Content Based Image Retrieval in iOS

Face Recognition using Local Binary Pattern Descriptors

Joseph Carson <jcarson8@fau.edu> - CAP 6411 - Foundations of Vision - Florida Atlantic University
https://github.com/JoeyCarson/cbir_database_ios

Abstract

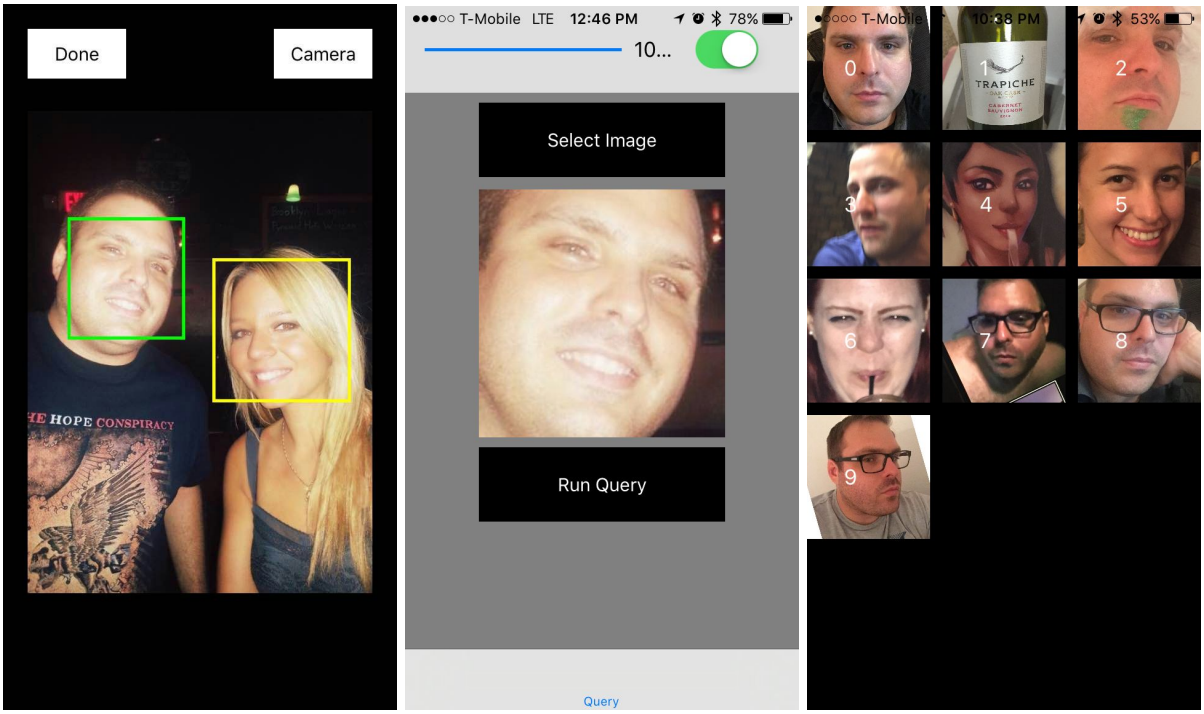
Content-Based Image Retrieval (CBIR) systems are commonly designed for execution on large scale parallel processing hardware that provide supercharged image processing and retrieval services for smart phones, embedded, and IoT client devices. Although today's smartphones are equipped with parallel processing hardware that supports high performance image processing tasks, much of it is traditionally utilized for photo editing and real time graphics processing tasks. As Moore's law perpetuates, CBIR systems will undoubtedly be making their way into even consumer smartphones for niche purposes. The intention of the following project is to build a rudimentary CBIR system designed for providing on-device image indexing and retrieval services for iOS applications, particularly in the context of face recognition, while allowing simple extension of the fundamental components and providing open access to utility functions used internally. The components leverage Apple's Core Image API that facilitates parallelizing image processing tasks provided by the system and custom implementations as well. The fundamental principles of the face recognition model are based in part on the Restricted Naive Bayes Nearest Neighbor algorithm proposed by Maturana et. al. that leverages feature descriptors composed of Local Binary Pattern (LBP) histograms and includes an optimization strategy proposed by Ahonen. The dataset is primarily unconstrained, composed of real life photos and *selfies* with natural variations in pose, lighting, and obstruction which may differ from common benchmarking face image training sets.

Content Based Image Retrieval methods are computationally expensive tasks that typically require a degree of parallelization to perform with realistic expectations for accuracy, running time, and efficient resource utilization. Traditionally, large scale server machines equipped with parallelizable hardware like high performance GPU modules are utilized to perform the complex and highly redundant image processing tasks necessary for visual feature extraction and reasoning. While this solves the use case of providing CBIR services to a vast number of clients, smartphones and embedded devices are equipped with powerful graphics processing hardware, making it possible to implement smaller scale CBIR tasks locally on the device with modest expectations of accuracy and performance. What follows is a rudimentary implementation of a CBIR database library, implemented in Objective-C, and designed for supporting various types of image indexing and query services in iOS applications. Indexing and searching images according to face is the only service implemented, but the framework is straightforward and easily extendable.

The obvious question on most readers minds is why CBIR services would be implemented on the device side in the first place when remote large scale systems are already so capable of performing them. Facebook already does such a great job of pre-identifying associated friend's faces in users uploaded images, and also attempts to identify and tag the owning user's face locally on the device before uploading. Image processing systems in the cloud are expensive to purchase and maintain, especially at the userbase scale of facebook and other social networks. While they're not going anywhere any time soon, utilizing this high performance hardware on user's devices can ultimately save on bandwidth as well as cloud based storage and compute expenses. The primary reason for embedded device CBIR is simply because we can and eventually it may be prove to be useful.

The reference implementation is broken into two distinct build targets, one being the database engine library and the other an application that consumes the library in order to test it. The library project (CBIRDatabase) is designed to be an abstract image database library for iOS,

intended to give applications the choice of which images to index and how to query them as opposed to being a dedicated iOS photo album search engine. This design allows for any kind of image to be indexed into the database whether it be device photos, user generated or downloaded images, etc. The CBIRDatabase persists image descriptor data using CouchbaseLite (a NoSQL object store designed for smartphones) and for simplicity only implements an indexer and query for faces. Additional indexer and query types can easily be implemented and registered inside or outside of the library, even custom implementations provided by the consuming application. The test application, simply named CBIRD, implements a component to index all images in the local iOS photo album in an efficient and thread-safe manner as well as a few user interface components to invoke the camera and select a face from an image captured by the camera to use as the query input, and display the search results. The combined result is effectively a CBIR photo album face search tool.

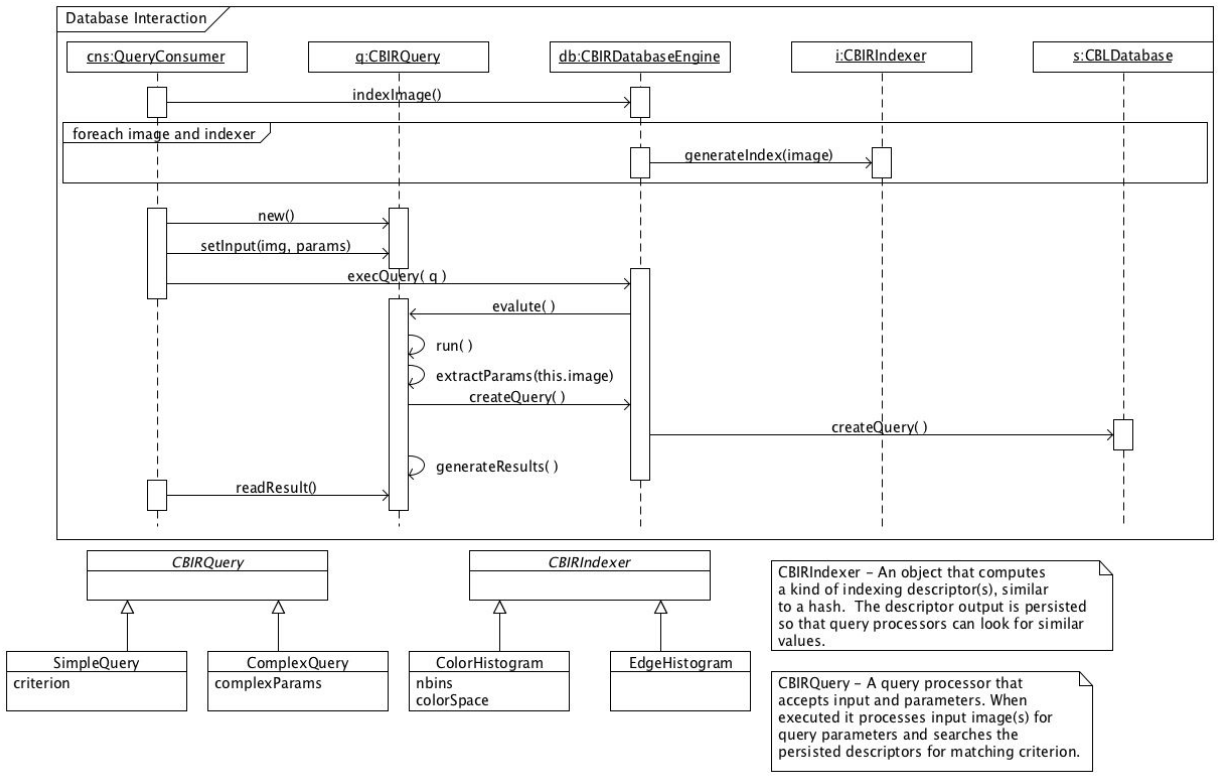


Application Workflow

Use the face capture tool to capture a picture with the camera and select which face (green box) to search for. Press the Run Query button and wait for results.

Database Architecture

The architecture is stable, but could be improved for optimal performance and reusability, due to putting more attention into the image processing and face matching features. The database engine is a singleton implementation that manages a serial transaction run loop on a dedicated background thread, mandatory for the CouchbaseLite layer. In most cases, the engine API handles synchronization across threads if necessary. At launch time, the engine instantiates and registers all indexers. The consuming application uses the API in order to extract descriptor data and write it into the database. In doing so, each registered CBIRIndexer is invoked by the framework to write its own descriptor data for the given image into the database, effectively making it possible to store several categories of indexer descriptors for each image. This allows CBIRQuery implementations to leverage the installed indexers and reason with their description data when producing query results.



Proper threading is a concern especially in mobile environments. The underlying database CouchbaseLite, requires all operations to occur on the same thread it was instantiated on. Any violation of this rule immediately results in CouchbaseLite throwing an internal inconsistency exception, which will crash the process. As such, when CBIRQuery objects are executed by the CBIRDatabaseEngine, they're executed serially on the engine's run loop, on the dedicated database thread. Classes derived from CBIRQuery must implement their query logic in the *run* method. This implementation typically involves retrieving, creating, or reading CouchbaseLite resources. Obtaining these resources via the CBIRDatabaseEngine API will block the calling thread if it's not the database thread until the resource has been obtained, otherwise the resource is obtained immediately since the call is being made on the database thread. This is an important distinction to understand, as it's possible to use the CBIRDatabaseEngine API to safely create and retrieve CouchbaseLite resources from any thread, since the API handles synchronization, but reading or manipulating those resources while not on the database thread is illegal, and as such is considered a programming error and results in an internal inconsistency exception.

Preprocessing

FaceIndexer performs several preprocessing operations prior to feature extraction with the intention of reducing noise, accentuating lines, and in general improving potential for accurate feature reasoning. The face detection API of Core Image provides various metadata extracted from the faces in an image, including the rectangle where the face can be found, whether the face is smiling, or even if their eyes are closed. The preprocessing step is most dependent on the finding the rectangle of each face and the angle of the face's orientation. For each face that the API returns, the image is cropped to the face rectangle and is rotated according to the orientation angle. This insures that all face images that are processed have roughly the same orientation.

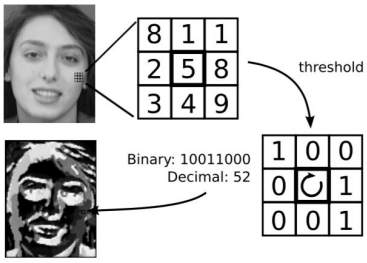
The preprocessor also follows two of Maturana's recommendations. The first is gamma correction with a gamma value of 0.2 to enhance the dynamic range of dark regions and compress

light areas and highlights. Core Image provides a built gamma adjust filter which is used to achieve this task. The next is Difference of Gaussians (DoG) filtering (with standard deviations of 1 and 2) that acts as a bandpass, suppressing high frequency noise and low frequency illumination variation[1]. Core Image does not support a built in DoG filter, but one was easily implemented by subclassing CIFilter to run the input image through two GaussianBlur filters, each with different standard deviations, and both outputs run through a custom CIKernel that computes the difference. Maturana's third preprocessing step is performing contrast equalization in order standardize intensity variation. The formula is quite sophisticated and far more challenging to implement, and was therefore left out for simplicity.

Local Binary Pattern Filtering

Finally, a Local Binary Pattern (LBP) filter is applied to the preprocessed image. The LBP filter is intended to reduce intensity variation to accentuate the lines of the facial features into fundamental shape and contrast patterns, of which histograms can be used to describe and reason with them. The LBP operation is defined to take a grayscale image as input and as output generates each pixel to be an 8 bit binary value, with each bit based on whether each pixel surrounding the current pixel is greater than or equal to (1) or less than (0) the current pixel. Doing so makes each output pixel reflect the energy of surrounding pixels. The standard LBP operation considers the 8 immediate surrounding pixels of each pixel, but other methods may extend the radius further and incorporate more surrounding pixels and even interpolate between pixels for increased accuracy. The reference implementation parallelizes the LBP operation using a custom Core Image kernel program, the language of which is not as dynamic as some LBP implementations would prefer, e.g. loop invariants must be known at kernel compile time and bit shifting operators do not exist. Therefore, the standard 8 surrounding pixels approach is used with bit shifting simply implemented as hard-coded multiplication. One difference from standard implementations is that the LBP filter doesn't convert the image to grayscale first. Instead, it determines the difference of neighboring

pixels according to their vector magnitudes, which may not be the best method of comparison. It may also be beneficial to use a grayscale filter beforehand, though there are so many listed methods of doing so with Core Image, it's possible that any given one may not be entirely correct either.



Output bit order (Maturana).

```
"// Returns 1 if the color at p in the given image is greater than or        \n"
"// equal to the given color, and zero otherwise.                           \n"
"int binColor(sampler img, vec2 p, vec4 cC) {                               \n"
"    vec4 pC = sample(img, samplerTransform(img, p));                       \n"
"                                                                           \n"
"    // Compare the color vectors according to their magnitudes.            \n"
"    // There are probably more visually accurate comparisons.              \n"
"    float pCN = sqrt( (pC.r * pC.r) + (pC.g * pC.g) + (pC.b * pC.b) );     \n"
"    float cCN = sqrt( (cC.r * cC.r) + (cC.g * cC.g) + (cC.b * cC.b) );     \n"
"                                                                           \n"
"                                                                           \n"
"                                                                           \n"
"    if ( pCN >= cCN ) return 1;                                            \n"
"    else return 0;                                                         \n"
"}                                                                          \n"
```

Pixel difference function in LBP kernel (output bit order not shown).

Face Feature Extraction & Reasoning

Facial feature extraction and query semantics are implemented as built-in CBIRIndexer and CBIRQuery implementations, FaceIndexer and FaceQuery respectively. These components utilize the iOS Core Image API to efficiently parallelize many of the image processing tasks necessary for implementation of feature extraction and OpenCV for reasoning with the artifacts of those tasks. FaceIndexer implements a feature extraction algorithm using Local Binary Pattern descriptors, based on one proposed by Maturana.[1]

Maturana presents a feature extraction algorithm which is a variation of Ahonen's well known method[2]. Ahonen's method involves dividing each LBP filtered face image into an NxM grid and concatenating the grayscale histogram of each block together, creating what Ahonen refers to as a *spatially enhanced histogram,* as each histogram can also be weighted according to its spatial importance, e.g. blocks that spatially encompass the eyes can count for more. The spatially enhanced histogram can be compared against other spatially enhanced histograms to determine the closest match (the spatially enhanced histogram with the least difference compared to that of a test

face). Maturana's method follows the same approach, though instead of building one large concatenated histogram, each block histogram (referred to as a *feature*) of the grid is stored separately, allowing for spatial reasoning on a per-feature basis. In Maturana's proposed Naive Bayes Nearest Neighbor system, searches are performed by comparing each feature in the test face image against all features of all faces in each image *G* of the training set (the database in this case), storing each nearest neighboring feature according to *G*. This means that each list G_i may contain features from different face images in G. The resultant image that is ultimately selected is the image associated with the G_i that has the least aggregate difference against the features of the test image.

In practice, Maturana's algorithm has two drawbacks. The first is that the algorithm itself doesn't actually resolve to a matched face. Instead it resolves to an image that may contain a suitable match, e.g. in cases of multiple faces in a gallery image (rather common in smartphone image galleries), the resolved image isn't directly a face, but a set of faces. The NBNN operation is still visiting every feature of every face, it's quite possible to yield the closest face match instead of the image with a potential closest match. The second drawback is an exorbitant performance cost, due to comparing every feature of the input image to every feature of every face image of every image in the training set. In practice, this cost is untenable without parallelization. A solution to parallelizing this computation is proposed later in this paper. To curb computational costs, Maturana adjusts his algorithm to restrict each neighbor in G_i to only come from the same block position of the test face, in what he refers to as Restricted NBNN.

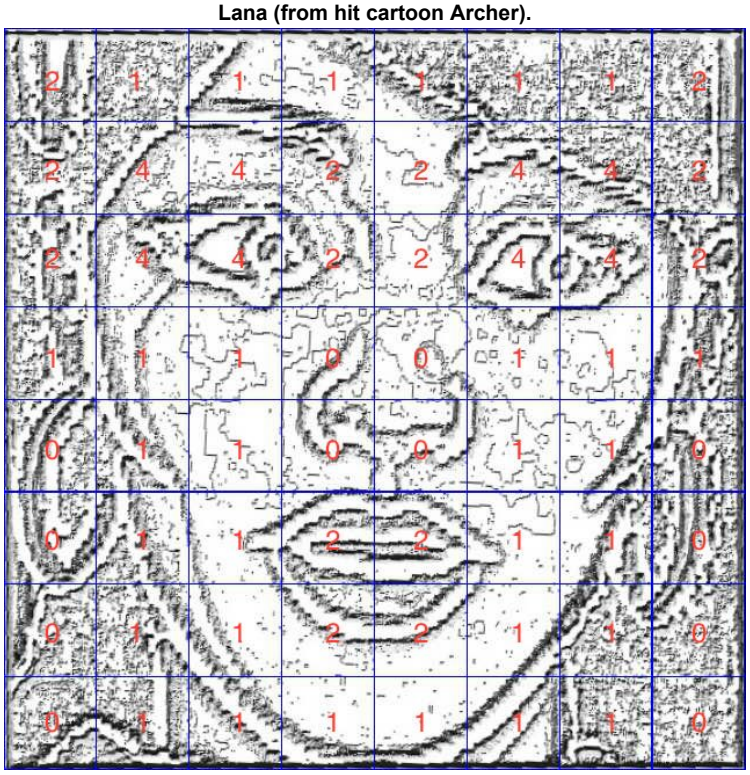
The CBIRDatabase implementation goes a slightly different way that leans to a little closer to Ahonen's method. Like Maturana's, it stores feature descriptors (block histograms) separately, as this permits greater flexibility in reasoning with the feature descriptors at query time. FaceIndexer stores all feature descriptors for each face in each image that it indexes in an image index (a dictionary of feature descriptors and other metadata extracted from the image). The FaceQuery logic iterates every image index object in the database and uses the data stored by FaceIndexer.

Each face feature encountered is compared to the feature at the same position in test face (or input) image. The feature comparison operation is implemented as the product of their Chi-Square difference (provided by OpenCV) and a weight multiplier based on the spatial position of the feature descriptor, fundamentally similar to Ahonen's method.

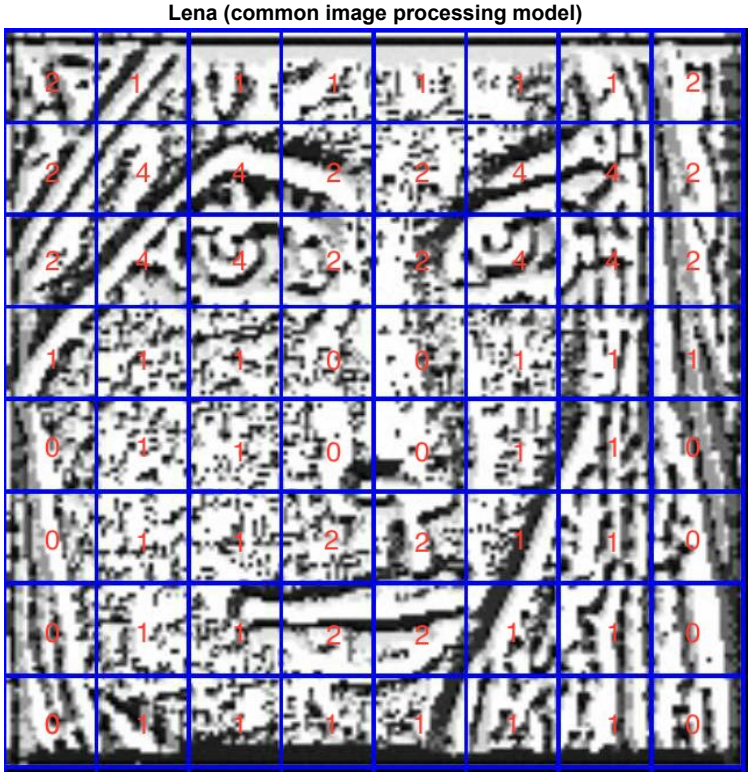
In practice on iOS, the concept of the weight multiplier is very useful. Apple's Core Image provides an API to detect face rectangles in images, which FaceIndexer relies on to find and crop the faces so that it can index just the faces in each image. In many cases though, the spatial boundaries of the face rectangle contain excess image, like the background features that aren't discernible to the face, a person's hair or physical structures behind/around them. This definition of a face is quite different from how research commonly defines the face to be, mostly just face with no background. In practice this tends to bias the difference as the top, bottom, far left, and far right features (28 features in a grid of 64) are simply recording information that is unnecessary, and will significantly contribute to overall face difference without much actual meaning. The use of a spatial weight map inspired by Ahonen allows for intelligently or even dynamically generating it to have 0 or 1 weight in these areas around the edge as well as increase the importance of other areas, e.g. 4 or 8 for features in the region where the eyes would be. This works to avoid the excess noise in the image and allows for preferential feature region mapping. Since the spatial weight map is an operation performed at query time, it can even be dynamic in nature, to allow the caller to specify a custom weight map, increasing their ability to control the results even more.

Visual example of a spatial weight map (not final values).

Lana (from hit cartoon Archer).

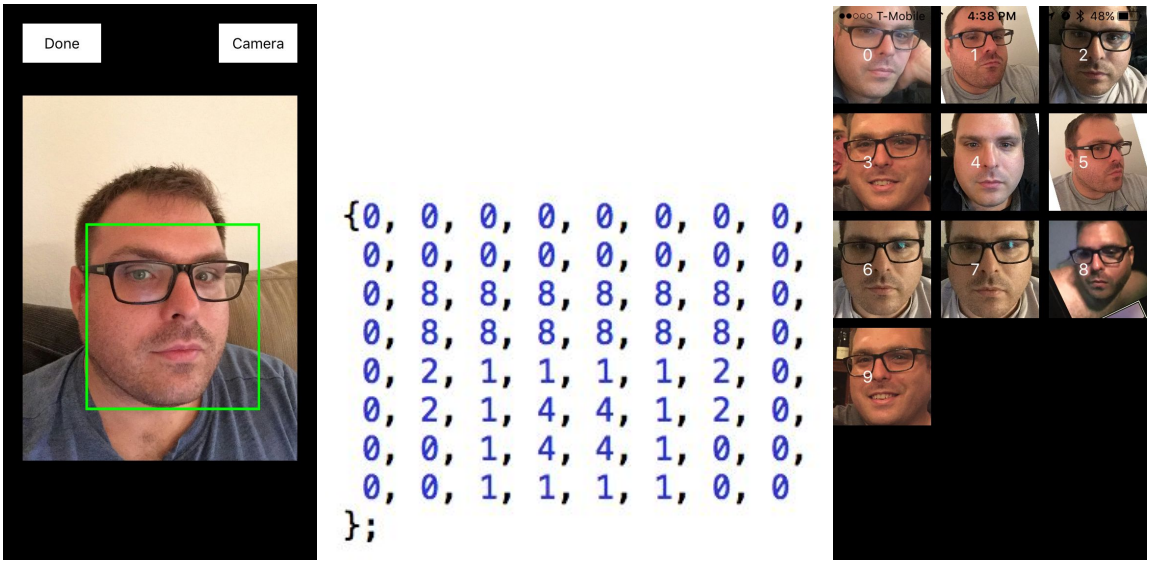


Lena (common image processing model)



Query Performance and Future Optimization

Accuracy tends to the be the most important metric to track. However, given the nature of the training set (unconstrained "wild" images), it's not as feasible to produce metrics that can be objectively compared with how other well known face recognition methods measure up to well known face databases. Empirical results can be observed to get an idea of how the algorithm performs.



Many of my own pictures show me with glasses on and match my face very well. The spatial weight map used shows more weight is put on the eyes and lips.

The design of the spatial weight map may be tailored to better suit the cropped faces that the Core Image face detection API provides. It's fairly obvious that the difference among the input and result face images benefits due to having the sides, top, and part of the bottom not contribute to differences, since they are zero weighted. Those features do not contribute much to the uniqueness of a person's face and in fact may even differ from picture to picture more often than the nose and lips. But the zero weighting also introduces a problem. Observe above that 32 features of the face image are essentially ignored due to being weighted as zero (FaceQuery is optimized to skip

comparison if weight is equal to zero), meaning that only 32 of the feature descriptors are being used to contribute to meaningful face differences, effectively reducing the resolution of the overall comparison. Increasing the grid dimensions from 8x8 to 10x10 or even more, may allow for properly utilizing 8x8 feature descriptors on the face directly, while the outlying features are weighted as zero. Maturana's algorithm suggests that 8x8 descriptor grids (composed nearly entirely of face image) are the most accurate, so it seems feasible that results can be optimized by raising the grid dimensions to 10x10 and updating the spatial weight map to better fit this model.

The difference between NBNN and RNBNN was explained earlier in this paper. The difference being that RNBNN limits the potential nearest neighbor of each feature in the test image to be from the same spatial location of each training face feature. This limits the algorithm from being sensitive to pose or translational transformations. The original NBNN method was designed to seek out the nearest neighbor of each test face feature in every face image, which is sensitive to pose and translational transforms, but the accuracy carries a high computational price tag. If we denote the number faces (subjects) in each image G as n_s, the number of feature descriptors per image as n_D, and the number of images in the database as n_G, then we can see that determining nearest neighbors for all images G has running time $O(n_s \cdot n_D^2 \cdot n_G)$. To put it in perspective, assume there are 2 faces in each image, each with 64 feature descriptors and 400 images total, (2 x 4096 x 400). A single face NBNN search requires 3,276,800 histogram difference operations, one at a time on the CPU. In layman's terms, I tried this method, and a query for a single face took about 30 minutes, which is by no stretch of the imagination feasible for any mobile application. This is a small scale combinatorial explosion problem that can only be remedied with more efficient hardware utilization. Trying a multithreaded solution could even slow it down as most iOS devices are only dual core and context switches are expensive.

A potential solution for curbing this computational cost is implementing the Chi-Square histogram difference function as a set of Core Image filter kernels to parallelize each nearest

neighbor computation. The intention of NBNN is to find the nearest neighboring feature of each test image feature in each training face image. If the face images are broken into 8x8 feature grids, this makes finding each nearest neighbor require 64 Chi-Square difference invocations. But it may be possible to build two filter kernels to implement each nearest neighbor operation in parallel. The training face image must be represented as a contiguous image composed of the feature histograms instead of its pixel data. Then similarly, for each feature in the test face image, we would build a contiguous image composed of the same feature histogram for all features. The result would be two memory buffers (images) filled with 256 bin histograms. Even if the original face images are different sizes, their histograms are always equal in dimension and data type, and both images have the same number of histograms, making both histogram images size compatible. Both images must be loaded into CIImage objects with data aligned properly according to grid rows and can be fed into a Core Image kernel that outputs the square difference ratio between both values at each pixel index. That output image can then be fed through another Core Image kernel that outputs the summation of each 256 component block (if the pixel is at the first block of each histogram) and zero otherwise. The calling code then would only need to run through the output buffer and find the smallest difference by reading the first index of each block and skipping the rest of the block. Of course in practice, this paradigm isn't as trivial to implement, due to constraints that Core Image puts on input data format. Another concern are any other aspects of the overall algorithm that could potentially be impacted by using this method, an example being that it's more efficient to build the full histogram image at index time so that there it doesn't have to be done at query time. It's also unclear whether this degree of parallelism would yield more reasonable performance results in the long run, but it's a good candidate for further research. The reference implementation contains a partially complete implementation that doesn't quite work for the aforementioned reasons and instead relies on the RNBNN-like method.

Proposed Parallelized Chi-Square Histogram Difference

```
// Consider the following approach.  Using two 4x4 block histogram images computing the nearest neighbor
// between e[3] inside the entire histogram training image t[i].  This process can be repeated for each
// training face against each histogram block in the source image.
//
// Expected(e)      Training_i(ti)
// [3][3][3][3]     [0 ][1 ][2 ][3 ]      [e3 - ti0 ][e3 - ti1 ][e3 - ti2 ][e3 - ti3 ]
// [3][3][3][3]  -  [4 ][5 ][6 ][7 ]  =   [e3 - ti4 ][e3 - ti5 ][e3 - ti6 ][e3 - ti7 ]
// [3][3][3][3]     [8 ][9 ][10][11]      [e3 - ti8 ][e3 - ti9 ][e3 - ti10][e3 - ti11]
// [3][3][3][3]     [12][13][14][15]      [e3 - ti12][e3 - ti13][e3 - ti14][e3 - ti15]
//
```

```
"kernel vec4 histogramDiff ( sampler expect, sampler training )              \n"
"{                                                                           \n"
"                                                                            \n"
"    // Recall that we're only concerned with the red component.             \n"
"    float expIntensity = samplerCoord(expect).r;                            \n"
"    float trainIntensity = samplerCoord(training).r;                        \n"
"                                                                            \n"
"    float diff = expIntensity - trainIntensity;                             \n"
"    float diffSquare = diff * diff;                                         \n"
"    float diffSquareExpRatio = diffSquare / expIntensity;                   \n"
"                                                                            \n"
"    return vec4(diffSquareExpRatio, 0, 0, 1);                               \n"
"}                                                                           \n";
```

```
// if ( destCoord().x % binCount == 0 )
//    end = destCoord.x + binCount;
//    for ( x = destCoord.x; x < end; x++ )
//        sum += sample(s, samplerTransform(s, vec2(x, destCoord.y)));
```

In conclusion, CBIR systems are definitely feasible in iOS and other mobile/embedded environments, provided they offer access to hardware acceleration and are utilized for modest purposes. One can't expect the same performance of a supercomputer tuned for image processing stacks, but for local indexing and reasoning of personal image content, the possibilities are plentiful. Face recognition is reasonably possible given a closed system of training images, but needs improvement in accuracy and performance. Like many great research topics in computer science it requires getting cozy with the services the system offers so that they can be used to full advantage.

References

1. **Maturana, Daniel, Domingo Mery, and Álvaro Soto. "Face Recognition with Local Binary Patterns, Spatial Pyramid Histograms and Naive Bayes Nearest Neighbor Classification."** ***2009 International Conference of the Chilean Computer Science Society*** **(2009). Database.**
2. **Ahonen, T., A. Hadid, and M. Pietikainen. "Face Description with Local Binary Patterns: Application to Face Recognition."** ***IEEE Transactions on Pattern Analysis and Machine Intelligence IEEE Trans. Pattern Anal. Machine Intell.*** **28.12 (2006): 2037-041. Web. 3 Dec. 2015.**