1 0 1 0 1 1 0 0 1 1   0 1 0 1 1 0 1 1 1 1 0 0 0 1 1 1
1 1 0 1 0 0 0 1 1 1 0 1 1 0 0 0   1 0 1 1 1 1 0 0 1 0 0 0 1 0 0 0
1 1 1 0 1 1 0 0 1 1 0 0 1 1 0 0   1 1 1 0 0 0 0 0 0 0 1 1 0 0 0 0
0 0 0 1 0 0 0 1 0 1 1 0 0 1 1 1   0 1 0 1 0 0 1 1 0 1 0 0 0 1 1 1
1 0 1 1 1 0 1 0 0 0 1 1 0 0 1 1   1 0 1 1 1 1 0 1 1 1 0 0 1 1 0 0
0 1 0 0 1 0 0 1 1 0 0 1 1 0 0 0   1 1 1 0 0 1 0 0 1 0 1 1 0 0 0 0
1 1 1 1 0 0 0 0 1 1 0 0 1 1 1 0   0 1 0 1 0 0 0 1 0 1 0 1 0 1 1 1
0 0 1 0 0 1 0 0 0 1 1 0 0 1 1 1   0 0 0 1 1 1 1 1 1 1 0 0 1 1 1 0
1 0 1 1 0 0 1 1 0 0 1 1 0 0 0 1   1 1 1 0 0 1 1 0 1 0 1 1 1 0 0 1
```

This is unsurprising: for PGQ, which involves nearest-neighbor horizontal and vertical gradients, information about adjacent downsample pixels is used to compute each output bit. There are nonetheless long runs of 0s and 1s. For PDQ, the DCT mixes information about **every** DCT-input downsample pixel into **every** DCT-output cell. This gives PDQ its matching strength: its essential **mixing property**. Further, as we will see below, the fact that various 0s and 1s are distributed throughout the PDQ hashes is utilized for efficient, sub-linear-time tree lookups of PDQ hashes.

From the histograms of the previous sections, we have the following information:

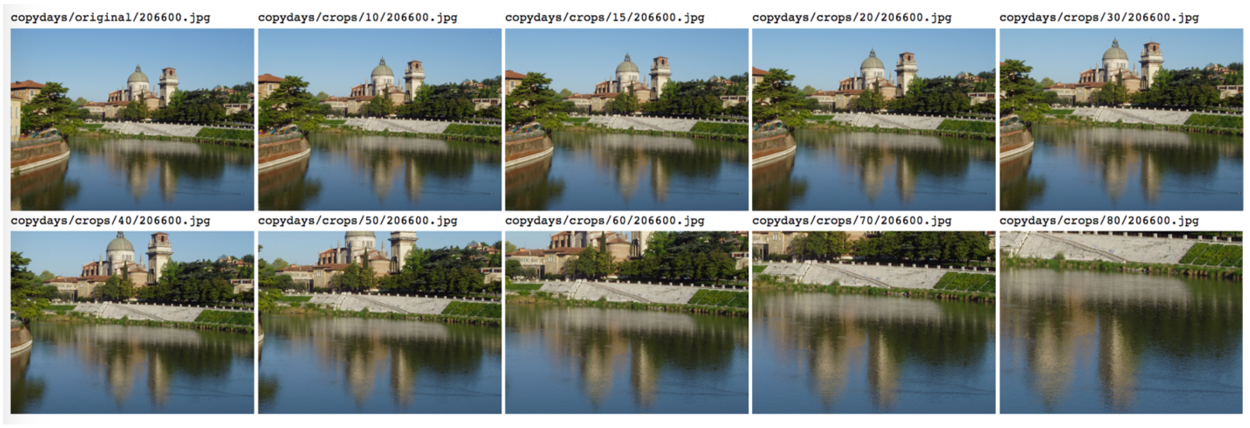
- PAQ is not to be used: there is no clear gap between distances of identical and unrelated images.
- PGQ has a gap of 1 to 21: no pairs of distinct images (in this initial, 157-image set) have those pairwise distances. Matching distances are tunable by users in their applications, of course, but already we know we certainly cannot recommend a tolerance level of 22 or more, and probably less if we look at a larger image-set.
- Similarly, for PDQ we certainly could not recommend a tolerance of more than 90, and probably less if we look at a larger image-set.

We're already done with PAQ in this document. For the reasons alluded to in the previous section – namely, PDQ's mixing property which facilitates an efficient hash-lookup data-structure and algorithm for PDQ, we'll focus on PDQ, leaving PGQ behind as well.

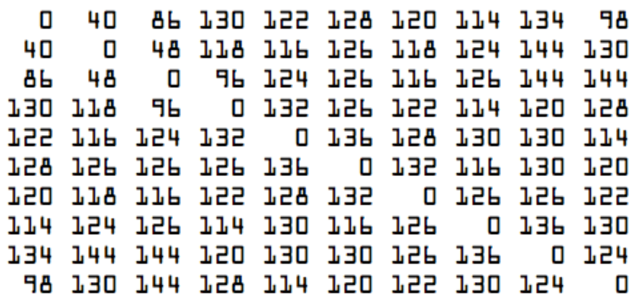
What we have now is an **upper bound** on our matching threshold: 90 for PDQ. To go further with a matching threshold, we need to enlarge our dataset to go beyond entirely distinct images to similar images. There are a lot of these in CopyDays as well as LabelMe, as well as in video-frame-tap data.

## CROPS: OUT OF SCOPE

Here's an example of crops in the CopyDays dataset:


copydays/original/206600.jpg
copydays/crops/10/206600.jpg
copydays/crops/15/206600.jpg
copydays/crops/20/206600.jpg
copydays/crops/30/206600.jpg
copydays/crops/40/206600.jpg
copydays/crops/50/206600.jpg
copydays/crops/60/206600.jpg
copydays/crops/70/206600.jpg
copydays/crops/80/206600.jpg

Here are pairwise distances for PDQ:


```
  0  40  86 130 122 128 120 114 134  98
 40   0  48 118 116 126 118 124 144 130
 86  48   0  96 124 126 116 126 144 144
130 118  96   0 132 126 122 114 120 128
122 116 124 132   0 136 128 130 130 114
128 126 126 126 136   0 132 116 130 120
120 118 116 122 128 132   0 126 126 122
114 124 126 114 130 116 126   0 136 130
134 144 144 120 130 130 126 136   0 124
 98 130 144 128 114 120 122 130 124   0
```

In the Design Goals section at the start of this document, we already threw in the towel on hard crops: they are outside the domain of fast, syntactic matchers such as PDQ. And here we see quantitative data on that: from the distance-histograms of the previous section we see that pairwise distances between these images are generally within the realm of distances between random images. Other algorithms work well in the cropping space. For the remainder of this evaluation, we'll restrict to non-crop cases.

As well, CopyDays has 'strong' modifications: namely, actively adversarial image modifications such as the following:

copydays/original/200000.jpg

copydays/strong/200001.jpg

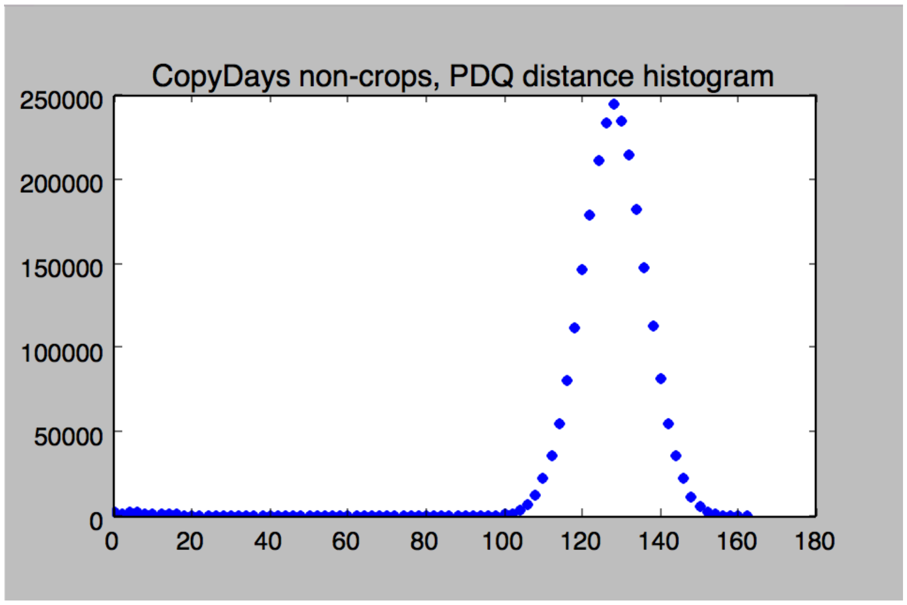
copydays/original/200100.jpg

copydays/strong/200102.jpg

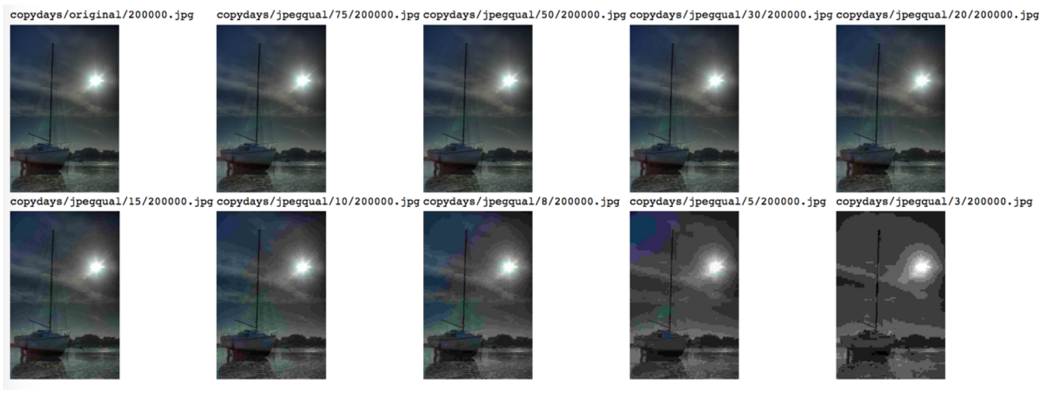
These, too, are far out of scope for PDQ.

## PROVISIONAL LOWER BOUND ON MATCHING-DISTANCE THRESHOLD

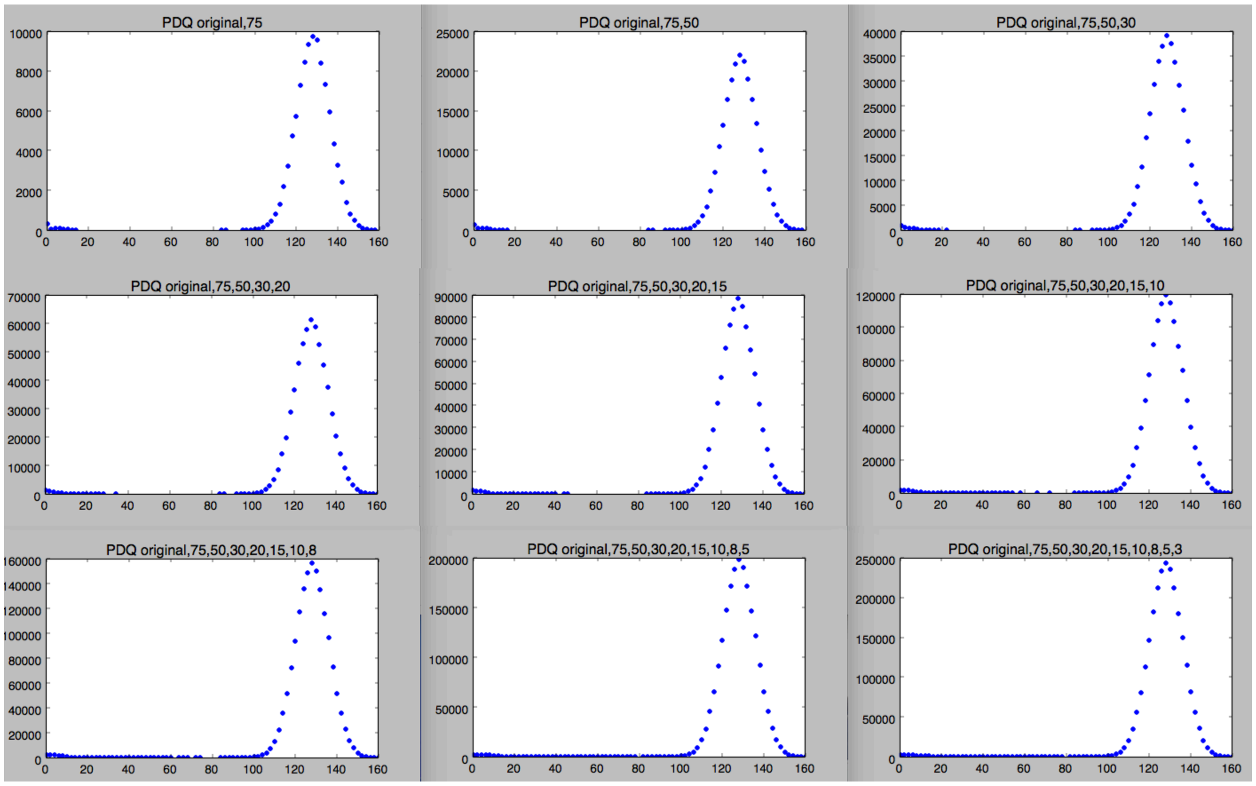
Picking out **all** non-crop, non-strong images from CopyDays, and sticking now with PDQ, we get the following pairwise-distance distributions.



Here it's clear that we have our work cut out: there's a range of pairwise distances from similar to dissimilar images. But these include JPEG qualities of 3, 5, 8, 10, 15, 20, 30, 50, and 75 – for example:


copydays/original/200000.jpg
copydays/jpegqual/75/200000.jpg
copydays/jpegqual/50/200000.jpg
copydays/jpegqual/30/200000.jpg
copydays/jpegqual/20/200000.jpg
copydays/jpegqual/15/200000.jpg
copydays/jpegqual/10/200000.jpg
copydays/jpegqual/8/200000.jpg
copydays/jpegqual/5/200000.jpg
copydays/jpegqual/3/200000.jpg

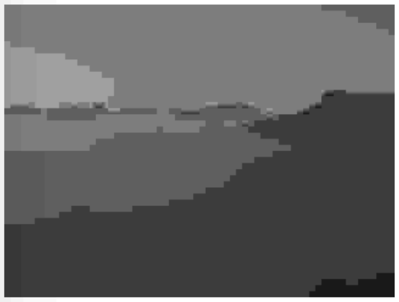
Plotting the distributions for larger and larger JPEG-quality reductions shows that the gap between the recall zone (on the left) and the precision zone (on the right) narrows down to nothing when we include the lowest-quality versions. Here are the distributions:

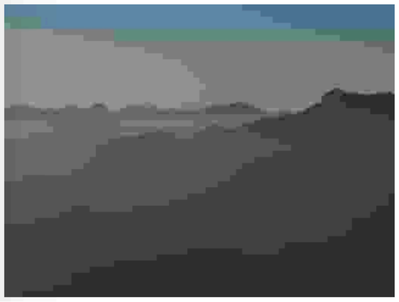


Above we found an upper bound of 90 for known-different images. The data we have here for known-similar images suggest a **provisional lower bound** of somewhere between 20-60 for the PDQ matching threshold. Again, enlarging our datasets even further will allow us to tighten these bounds even further. I say *provisional* because we need to decide which JPEG qualities we want to be matched. Looking at more data, we might wish to abandon matching at lower JPEG qualities, since this distance threshold might conflate interesting features in similar images which we'd rather keep distinct. If it takes a threshold of 50 to make a high-JPEG-quality and low-JPEG-quality version of the same image match, but a threshold of 25 to separate two photos of similar doorway scenes taken a few seconds apart, which parameter would we choose when we deploy our automation? This is up to the system implementer, depending on the particular use-case. Nonetheless a bit more work here in this document will help to develop guidelines.

Selecting out some of the images in the filled gap for PDQ (distance between 75 and 85) yields images such as the following:

hash-data/external/copydays/jpegqual/10/200300.jpg


hash-data/external/copydays/jpegqual/3/200300.jpg


hash-data/external/copydays/jpegqual/5/200300.jpg

These are unconcerning. The primary use-case for PDQ is detection of abusive content. These gap-filling elements (and others not screenshotted here) are not what we're after.

For brevity, a few such sets are displayed here; the rest are similar, at the very lowest-JPEG-quality end.

## CLUSTERING RESULTS FOR COPYDAYS JPEG-QUALITY TEST DATA

Here we simply tabulate and quantify the examples which concluded the previous section.

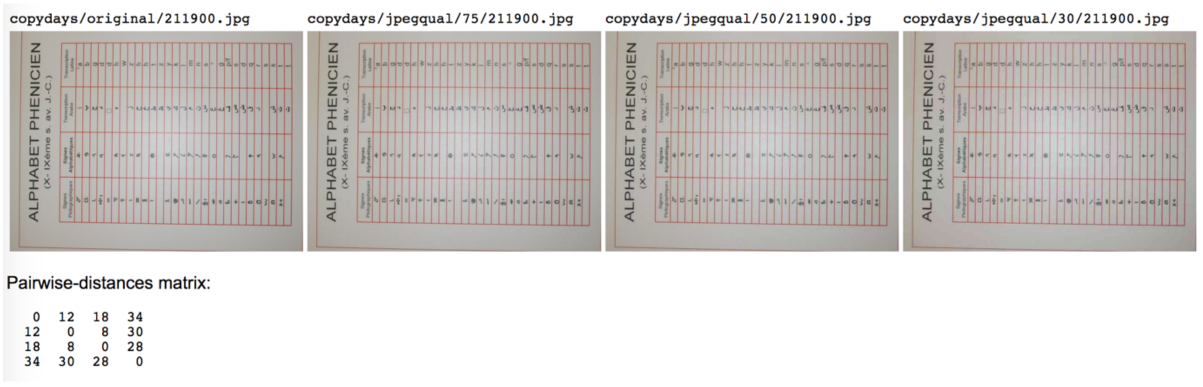
Since we're using matching algorithms with tunable matching thresholds, we know we're trading precision for recall. In the peaked blue plots above we saw that, for CopyDays originals + JPEG-quality mods, there was a notch in the pairwise-distance distributions between matches and non-matches, which gradually filled in as lower and lower JPEG-quality mods were thrown into the mix. In this section, we choose a distance threshold located toward the left (strict) side of those notches: **32**. We expect that with lower JPEG-quality images included, we'll find mismatches – this is an intentional part of the test. As is shown below, this causes a single cluster split at JPEG quality 30.

An automated clustering test is as follows:

- Take all 157 CopyDays originals plus JPEG-quality modifications, including lower and lower qualities, resulting in *n* copies of each image. (For example, with originals and JPEG qualities 75, 50, and 30, we have *n*=4).
- Hash all 157*n* images and cluster them, using the matching-distance thresholds as chosen above.
- Results: in almost all cases, PDQ successfully forms full clusters of *n* images each. The exceptions are as shown below, and these are always at the lowest-end image qualities.
- The fact that clusters are of size at most *n* demonstrates **precision** of both algorithms; the fact that clusters are of full size *n* demonstrates **recall** of both algorithms, except in the lowest-quality end-cases of certain less featureful photos which are beyond our interest. (It is important to establish that not only are clusters of the right size, but that the correct images are clustered together within them. The dataset here is small enough that it is easy to verify this by simply looking at all the clusters.)

| **Qualities** | **n** | **PDQ with threshold 32** |
|---|---|---|
| Original, 75 | 2 | 157 size-2 clusters |
| Original, 75, 50 | 3 | 157 size-3 clusters |
| Original, 75, 50, 30 | 4 | 156 size-4 clusters; 1 split |
| Original, 75, 50, 30, 20 | 5 | 155 size-5 clusters; 2 splits |
| Original, 75, 50, 30, 20, 15 | 6 | 152 size-6 clusters; 5 splits |

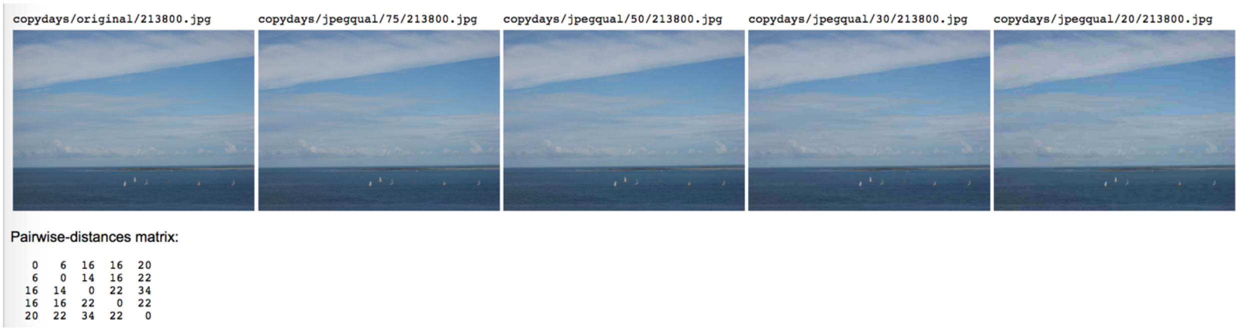
This is the first cluster-split:


```
copydays/original/211900.jpg    copydays/jpegqual/75/211900.jpg    copydays/jpegqual/50/211900.jpg    copydays/jpegqual/30/211900.jpg
```

Pairwise-distances matrix:

```
  0  12  18  34
 12   0   8  30
 18   8   0  28
 34  30  28   0
```

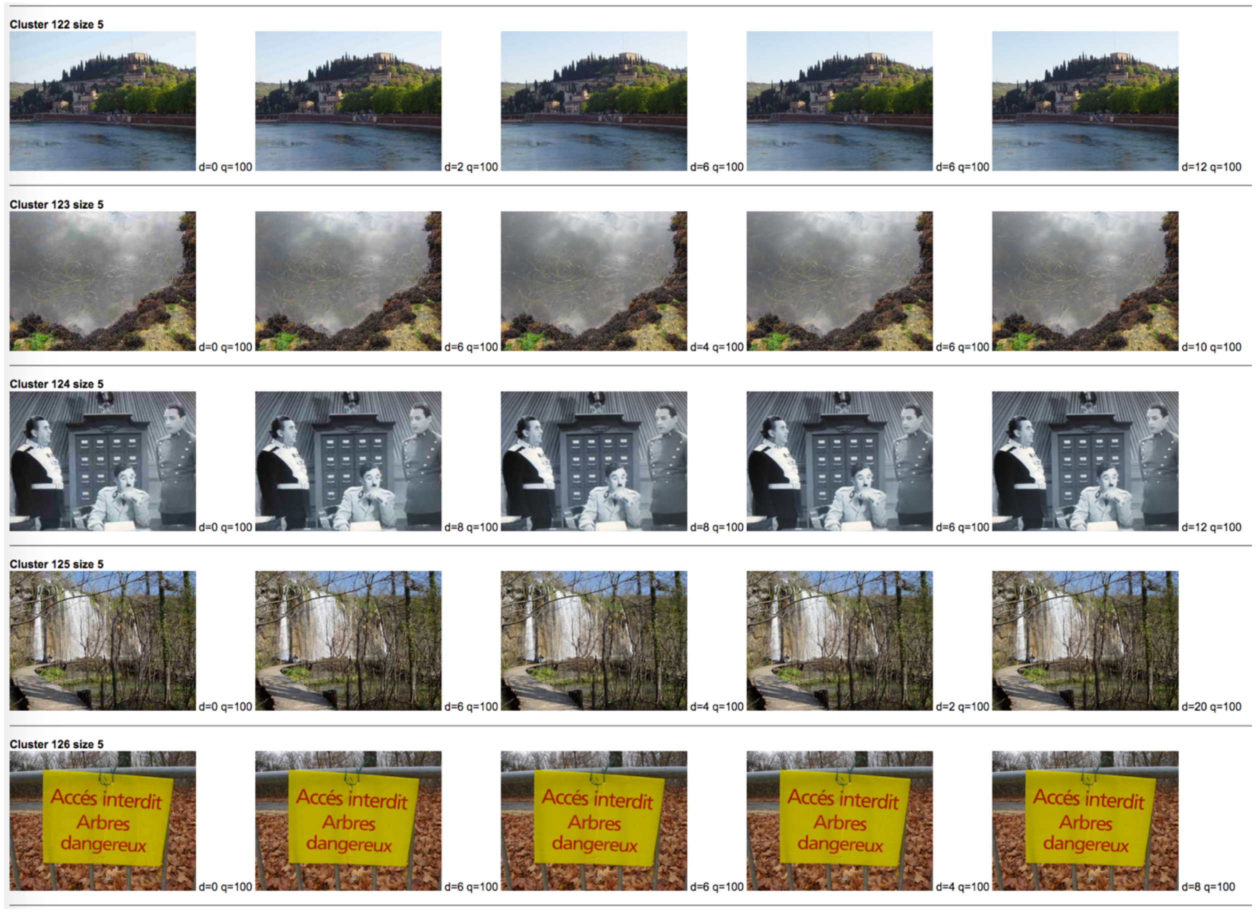
and this is the second:


```
copydays/original/213800.jpg    copydays/jpegqual/75/213800.jpg    copydays/jpegqual/50/213800.jpg    copydays/jpegqual/30/213800.jpg    copydays/jpegqual/20/213800.jpg
```

Pairwise-distances matrix:

```
  0   6  16  16  20
  6   0  14  16  22
 16  14   0  22  34
 16  16  22   0  22
 20  22  34  22   0
```

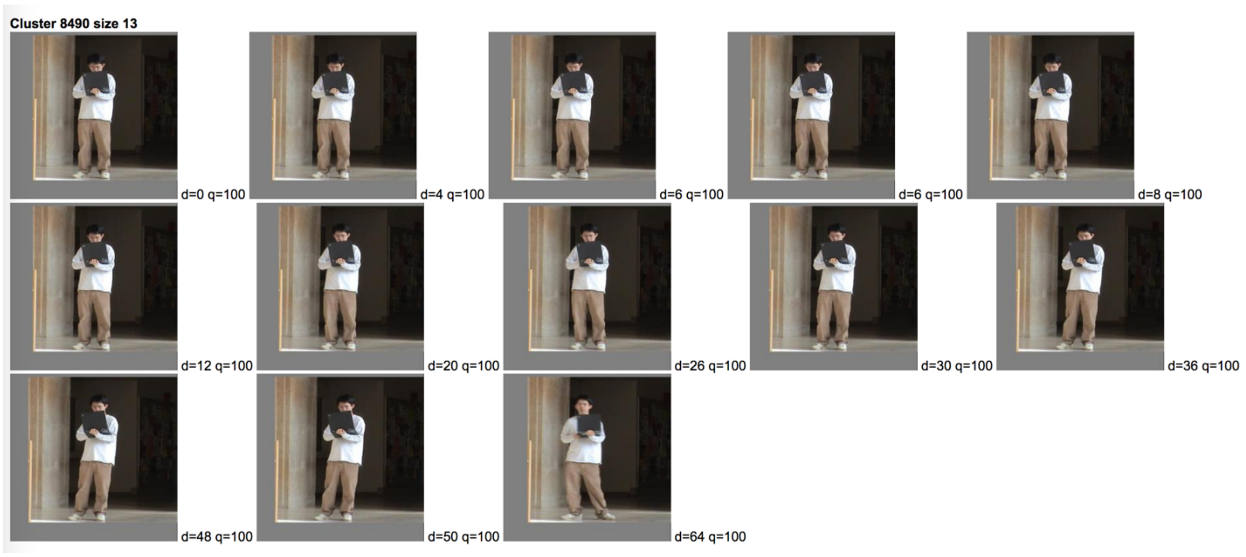
Lastly, most images are clustered together. There are many images, but not too many to verify visually. Here are some example clusters (again, this is using a **threshold of 32**):



## REFINEMENT OF MATCHING-DISTANCE THRESHOLD

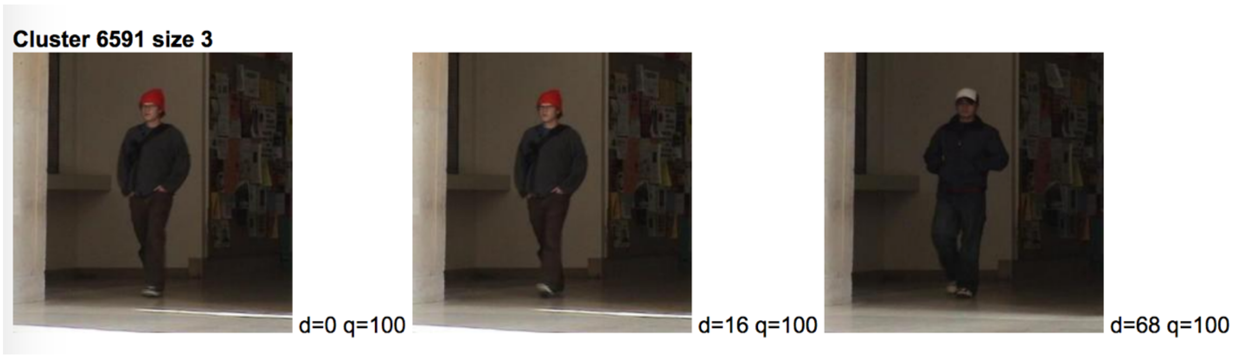
Test data up until now have consisted of images which are clearly perceptually distinct, then modified by lowering their image quality farther and farther to (and beyond) the point at which they are useful. Our next step is to try to cluster images which are similar to various degrees. Here the input dataset has no sharp pre-existing clustering to draw on, so the work will get more interesting.

Turning now to the LabelMe dataset (http://labelme.csail.mit.edu), we first hash all photos, then cluster them. For a first experiment, we use PDQ distance of 70. Sorting the output descending by cluster size, at the top we see a cluster of relatively featureless images such as those shown in the quality-metric section below, then nearby (**threshold is 70**):



Given the loose threshold of 70, we see the same individual in slightly different postures. Tightening the threshold splits this cluster into subclusters.

We also see following triple at threshold 70. This is interesting since much of the image (the background) is similar, and both individuals are wearing dark jackets. The parts that are different occupy relatively little of the images, and will not occupy very many pixels of the downsampled images.

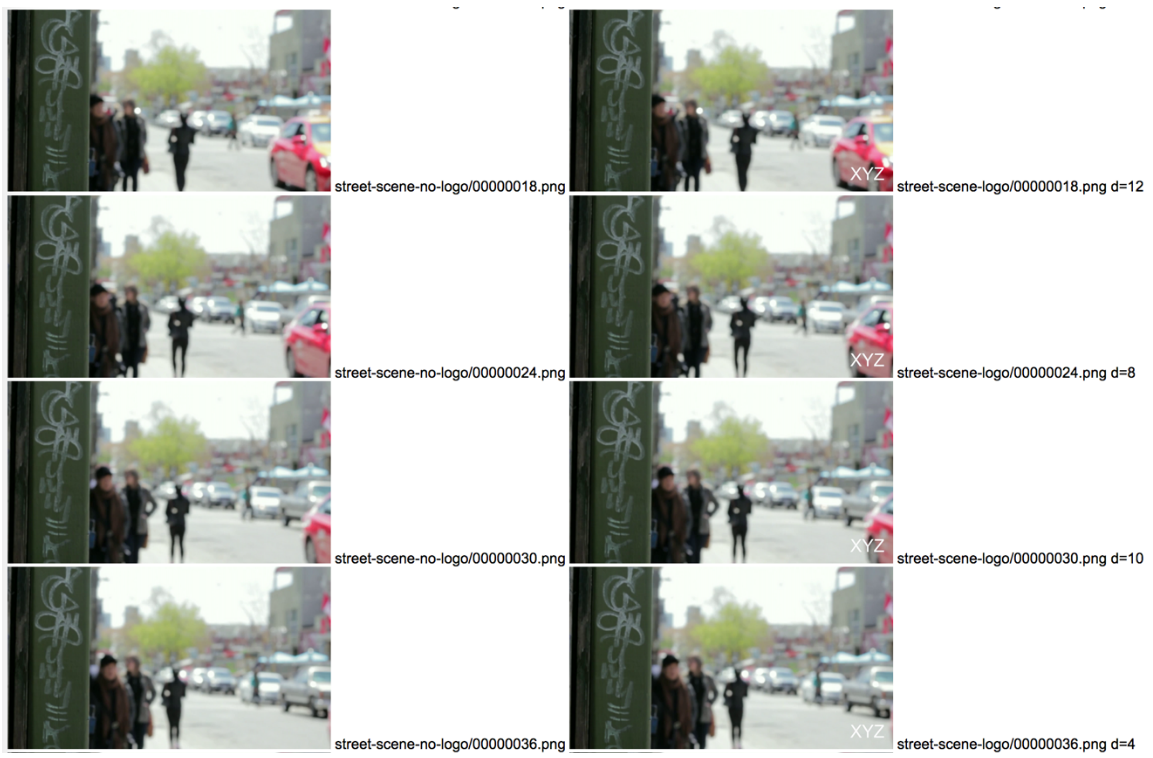


The desired pair (the left and center images) differ by 16; the left and right differ by 68 – here again, using a narrower threshold splits this cluster

Ultimately, evaluation of a perceptual hash must be visual. There are sufficiently few images in our test datasets that it is possible to cluster them all and look at all the clusters. Details of the tooling used to generate data for this document are discussed in the Reference Implementation section.

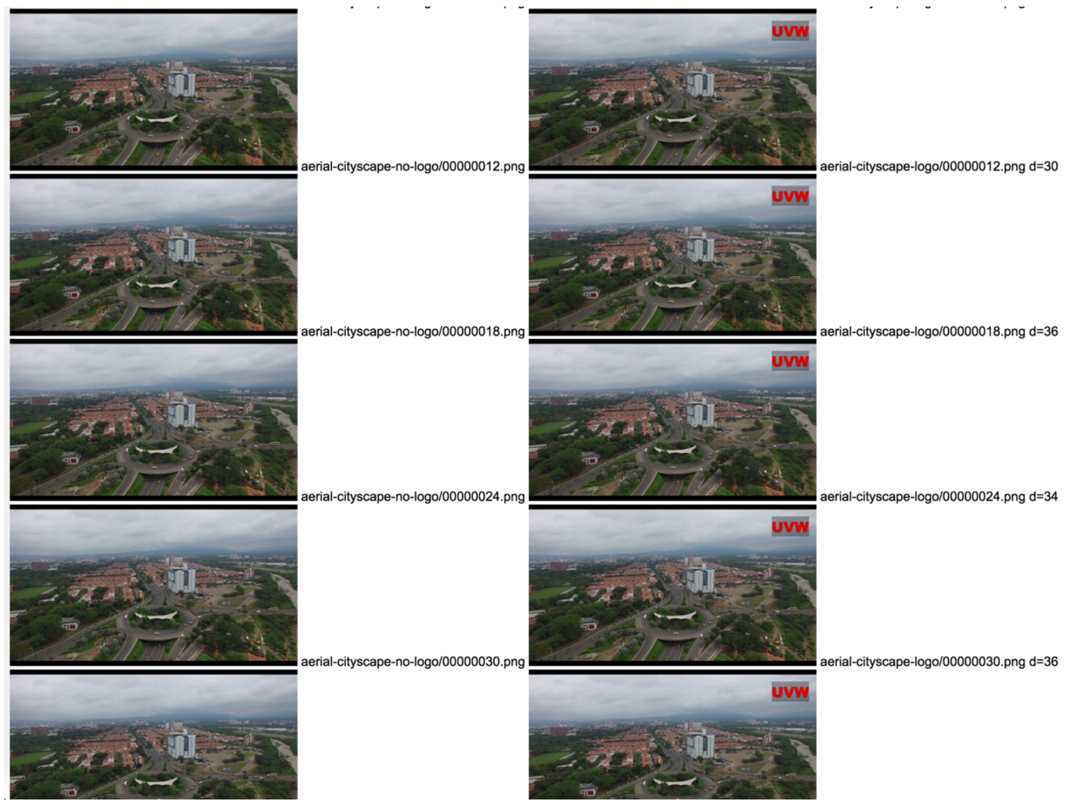
## WATERMARK TOLERANCE

This is a frequently encountered in abuse-detection practice at Facebook. Here's an excerpt from a lightly watermarked video, frame-pair by frame-pair:


XYZ
street-scene-no-logo/00000018.png
street-scene-logo/00000018.png d=12
street-scene-no-logo/00000024.png
street-scene-logo/00000024.png d=8
street-scene-no-logo/00000030.png
street-scene-logo/00000030.png d=10
street-scene-no-logo/00000036.png
street-scene-logo/00000036.png d=4

Here we see the PDQ hamming distances are quite tight, relative to our expectations set by the bell-graphs above.

Next, we try a more obtrusive watermark:


UVW
aerial-cityscape-no-logo/00000012.png
aerial-cityscape-logo/00000012.png d=30
aerial-cityscape-no-logo/00000018.png
aerial-cityscape-logo/00000018.png d=36
aerial-cityscape-no-logo/00000024.png
aerial-cityscape-logo/00000024.png d=34
aerial-cityscape-no-logo/00000030.png
aerial-cityscape-logo/00000030.png d=36

The distances here are bigger, but still within the threshold range we've developed. (If we'd used even more invasive watermarks we'd expect mismatches.)

This is an important use-case: in abuse-detection work at Facebook, we find that sometimes people will place a watermark/logo in the corner of a photo/video where there was none – or, overwrite one on top of an existing one.

The intuition is as follows: for PDQ, which is based spatial frequencies, the relatively small-scale modifications introduce frequency components which are not retained in the lowpass-filtered DCT output.

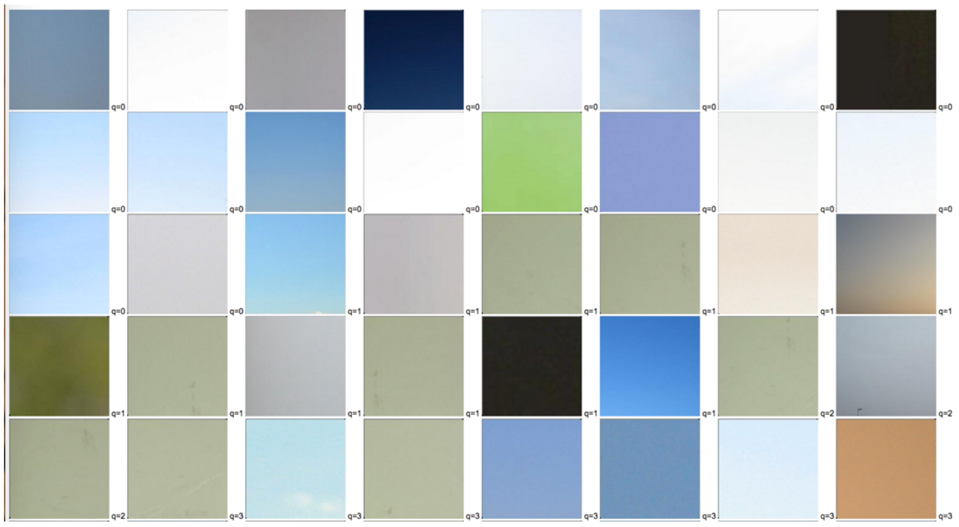
## DEFINITION AND EVALUATION OF QUALITY METRIC

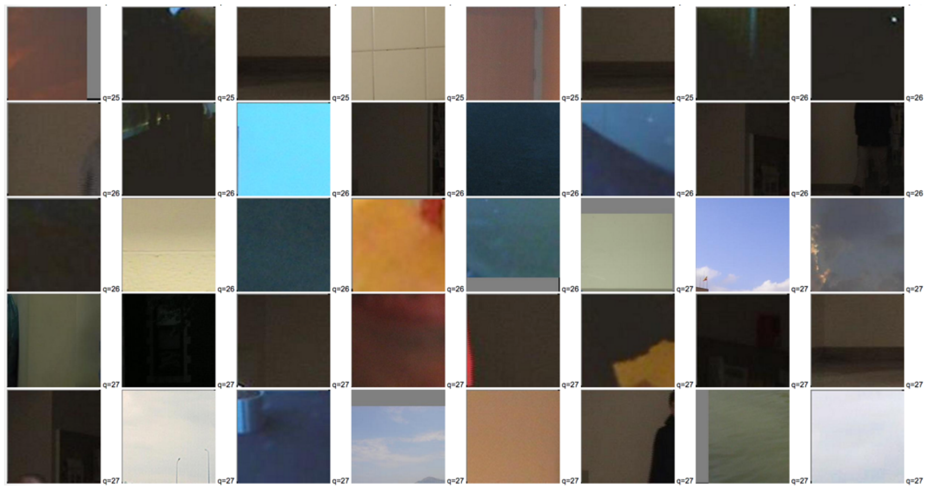
As noted in the algorithm section, PDQ features a quality metric. It measures **gradients**: it is the sum of absolute values of quantized horizontal and vertical nearest-neighbor differences in the 64x64 downsampled luminance image. The raw sum is then divided by a heuristic constant, then capped at 100. In this section, we discuss the determination of that constant.

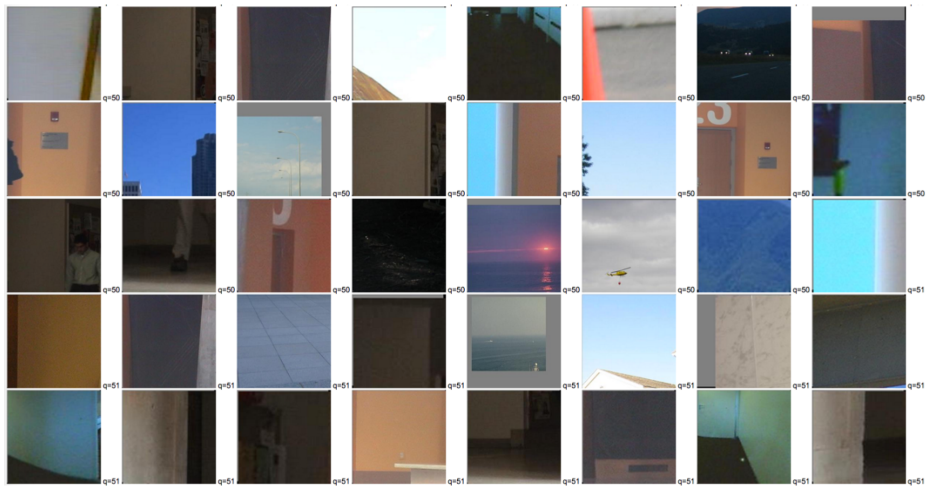
Steps are:

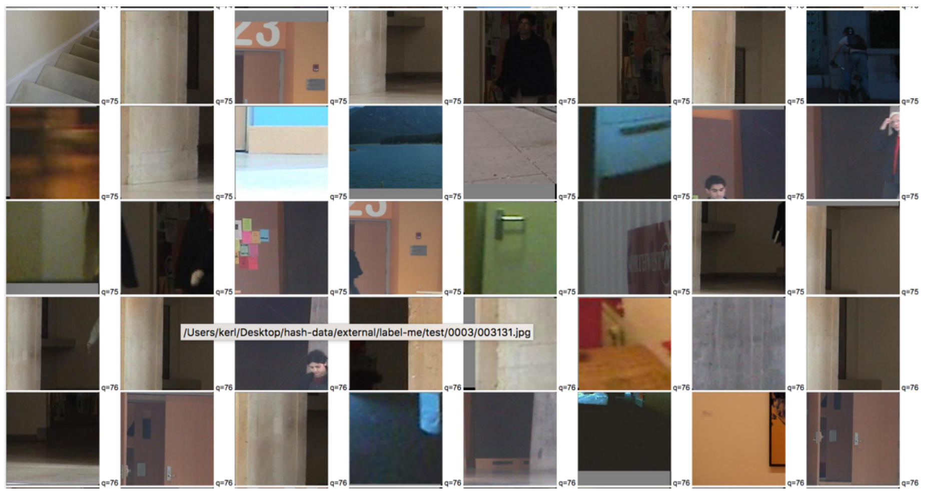
- Pick an arbitrary scaling constant, and do not cap at 100.
- Hash all the photos in the LabelMe dataset, sorting ascending by quality.
- Verify **consistency**, namely, that the quality metric indeed sorts images least featureful to most featureful. (During the development of PDQ, a frequency-domain quality metric was evaluated, which applied similar logic not to the 64x64 luminance downsample but to the 16x16 DCT-output matrix. That frequency-domain quality metric did roughly sort from less to more featureful, but did not correlate well enough with perceptual featurefulness.)
- Select the least not-entirely-feature images and adjust the scaling factor so that these will have score 99.
- Restore the cap at 100.

Some output examples are as follows:

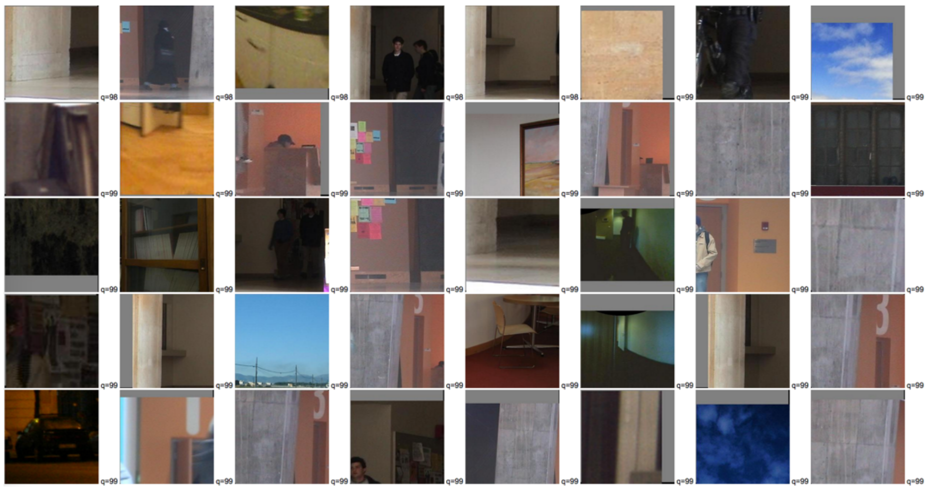



q=25 q=25 q=25 q=25 q=25 q=25 q=26 q=26
q=26 q=26 q=26 q=26 q=26 q=26 q=26 q=26
q=26 q=26 q=26 q=26 q=26 q=26 q=27 q=27
q=27 q=27 q=27 q=27 q=27 q=27 q=27 q=27
q=27 q=27 q=27 q=27 q=27 q=27 q=27 q=27


q=50 q=50 q=50 q=50 q=50 q=50 q=50 q=50
q=50 q=50 q=50 q=50 q=50 q=50 q=50 q=50
q=50 q=50 q=50 q=50 q=50 q=50 q=50 q=51
q=51 q=51 q=51 q=51 q=51 q=51 q=51 q=51
q=51 q=51 q=51 q=51 q=51 q=51 q=51 q=51


q=75 q=75 q=75 q=75 q=75 q=75 q=75 q=75
q=75 q=75 q=75 q=75 q=75 q=75 q=75 q=75
q=75 q=75 q=75 q=75 q=75 q=75 q=75 q=75
q=76 q=76 q=76 q=76 q=76 q=76 q=76 q=76
q=76 q=76 q=76 q=76 q=76 q=76 q=76 q=76

/Users/kerl/Desktop/hash-data/external/label-me/test/0003/003131.jpg


q=98
q=98
q=98
q=98
q=98
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99
q=99

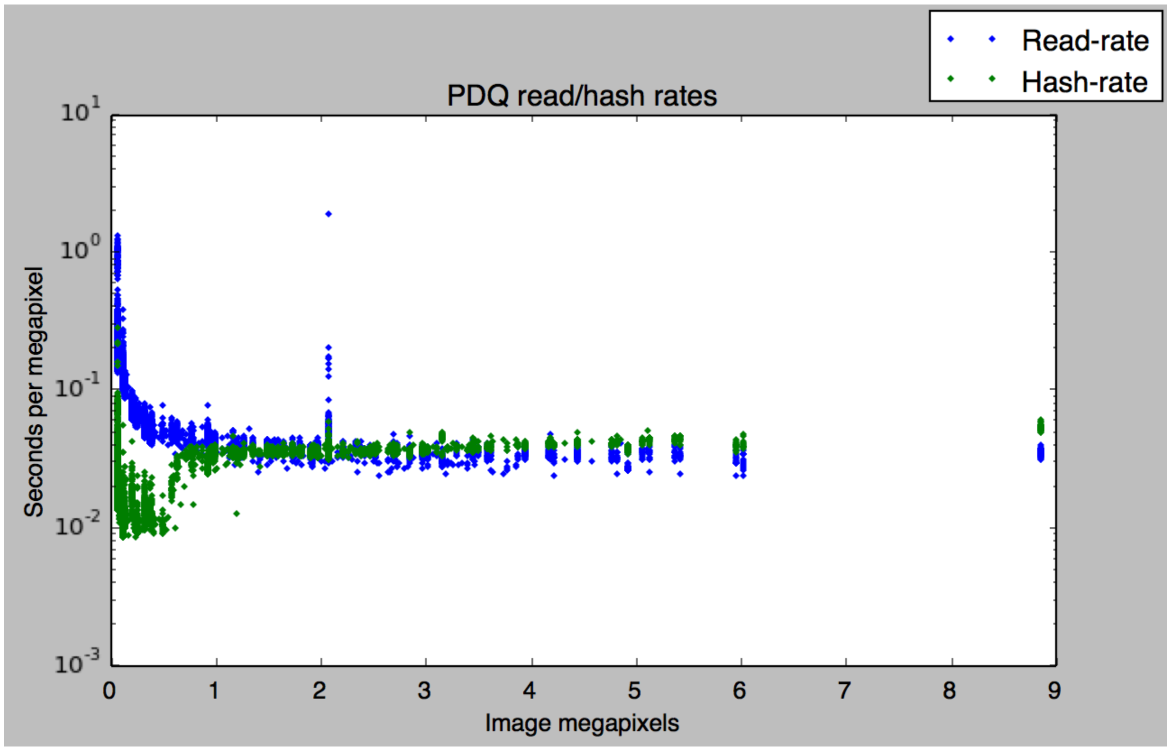
## PERFORMANCE

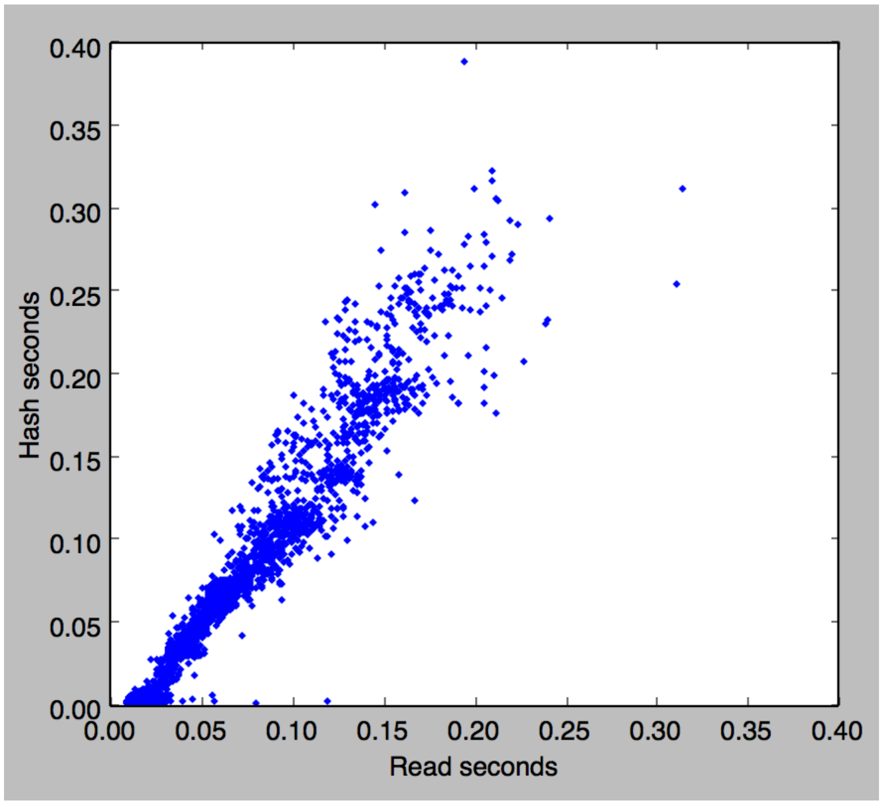
### TMK+PDQF HASHING PERFORMANCE

(To be written.)

### PDQ HASHING PERFORMANCE

Over all test data, we have the following rates for PDQ:





Discussion:

- Disk-read rate is roughly constant in file size – it takes about twice as long to read a file twice as big. However, this general throughput rule is accompanied by file-open latency: it takes more than zero time to open a zero-length file. Hence the blue peak at the left in the first plot: here is where latency dominates over throughput. Do note that this is a general property of file I/O, having nothing to do with PDQ hash – nonetheless we mention it for completeness.
- As discussed above, PDQ has two compute-intensive parts: the downsample-to-64x64 phase which is dependent on input file size, and the DCT phase which always takes 64x64 input regardless of input-file size. For files above about a megapixel, the hashing times are quite close to file-read times. For smaller files, this time component which is independent of file size yields a green peak at the left. Note these smaller file sizes are actually *normal* sizes for lower-resolution video frames (e.g. 480x720).

## INDEXING

### TMK+PDQF QUERIES

#### TMK+PDQF QUERIES VIA LINEAR SEARCH

To be written. Summary is: If you have a small collection, don't underestimate the efficiency of linear search. It's easy to code up, hard to get wrong, and has low data-structure overhead.

#### TMK+PDQF QUERIES USING FAISS

To be written. Summary is: we have performed an initial integration, as presented within the GitHub repo, which functions well but needs additional productization.

### PDQ QUERIES

We turn next to the question of lookups: namely, if you have a collection of PDQ hashes of images, and another image arrives on your doorstep, how do you efficiently search?

#### HOW NOT TO DO IT

K-dimensional trees are a standard tool for looking up multi-dimensional vectors. However, they don't work in hamming space. K-d trees rely on splitting a set of k-dimensional vectors across hyperplanes, so that if you're looking for near-neighbors to a particular input hash, at each node of the tree you can go left or right or both, but usually not both. In hamming space there is only one bit in each dimension: there's not information to make a left-or-right decision, so you must go left *and* right. This turns into a linear search – the number of hash compares is the same as the total number of hashes. Trying to break the 256-bit hashes up into 16-bit words and doing a k-d tree on those also doesn't work. The 16 bits are actually independent, and grouping them together doesn't yield a branching decision.

#### WAYS TO DO IT

Indexing bit-vectors by hamming distance is a well-known problem; there are many techniques going back decades.

- If you have your own system for indexing bit-vectors that you're already comfortable with, by all means you should use that. And if you want to start using a different method, again, by all means do that. For purposes of cross-industry hash-sharing, what we must agree on is the hashing algorithm – how images map to 1s and 0s. The indexing techniques, by contrast, are implementation details within each company.
- If you have a small collection, don't underestimate the efficiency of linear search. It's easy to code up, hard to get wrong, and has low data-structure overhead.
- Nonetheless the PDQ reference implementation should offer a default implementation, and that implementation is **mutually-indexed hashing**.

#### MUTUALLY-INDEXED HASHING

PDQ hashes are 256 bits. These could be stored as four 64-bit words; within the reference implementation they're stored as 16 16-bit words, specifically for MIH indexing on 16-bit words.

Suppose you want to use a hamming-distance threshold of 32. Suppose you have a candidate hash ('needle') and you want to look it up in a large hash bank ('haystack'). On average you might imagine that matches within the haystack would differ within about 2 bits per 16-bit word – if the 32 bits of hamming distance were distributed evenly throughout all the words. While that makes sense for the average case, in a particular case a lot of that distance might be concentrated within a word or two.

The key concept of MIH is the following: if the differing bits are clustered into a few words, the *rest* must have *smaller* slotwise hamming distances. Concretely, given a distance threshold of 32, if a needle hash *N* and a haystack hash *H* have hamming distance *d*(*N*,*H*) <= 32, then *in at least one slot* they must differ by 2 or less (2 being the floor of the quotient of the distance threshold 32 by the slot-count 16). This is true because otherwise – if they differed by more than 2 in each slot – then their overall distance would exceed 32.

Using MIH we **build an index** as follows:

- For each haystack hash *H*, append it to an array which holds all the haystack hashes.
- For each 16-bit word (or 'slot') of *H*, get the slot value (maybe 0x7ef4 at slot 12, for example).
- At that slot index (0, 1, 2, …, up through 15), for the slot value, insert *H*'s array index into a hash-set of array indices.
- That is, we have the following:
    - An array indexed by slot index, 0..15.
    - Each element of that array is a hashmap keyed by 16-bit slot value (e.g. 0x7ef4).
    - The value of that hashmap is a hashset of the array indices of all haystack hashes having that slot value at that slot index.

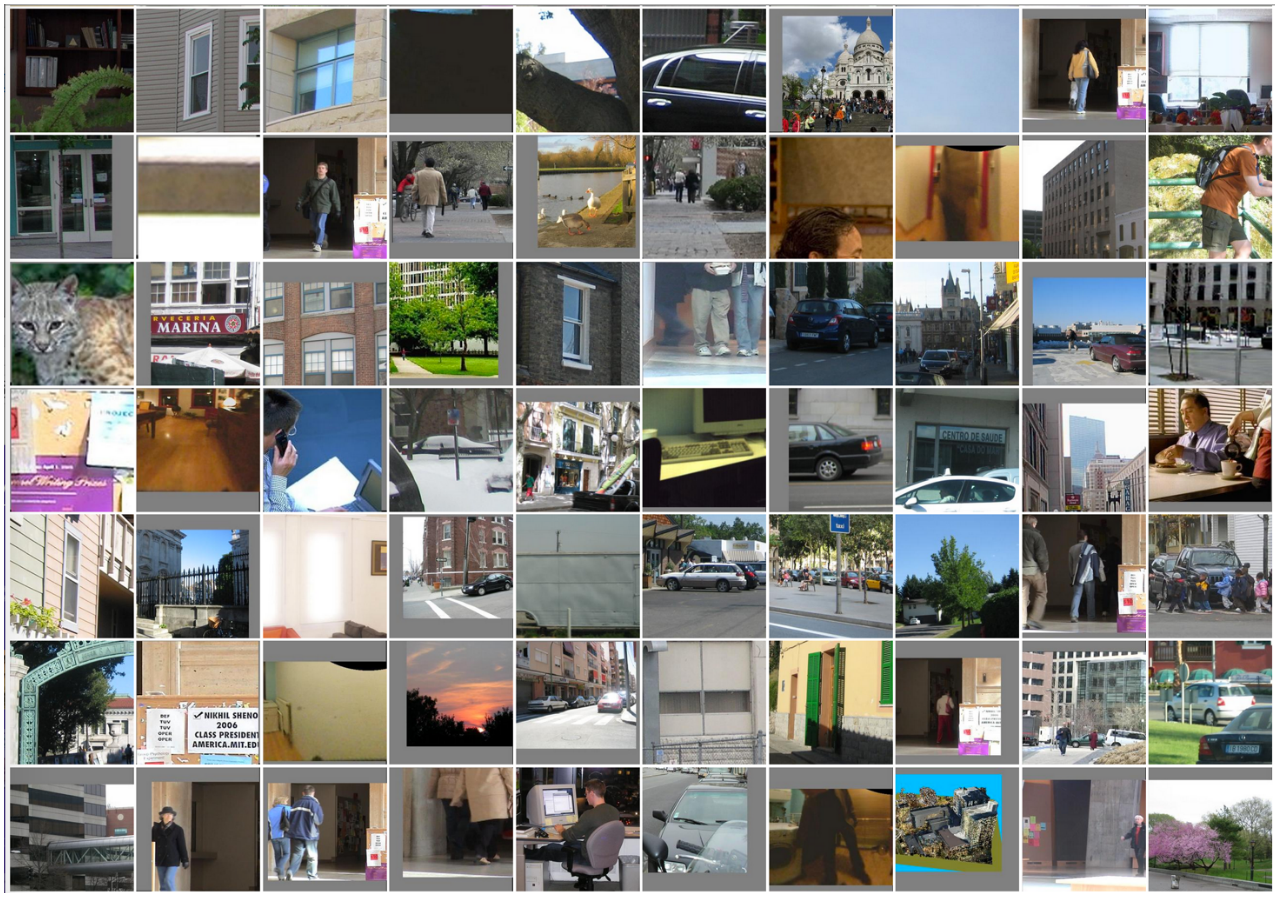
At **query** time:

- Given a needle hash *N*, loop over all the 16-bit words (or 'slots') of *N*.
- Knowing all the possible 16-bit values which differ by 0, 1, or 2 from a given slot value (this is [lazily] precomputed and cached), look up in the build-step's hashmap all the array indices of haystack hashes which are within slotwise distance 2 at the given slot and insert these into a *candidate set*.
- For example, if *N* has slot value 0x7eb4 at slot 12, then when we look at slot 12 for array indices of hashes whose 12th slot is within distance 2 of 0x7eb4, we'll find the example hash from the index-build example, which had a 12th slot value of 0x7ef4 which is one bit away from 0x7eb4.
- Given the candidate set – all the haystack hashes which differ by 2 or less in *any* slot from the given needle hash – then compute the full hashwise hamming distances and return only those which match within the hashwise distance threshold of 32.

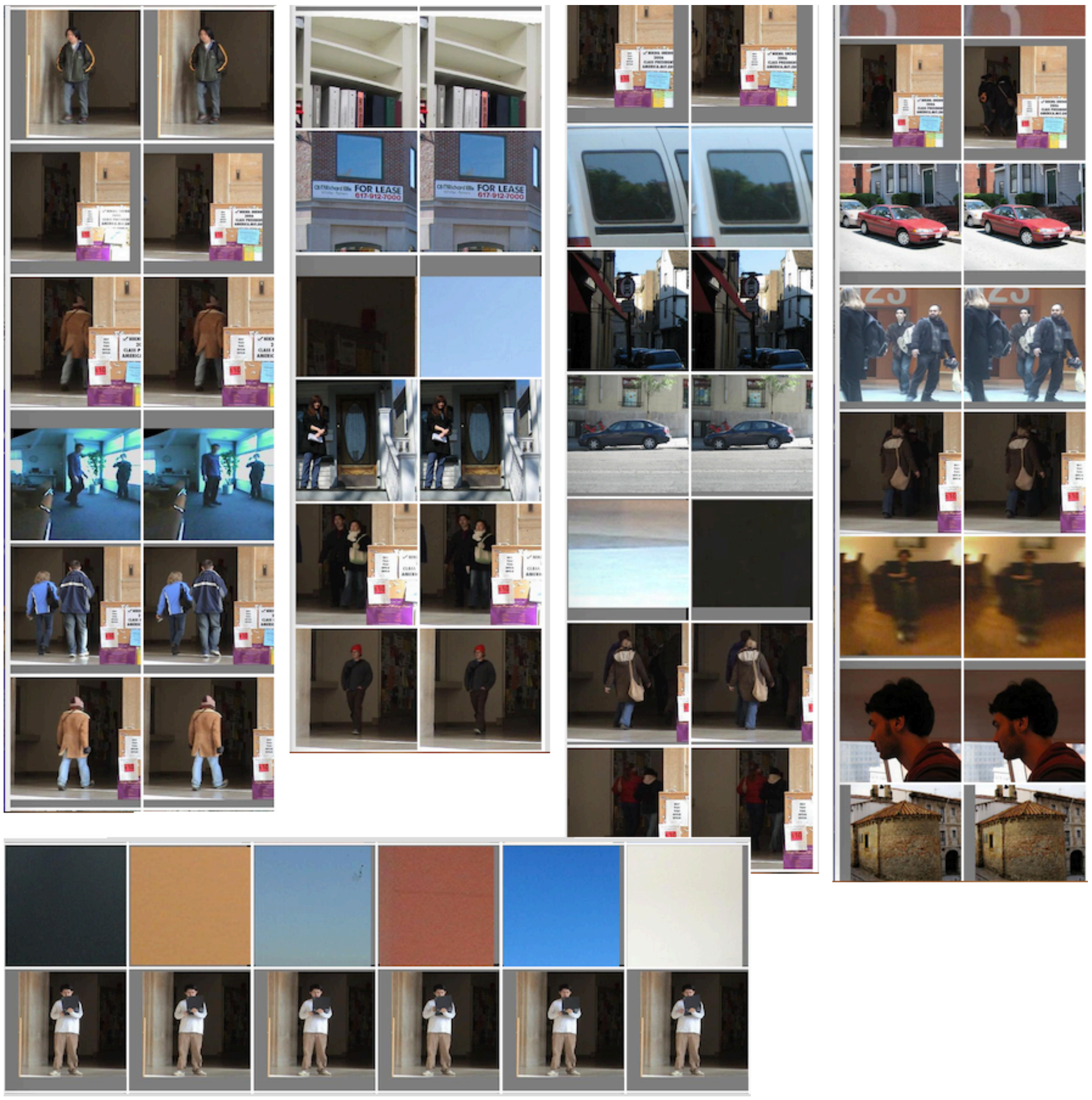
The MIH algorithm is not an approximate or heuristic indexer; it finds all the same matches that a linear search would, but in sublinear time. For full details, please see (Norouzi, 2014).

## PDQ CLUSTERING RESULTS

Loading random-ordered images from the LabelMe dataset discussed above, we have 50,000 images such as the following excerpt:



Building an MIH data structure and forming similarity clusters, we get results such as the following – shown here are a few dozen pairs (i.e. clusters of two), and two clusters of six:



Steps taken to produce the PDQ-related figures in this document are documented within the evaluation-notes.txt file within the repository.

## BIBLIOGRAPHY

Douze, M. e. (2009, 07 10). *Evaluation of GIST descriptors for web-scale image search*. Retrieved from https://pdfs.semanticscholar.org/2270/c94d3f9d9451b3d337aa5ba2d5681cb98497.pdf: https://pdfs.semanticscholar.org/2270/c94d3f9d9451b3d337aa5ba2d5681cb98497.pdf

INRIA. (2006, 12 01). *http://lear.inrialpes.fr/~jegou/data.php#copydays*. Retrieved from http://lear.inrialpes.fr/~jegou/data.php#copydays: http://lear.inrialpes.fr/~jegou/data.php#copydays

Jarosz, W. (2001, 1 1). *Fast Image Convolutions*. Retrieved 8 1, 2017, from ACM SIGGRAPH @ UIUC: elynxsdk.free.fr/ext-docs/Blur/Fast_box_blur.pdf

Laboratory, MIT Computer Science and Artificial Intelligence. (2005, 01 01). *http://labelme.csail.mit.edu/Release3.0/browserTools/php/dataset.php*. Retrieved from http://labelme.csail.mit.edu/Release3.0/browserTools/php/dataset.php: http://labelme.csail.mit.edu/Release3.0/browserTools/php/dataset.php

Lee, B. G. (1984, 12 1). *A New Algorithm to Compute the Discrete Cosine Transform*. Retrieved 08 01, 2017, from tsp7.snu.ac.kr/int_jour/IJ_2.pdf: tsp7.snu.ac.kr/int_jour/IJ_2.pdf

Mogensen, T. (1998, 07 01). *Median algorithm*. Retrieved 08 01, 2017, from http://ndevilla.free.fr/median/median/index.html: http://ndevilla.free.fr/median/median/index.html

Norouzi, M. e. (2014, 4 25). *https://www.cs.toronto.edu/~norouzi/research/papers/multi_index_hashing.pdf*. Retrieved 10 8, 2017, from https://www.cs.toronto.edu/~norouzi/research/papers/multi_index_hashing.pdf: https://www.cs.toronto.edu/~norouzi/research/papers/multi_index_hashing.pdf

Sébastien Poullot, Shunsuke Tsukatani, Anh Phuong Nguyen, Hervé Jégou, Shin'Ichi Satoh. Temporal Matching Kernel with Explicit Feature Maps. ACM Multimedia 2018, Oct 2015, Brisbane, Australia.pp.1-10, 10.1145/2733373.2806228. hal-01842277

Wikipedia. (2017, 08 22). *https://en.wikipedia.org/wiki/Discrete_cosine_transform*. Retrieved from https://en.wikipedia.org/wiki/Discrete_cosine_transform: https://en.wikipedia.org/wiki/Discrete_cosine_transform

Zauner, C. (2010, 7 1). *Implementation and Benchmarking of Perceptual Image Hash Functions*. Retrieved 8 1, 2017, from https://www.phash.org/docs/pubs/thesis_zauner.pdf: https://www.phash.org/docs/pubs/thesis_zauner.pdf

## GLOSSARY

**decimation**: Selection of a smaller image from a larger one by simply selecting a subset – for example, retaining every 10th pixel in the horizontal and vertical directions. While easy to implement and understand, this technique when used alone leads to *aliasing*, namely, effects appear in the downsample which are not related to what we would perceptually expect. For example, imagine a picture of a picket for which the decimation misses all the posts (or, all the gaps). For this reason, image downsampling can be accomplished by a combination of blurring and decimation. Once an image has been blurred, individual pixels selected in the decimation will have contributions from many neighboring pixels, and thus will be more representative of the visual contents of the block they have been chosen to represent.

**discrete cosine transform (DCT)**: A mathematical transformation which, given an image, produces an itemization of which spatial frequencies contribute to the image and to what extent.

**downsampling**: See the entry on decimation.

**false negative**: two things *should* match, but our algorithm says they *do not* match.

**false positive**: two things *should not* match, but our algorithm says they *do* match.

**filter**: General terms for a simple algorithm which takes an image as input and produces a modified image as output. One example is a *box filter*, which forms every pixel of the output by the average (perhaps weighted, perhaps non-weighted) of an *m* x *m* square of pixels in the input. This leads to blurring. A second example is an *edge-detection filter* (useful in image processing, though not used in this document) which computes each output pixel by 4x a center input pixel minus its above/below/left/right neighbors. In a region of nearly solid input color, this will produce nearly black (regardless of the input color); across brightness gradients (edges), this will produce more brightness. This produces white-on-black outline-sketch effect. Third and subsequent examples include all sorts of filters you'll find in photo-manipulation software packages (ripple, distort, posterize, etc.)

**greyscale**: An unweighted average of red, green, and blue pixels. Roughly similar to luminance but mathematically distinct.

**hamming distance**: Given two arrays of 0s and 1s (AKA *binary hashes*), the hamming distance between them is the count of bit positions in which they differ. For example, 0010 and 0111 are length-four bit arrays with hamming distance two: the 0 in the first slot and the 1 in the third are the same; the second and fourth slots differ. This is same as the *hamming norm* of their *XOR*.

**hamming norm:** Given an array of 0s and 1s (AKA a *binary hash*), the hamming norm is the count of 1-bits. For example, 00100111 has hamming norm four since it has four 1s in it. The *hamming distance* between two arrays is the hamming norm of their *XOR*.

l**uminance**: A weighted average of red, green, and blue pixels. The weighting coefficients are industry-standard and produce an output roughly similar to greyscale but in better correspondence with human perception of brightness.

**mutually-indexed hashing (MIH):** An indexing algorithm for bit-vectors by hamming distance. It relies on splitting hashes into subcomponents, then tracking which hashes within the index are within a componentwise distance of one another, for all subcomponents.

**precision:** the ratio of true positives to true positives plus false positives: TP/(TP+FP). It measures how often our detection is wrong.

**recall**: the ratio of true positives to false negatives: TP/(TP+FN). It measures how often we miss detecting what we want to detect.

**true negative**: two things *should not* match, and our algorithm says they *do not* match.

**true positive**: two things *should* match, and our algorithm says they *do* match.

**XOR** (exclusive OR): Given two arrays of 0s and 1s, the XOR of them is an array of the same length with a 1 at slots where they differ and a 0 at slots where they are the same. For example, 0011 and 0110 have XOR 0101 since the first and third slots are the same while the second and fourth differ. The *hamming distance* between two arrays is the *hamming norm* of their XOR.

## INDEX