Expected Maximization as part of a Bayesian/Gaussian Framework

Dan Beatty

June 28, 2007

1 Overview of EM

The goal of the EM algorithm is to determine a set of parameters that define a statistical model given a certain data set. If the data set is incomplete, then is some representation of the complete data set. If the data set can be represented by a set of statistical distributions, then EM can be used to obtain the sufficient statistics of those distributions. EM extends properties of maximum likelihood estimation methods by determining via iterative convergence the governing parameters of a particular data-set. In particular, uncorrupted cases may use sufficient statistics acquired via MLE, [**?**]. In general, the equation for the terminating condition of the EM algorithm [**?**] is stated in equation 1 . The meaning of the terms in equation 1 are as follows:

- $\mathbf{X}$ is the true data set, $\vec{x}$ is the true data set sample,
- $\vec{\Theta}$ are the set of parameters for the statistic, and
- $\vec{y}$ is the data sample in the given data set.

$$\vec{\Theta}^* = \arg\max_{\Theta} \sum_{\vec{x} \in \mathcal{X}^n} E[\ln p(\mathbf{x}|\vec{\theta}, \vec{y})] \tag{1}$$

Equation 1 may be refined into 2 with the following notation obtained from [**?**]:

$$Q(\vec{\theta}; \vec{\theta^i}) = E_{D_b}[\ln p(D_g, D_b; \vec{\theta})|D_g; \vec{\theta^i}] \tag{2}$$

- $\vec{\theta^i}$ is the current (best) estimate for the full distribution;
- $\vec{\theta}$ is a candidate vector for an improved estimate
- $Q(\vec{\theta}; \theta^i)$ is a function of $\vec{\theta}$ and $\vec{\theta^i}$
- D_b is the actual data set.

- D_g is the unknown and uncorrupted data set.
- $E_{D_b}[\ln p(D_g, D_b; \vec{\theta})|D_g; \vec{\theta^i}]$ is the expected value over the missing features. The expected value hinges on $\vec{\theta^i}$ which are the true parameters.

One of the goals is marginalize D_b with respect to $\vec{\theta^i}$. Another goal of the EM algorithm is select from the candidate $\vec{\theta}$ from a set of $\vec{\theta}$s, and iterate to a $\vec{\theta^{i+1}}$ which yields the greatest $Q(\vec{\theta}; \vec{\theta^i})$. In order for EM to work the assumption of that D_g is an independent and identically distributed (i.i.d) observations must hold.

1.1 Specification for Gaussian Case

In the Gaussian case, each class has a proportion that defines how much its class contributes, a mean and covariance (denoted $\alpha, \vec{\mu}, \mathbf{\Sigma}$ respectively) . If one acquires a sample as a initial set, the question is does an EM algorithm that will refine these sufficient statistics? In [**?**], there is an example using a matrix as the result of the expectation step. From this expectation, the sufficient statistics for the next step is computed. The expected equation is as in equation 3 forms a matrix $\mathbf{A}$ called the expected matrix.

$$a_{ij}^{(k)} = \frac{\alpha_j p(\vec{y_i}^{(k)}|\vec{\mu_j}^{(k)}, \Sigma_j^{(k)})}{\sum_{j=1}^{M} \alpha_j p(\vec{y_i}^{(k)}|\vec{\mu_j}^{(k)}, \Sigma_j^{(k)})} \tag{3}$$

The three sufficient statistics are computed in the maximization step. It is computed by

$$\vec{\mu}_j^{(k+1)} = \frac{\sum_{i=1}^{N} a_{ij}^{(p)} \vec{y}_i}{\sum_{i=1}^{N} a_{ij}} \tag{4}$$

$$\mathbf{\Sigma}_j^{(k+1)} = \frac{\sum_{i=1}^{N} (\vec{y}_i \vec{\mu}_j^{(p)})^T (\vec{y}_i \vec{\mu}_j^{(p)})}{\sum_{i=1}^{N} a_{ij}^{(k)}} \tag{5}$$

$$\alpha_j^{(k+1)} = \frac{1}{N} \sum_{i=1}^{N} a_{ij}^{(k)} \tag{6}$$

The obvious question is what reduction on $\mathbf{A}$ is used to assess the convergence of the estimations.

$$Q(\theta^*; \theta) = \log(\prod_{i=1}^{N} a_{ii}) \tag{7}$$

Another observation, it is clear that the initial guess can not be the zero matrix. As such, the sufficient statistics would be rendered zero, and no convergence would occur.

Typically, the guesses are for μ to be scattered for each class, and for the expected matrix to be

$$\mathbf{A} = \frac{1}{N}\mathbf{I}.$$

In the most basic notion, the EM Algorithm should be implemented as stated in its algorithm, Algorithm 1. Such an Objective C implementation is listed in 1. Notice though the expected step and maximization step are separate methods, and may be considered private methods and overridden in each subclass. In this case, they are publicly accessible although only the computeEM and probability classes need to be called to obtain the parameters.

Algorithm 1 Expectation Maximization

initialize $\vec{\theta}^0$, T, and $i \leftarrow 0$
repeat
  $i \leftarrow i + 1$
  E Step: compute $Q(\vec{\theta}; \vec{\theta}^i)$
  M step: $\theta^{i+1} \rightarrow \arg\max_\theta Q(\vec{\theta}; \vec{\theta}^i)$
until $Q(\vec{\theta}^{i+1}; \vec{\theta}^i) - Q(\vec{\theta}^i; \vec{\theta}^{i-1}) \leq T$
return $\hat{\vec{\theta}} \rightarrow \vec{\theta}^{i+1}$

Listing 1: Compute EM

```
-(void) computeEM
{
        epsilon = 0.05;
        [self estimationStep];
        do
        {
                lastQ = Q;
                [self maximizationStep];
                [self estimationStep];
        } while ( fabs(Q- lastQ) / Q > epsilon);

}
```

In even the simplicity of the univariate case, there is the possibility of EM algorithm to require many iterations. A consequence are the results from each iterative step. In some cases, these steps may be discarded. Others they may be retained. In this implementation they are discarded. The presumption is that sufficient statistics computed in both the last and next to last maximization step have in fact converged. This is measured by the

values of the estimation function computed in the estimation step. While it is a waste to compute the maximization upon convergence, it is also harmless. Thus the initialization of the univariate object is defined in listing 2.

Listing 2: Initialize EM

```
-(id) initWithSamples:(dcgVector *) someSamples
                                    numberOfGaussians:(int) M
{
        [super init];
        numberOfClasses = M;
        numberOfSamples = [someSamples vecLength];
        samples = [someSamples retain];

        double max = [samples max];
        double min = [samples min];

        double scale = (max - min)  / M ;
//      scale *= 0.5;
        int r;

        mean = [[dcgVector alloc] initWithLength:numberOfClasses];
        variance = [[dcgVector alloc] initWithLength:numberOfClasses];
        proportions = [[dcgVector alloc] initWithLength:numberOfClasses];
        mixture = [[dcgMatrix alloc] initWithRows:numberOfSamples
                                    columns:numberOfClasses
                                    ];
        double *mu = [mean localVector];
        double *sigma = [variance localVector];
        double *alpha = [proportions localVector];

        for ( r = 0; r < M; r++) {
                mu[r] = (max - min ) * r * scale + min ;
                sigma[r] = scale;
                alpha[r] = scale;
        }

        [self computeEM];
        return self;
}
```

One note for improvement would be the implementation of EM Gaussian Univariate, a structure which houses the results of each step. This where the loop initializing the Gaussian classes should actually be located.

2 First Experiments

One of the first experiments tried on the EM segmentation experiment was conducted in Octave (Matlab's Open Source cousin). In this example, I fed in the original optic disc image along with the request for 5 classifications. The results are seen in figures 3 and 4. One of the things noticed in this implementation of the EM segmentation is the use of histograms as opposed to generating a Gaussian structure for the sample data.

The Octave version supply some means for comparing a unit test of the Objective C version. However, most unit tests do not require a document window to allow the results of each unit to be selected and examined individually or in a combined fashion. All of the EM implementations generate a set of statistical classes defined by a set of sufficient statistics. Each collection provides a method of classifying every subsequent element in to one of these classifications. In the case of image processing, these classifications represent a collection of masks. These masks identify segments to either be retained or eliminated.

A consequence of this need is that the interface consists of two additional sets of windows or panels. One of these panels contains a table or listing of each component of the sample image. The other references the controller's array of masks. The checking of a box in the masks panel applies the mask to the original image, and adds it to the result. The removing of the check of a mask, subtracts that application of the mask.

Selecting a component causes the univariate EM to be computed for that component. The removing of such a selection must inherently remove the selection from its associated masks, as they would be destroyed in the process. Also, one other design decision solely for the purpose of the experiment the number of classes are determined by a slider control. As such, this number is applied to all sub-components. A design depiction of this panel is shown in figure.

2.1 Core Graphics Version

The two steps for EM are explicitly stated in emUnivariateGaussian. One consequence of this development are revelations about necessary steps for its computation. The estimation step is computed implicitly first before the loop and again at the end of the loop. The reason is to compute the estimation termination value. Another revelation, in listing 4, shows the need for careful zeroizing of summing values. The previous mean is needed in computing the maximization steps variance. Therefore caution must be applied as to when the mean is computed.

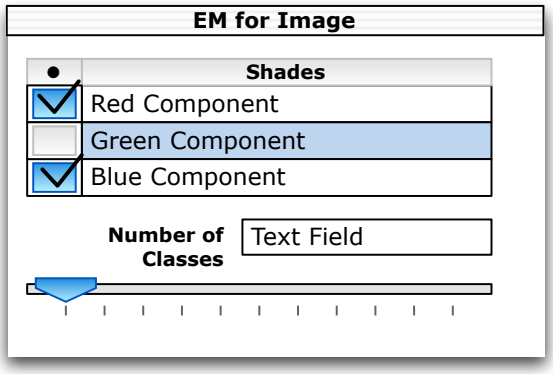

Figure 1: The results of an EM for Image a set of segmenting classifiers, and the masks that result from the classification

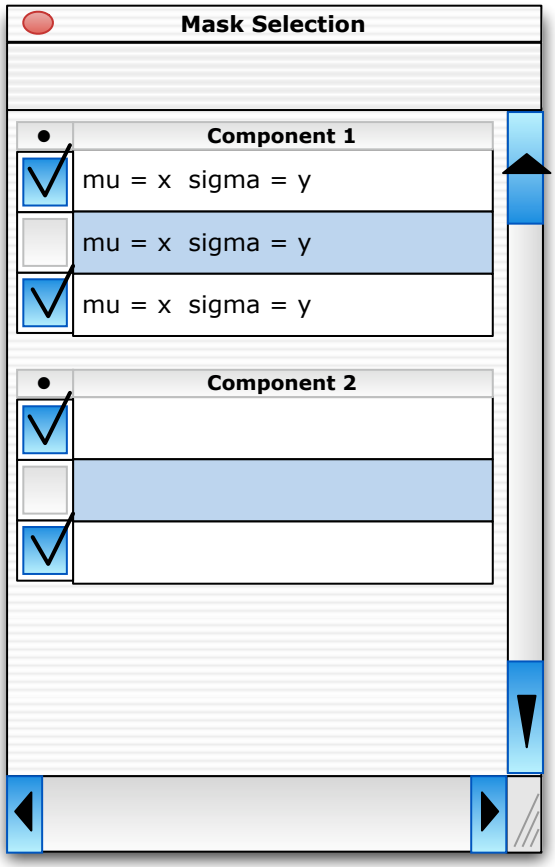

Figure 2: This holds the masks generated by the EM algorithm

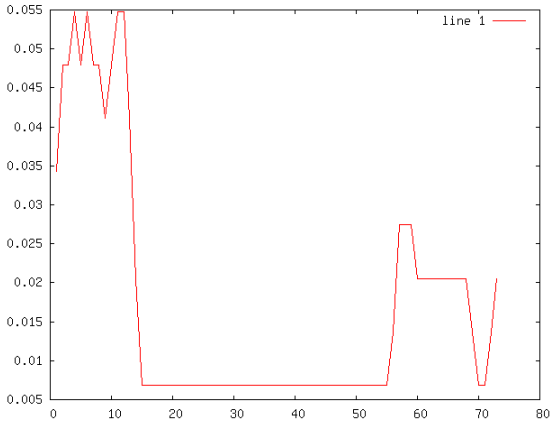

Figure 3: Plot of Guassians for the Optic Disc

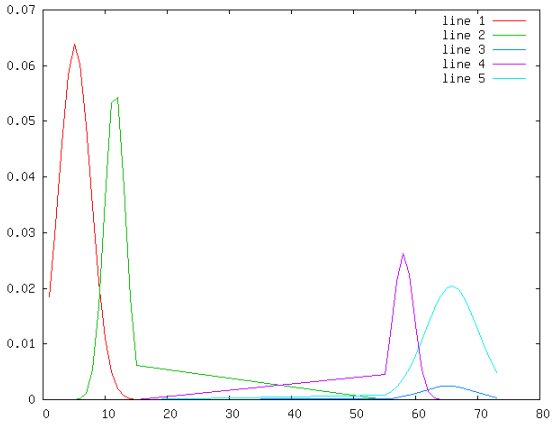

Figure 4: Plot of sub-Gaussians for the Optic Disc

Listing 3: Estimation step of EM, computed explicitly in Objective C

```
-(void) estimationStep
{
        double *mu = [mean localVector];
        double *sigma = [variance localVector];
        double *alpha = [proportions localVector];
        double **A = [mixture localMatrix];
        double *D = [samples localVector];

        double amp, sumAlpha = 0;
        Q = 0;
        int r, c;

        for ( r = 0 ; r < numberOfSamples; r++)
                for ( c = 0; c < numberOfClasses; c++)
                {
                        amp = alpha[c];
                        amp /= sqrt(2 *M_PI * sigma[c]);
                        A[r][c] = amp * exp (-0.5 * (D[r] - mu[c])
* (D[r] -mu[c]) / sigma[c]) ;
                }

        for ( r = 0 ; r < numberOfSamples; r++)
        {
                sumAlpha = 0;
                for ( c = 0; c < numberOfClasses; c++)
                        sumAlpha += A[r][c] ;
                for ( c = 0; c < numberOfClasses; c++)
                        A[c][r] /= sumAlpha ;
        }

        for ( r = 0; r < numberOfSamples; r++)
        {
                 sumAlpha = 0.0;
                 for ( c = 0; c < numberOfClasses; c++)
                        sumAlpha += A[r][c] ;
                 Q += fabs(log(sumAlpha));
        }
```

```
}
```

Listing 4: Maximization step of EM, computed explicitly in Objective C

```
-(void) maximizationStep
{
        double *mu = [mean localVector];
        double *sigma = [variance localVector];
        double *alpha = [proportions localVector];
        double **A = [mixture localMatrix];
        double *Y = [samples localVector];

        int r, c;
        double meanDifference;


        [proportions zeroize];
        for ( r =0; r < numberOfSamples; r++)
                for ( c = 0; c < numberOfClasses; c++)
                        alpha[c] += A[r][c];

        [variance zeroize];
        for ( r =0; r < numberOfSamples; r++)
                for ( c = 0; c < numberOfClasses; c++)
                {
                        meanDifference = Y[r] - mu[c];
                        sigma[c]  += A[r][c] * meanDifference * meanDifference;
                }
        for ( c = 0; c < numberOfClasses; c++)
                        sigma[c] /= alpha[c];


        [mean zeroize];
        for ( r =0; r < numberOfSamples; r++)
                for ( c = 0; c < numberOfClasses; c++)
                        mu[c]  += A[r][c] * Y[r];

        for ( c = 0; c < numberOfClasses; c++)
                        mu[c] /= alpha[c];
```

```
        for ( c = 0; c < numberOfClasses; c++)
                alpha[c] /= numberOfSamples;

}
```

Other components are needed in an Objective-C version to make it more viable, and this is where design decisions must be considered. In the initial design, obtaining the image data itself was a matter of drawing to a Core Graphics Grey-Scale Bitmap and coping the data out. This forces the computation to be CPU bound. Either CPU bound or Graphics Processing Unit bound versions are acceptable. This subsection illustrates the CPU bound version. A future report shall review the GPU bound equivalent.

In order to continue with a DCGMatrix obtained from a Core Graphics Grey-Scale Bitmap, it must be transformed into a vector of equal number of elements. This vector and the number of desired classifications must then initialize an emUnivariateGaussian. Once initialized, it is convenient to extract the results into an NSArray containing the sufficient statistics. Such an array need not retain the proportions value.

Once obtain the array of sufficient statistics becomes the classifier for every grey scale value in the image. In this case, the user is allowed to select which classifiers he/she wishes to see. It is the point of selecting classifiers that a mask is drawn. A mask has the sole purpose of blending sections of the image. In this case, either the color point belongs or it does not. It is here that Core Graphics Data Providers and Image Providers become useful. Also, it is here where the DCGMatrix obtained from the Grey-Scale Bitmap is reused, and fed into the classifier. If the point belongs, a white value is inserted into the mask. The mask is black otherwise. Note that a true color filter would have use a multivariate variety of the classifier as it is a vector (color vector).

It should be noted that obtaining the DCGMatrix from the Core Graphics Grey Scale Bitmap is quite tricky. It is one section that is anything, but universal between PPC and x86 architectures. Care has to be taken to ensure byte ordering on the bitmap. Otherwise, the results are garbage, and the EM will compute nonsense.

The first successful version used a subclass of NSOpenGLView and model controller to test the concept of an EM filter. A better version should move the computation to a CIFilter, and use threads to move the computation away from the main display thread. Otherwise, the intensity of computation can overwhelm the main thread, and waste the existence of multiple-core CPUs and computational grids. With the first version, the control includes a panel to show the segments (displayed by mean and variance show in figure 5) and view window that shows the image masked with the selected segment. If no segment is selected, the mask is clear and shows the original image, shown in figure 6.

There is the possibility of computing the EM directly from the Graphics Card, but that implies providing an expected step kernel, a maximization step kernel, termination test kernel.

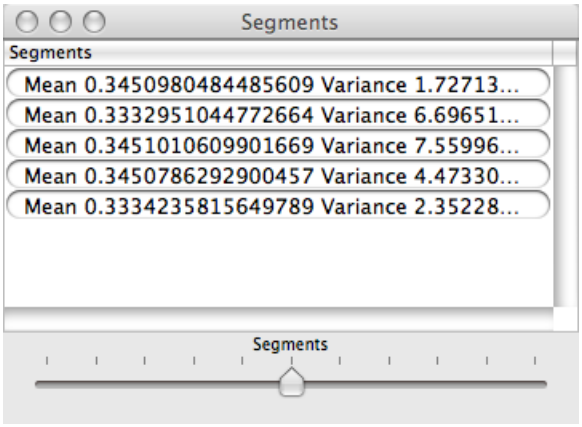

Figure 5: EM Segments shown by mean and variance for a photo of a human iris.

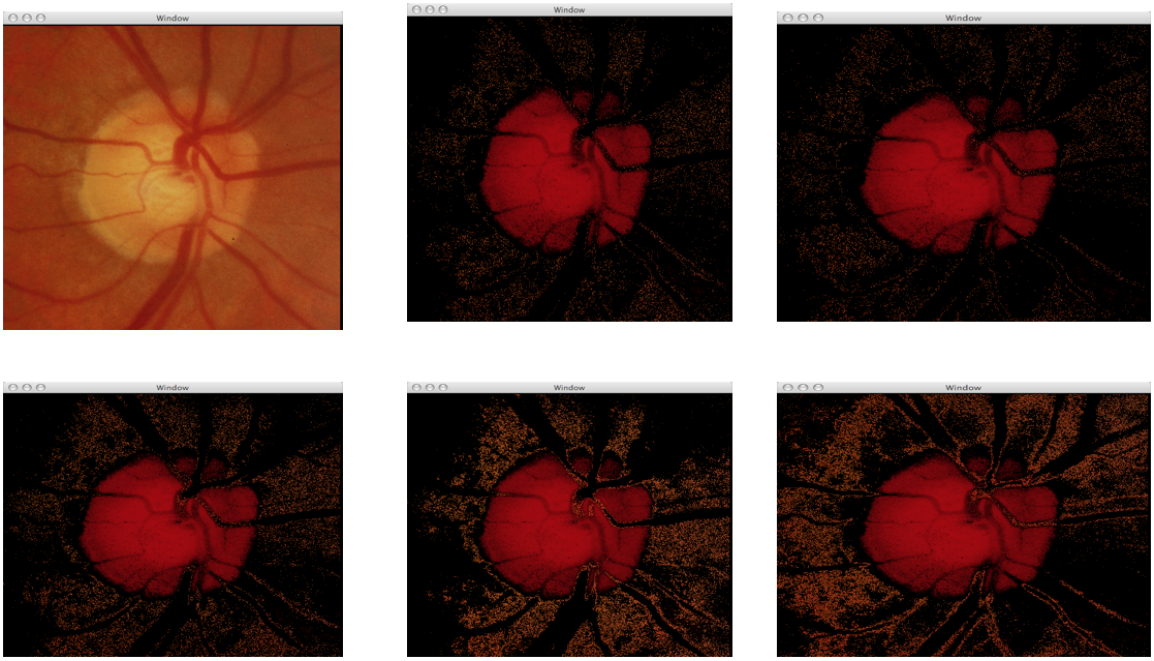

Figure 6: Both the original image and 5 EM Segments of a human iris.