Fitting Gaussian Mixture Models with EM

In this assignment you will

- implement the EM algorithm for a Gaussian mixture model
- apply your implementation to cluster images
- explore clustering results and interpret the output of the EM algorithm

Note to Amazon EC2 users: To conserve memory, make sure to stop all the other notebooks before running this notebook.

Import necessary packages

The following code block will check if you have the correct version of GraphLab Create. Any version later than 1.8.5 will do. To upgrade, read this page (https://turi.com/download/upgrade-graphlab-create.html).

In [1]:
```
import graphlab as gl
import numpy as np
import matplotlib.pyplot as plt
import copy
from scipy.stats import multivariate_normal

%matplotlib inline

'''Check GraphLab Create version'''
from distutils.version import StrictVersion
assert (StrictVersion(gl.version) >= StrictVersion('1.8.5')), 'GraphLab Create
 must be version 1.8.5 or later.'
```

```
[INFO] graphlab.cython.cy_server: GraphLab Create v2.1 started. Logging: /tmp/g
raphlab_server_1478355903.log
INFO:graphlab.cython.cy_server:GraphLab Create v2.1 started. Logging: /tmp/grap
hlab_server_1478355903.log

This non-commercial license of GraphLab Create for academic use is assigned to
tmandzak@gmail.com and will expire on May 12, 2017.
```

Implementing the EM algorithm for Gaussian mixture models

In this section, you will implement the EM algorithm. We will take the following steps:

- Provide a log likelihood function for this model.
- Implement the EM algorithm.
- Create some synthetic data.
- Visualize the progress of the parameters during the course of running EM.
- Visualize the convergence of the model.

Log likelihood

We provide a function to calculate log likelihood for mixture of Gaussians. The log likelihood quantifies the probability of observing a given set of data under a particular setting of the parameters in our model. We will use this to assess convergence of our EM algorithm; specifically, we will keep looping through EM update steps until the log likehood ceases to increase at a certain rate.

In [2]:
```
def log_sum_exp(Z):
    """ Compute log(\sum_i exp(Z_i)) for some array Z."""
    return np.max(Z) + np.log(np.sum(np.exp(Z - np.max(Z))))

def loglikelihood(data, weights, means, covs):
    """ Compute the loglikelihood of the data for a Gaussian mixture model wit
h the given parameters. """
    num_clusters = len(means)
    num_dim = len(data[0])

    ll = 0
    for d in data:

        Z = np.zeros(num_clusters)
        for k in range(num_clusters):

            # Compute (x-mu)^T * Sigma^{-1} * (x-mu)
            delta = np.array(d) - means[k]
            exponent_term = np.dot(delta.T, np.dot(np.linalg.inv(covs[k]), del
ta))

            # Compute loglikelihood contribution for this data point and this
cluster
            Z[k] += np.log(weights[k])
            Z[k] -= 1/2. * (num_dim * np.log(2*np.pi) + np.log(np.linalg.det(c
ovs[k])) + exponent_term)

        # Increment loglikelihood contribution of this data point across all c
lusters
        ll += log_sum_exp(Z)

    return ll
```

E-step: assign cluster responsibilities, given current parameters

The first step in the EM algorithm is to compute cluster responsibilities. Let r_{ik} denote the responsibility of cluster k for data point i. Note that cluster responsibilities are fractional parts: Cluster responsibilities for a single data point i should sum to 1.

$$r_{i1} + r_{i2} + \ldots + r_{iK} = 1$$

To figure how much a cluster is responsible for a given data point, we compute the likelihood of the data point under the particular cluster assignment, multiplied by the weight of the cluster. For data point i and cluster k, this quantity is

$$r_{ik} \propto \pi_k N(x_i|\mu_k, \Sigma_k)$$

where $N(x_i|\mu_k, \Sigma_k)$ is the Gaussian distribution for cluster k (with mean μ_k and covariance Σ_k).

We used $\propto$ because the quantity $N(x_i|\mu_k, \Sigma_k)$ is not yet the responsibility we want. To ensure that all responsibilities over each data point add up to 1, we add the normalization constant in the denominator:

$$r_{ik} = \frac{\pi_k N(x_i|\mu_k, \Sigma_k)}{\sum_{k=1}^{K} \pi_k N(x_i|\mu_k, \Sigma_k)}.$$

Compute the following function that computes r_{ik} for all data points i and clusters k.

Drawing from a Gaussian distribution. SciPy provides a convenient function multivariate_normal.pdf (http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.multivariate_normal.html) that computes the likelihood of seeing a data point in a multivariate Gaussian distribution. The usage is

```
multivariate_normal.pdf([data point], mean=[mean vector], cov=[covariance matrix])
```

In [7]:
```
def compute_responsibilities(data, weights, means, covariances):
    '''E-step: compute responsibilities, given the current parameters'''
    num_data = len(data)
    num_clusters = len(means)
    resp = np.zeros((num_data, num_clusters))

    # Update resp matrix so that resp[i,k] is the responsibility of cluster k
for data point i.
    # Hint: To compute likelihood of seeing data point i given cluster k, use
multivariate_normal.pdf.
    for i in range(num_data):
        for k in range(num_clusters):
            # YOUR CODE HERE
            resp[i, k] = weights[k]*multivariate_normal.pdf(data[i], means[k],
 covariances[k])

    # Add up responsibilities over each data point and normalize
    row_sums = resp.sum(axis=1)[:, np.newaxis]
    resp = resp / row_sums

    return resp
```

Checkpoint.

In [8]:
```
resp = compute_responsibilities(data=np.array([[1.,2.],[-1.,-2.]]), weights=np
.array([0.3, 0.7]),
                                 means=[np.array([0.,0.]), np.array([1.,1.])],
                                 covariances=[np.array([[1.5, 0.],[0.,2.5]]), n
p.array([[1.,1.],[1.,2.]])])

if resp.shape==(2,2) and np.allclose(resp, np.array([[0.10512733, 0.89487267],
 [0.46468164, 0.53531836]])):
    print 'Checkpoint passed!'
else:
    print 'Check your code again.'
```

```
Checkpoint passed!
```

M-step: Update parameters, given current cluster responsibilities

Once the cluster responsibilities are computed, we update the parameters (weights, means, and covariances) associated with the clusters.

Computing soft counts. Before updating the parameters, we first compute what is known as "soft counts". The soft count of a cluster is the sum of all cluster responsibilities for that cluster:

$$N_k^{\text{soft}} = r_{1k} + r_{2k} + \ldots + r_{Nk} = \sum_{i=1}^{N} r_{ik}$$

where we loop over data points. Note that, unlike k-means, we must loop over every single data point in the dataset. This is because all clusters are represented in all data points, to a varying degree.

We provide the function for computing the soft counts:

In [9]:
```
def compute_soft_counts(resp):
    # Compute the total responsibility assigned to each cluster, which will be
 useful when
    # implementing M-steps below. In the lectures this is called N^{soft}
    counts = np.sum(resp, axis=0)
    return counts
```

Updating weights. The cluster weights show us how much each cluster is represented over all data points. The weight of cluster k is given by the ratio of the soft count N_k^{soft} to the total number of data points N:

$$\hat{\pi}_k = \frac{N_k^{\text{soft}}}{N}$$

Notice that N is equal to the sum over the soft counts N_k^{soft} of all clusters.

Complete the following function:

In [10]:
```
def compute_weights(counts):
    num_clusters = len(counts)
    weights = [0.] * num_clusters
    N = sum(counts)

    for k in range(num_clusters):
        # Update the weight for cluster k using the M-step update rule for the
 cluster weight, \hat{\pi}_k.
        # HINT: compute # of data points by summing soft counts.
        # YOUR CODE HERE
        weights[k] = counts[k]/N

    return weights
```

Checkpoint.

In [11]:
```
resp = compute_responsibilities(data=np.array([[1.,2.],[-1.,-2.],[0,0]]), weig
hts=np.array([0.3, 0.7]),
                                means=[np.array([0.,0.]), np.array([1.,1.])],
                                covariances=[np.array([[1.5, 0.],[0.,2.5]]), n
p.array([[1.,1.],[1.,2.]])])
counts = compute_soft_counts(resp)
weights = compute_weights(counts)

print counts
print weights

if np.allclose(weights, [0.27904865942515705, 0.720951340574843]):
    print 'Checkpoint passed!'
else:
    print 'Check your code again.'
```

```
[ 0.83714598  2.16285402]
[0.27904865942515705, 0.720951340574843]
Checkpoint passed!
```

Updating means. The mean of each cluster is set to the weighted average (https://en.wikipedia.org/wiki/Weighted_arithmetic_mean) of all data points, weighted by the cluster responsibilities:

$$\hat{\mu}_k = \frac{1}{N_k^{\text{soft}}} \sum_{i=1}^{N} r_{ik} x_i$$

Complete the following function:

In [14]:
```
def compute_means(data, resp, counts):
    num_clusters = len(counts)
    num_data = len(data)
    means = [np.zeros(len(data[0]))] * num_clusters

    for k in range(num_clusters):
        # Update means for cluster k using the M-step update rule for the mean
 variables.
        # This will assign the variable means[k] to be our estimate for \hat{\
mu}_k.
        weighted_sum = 0.
        for i in range(num_data):
            # YOUR CODE HERE
            weighted_sum += resp[i, k] * data[i]
        # YOUR CODE HERE
        means[k] = weighted_sum / counts[k]

    return means
```

Checkpoint.

In [15]:
```
data_tmp = np.array([[1.,2.],[-1.,-2.]])
resp = compute_responsibilities(data=data_tmp, weights=np.array([0.3, 0.7]),
                                means=[np.array([0.,0.]), np.array([1.,1.])],
                                covariances=[np.array([[1.5, 0.],[0.,2.5]]), n
p.array([[1.,1.],[1.,2.]])])
counts = compute_soft_counts(resp)
means = compute_means(data_tmp, resp, counts)

if np.allclose(means, np.array([[-0.6310085, -1.262017], [0.25140299, 0.502805
99]])):
    print 'Checkpoint passed!'
else:
    print 'Check your code again.'
```

```
Checkpoint passed!
```

Updating covariances. The covariance of each cluster is set to the weighted average of all outer products (https://people.duke.edu/~ccc14/sta-663/LinearAlgebraReview.html), weighted by the cluster responsibilities:

$$\hat{\Sigma}_k = \frac{1}{N_k^{\text{soft}}} \sum_{i=1}^{N} r_{ik}(x_i - \hat{\mu}_k)(x_i - \hat{\mu}_k)^T$$

The "outer product" in this context refers to the matrix product

$$(x_i - \hat{\mu}_k)(x_i - \hat{\mu}_k)^T.$$

Letting $(x_i - \hat{\mu}_k)$ to be $d \times 1$ column vector, this product is a $d \times d$ matrix. Taking the weighted average of all outer products gives us the covariance matrix, which is also $d \times d$.

Complete the following function:

In [16]:
```
def compute_covariances(data, resp, counts, means):
    num_clusters = len(counts)
    num_dim = len(data[0])
    num_data = len(data)
    covariances = [np.zeros((num_dim,num_dim))] * num_clusters

    for k in range(num_clusters):
        # Update covariances for cluster k using the M-step update rule for co
variance variables.
        # This will assign the variable covariances[k] to be the estimate for
\hat{\Sigma}_k.
        weighted_sum = np.zeros((num_dim, num_dim))
        for i in range(num_data):
            # YOUR CODE HERE (Hint: Use np.outer on the data[i] and this clust
er's mean)
            weighted_sum += resp[i, k] * np.outer(data[i] - means[k], data[i]
- means[k])
        # YOUR CODE HERE
        covariances[k] = weighted_sum / counts[k]

    return covariances
```

Checkpoint.

In [17]:
```
data_tmp = np.array([[1.,2.],[-1.,-2.]])
resp = compute_responsibilities(data=data_tmp, weights=np.array([0.3, 0.7]),
                                means=[np.array([0.,0.]), np.array([1.,1.])],
                                covariances=[np.array([[1.5, 0.],[0.,2.5]]), n
p.array([[1.,1.],[1.,2.]])])
counts = compute_soft_counts(resp)
means = compute_means(data_tmp, resp, counts)
covariances = compute_covariances(data_tmp, resp, counts, means)

if np.allclose(covariances[0], np.array([[0.60182827, 1.20365655], [1.20365655
, 2.4073131]])) and \
    np.allclose(covariances[1], np.array([[ 0.93679654, 1.87359307], [1.873593
07, 3.74718614]])):
    print 'Checkpoint passed!'
else:
    print 'Check your code again.'
```

```
Checkpoint passed!
```

The EM algorithm

We are almost done. Let us write a function that takes initial parameter estimates and runs EM. You should complete each line that says `# YOUR CODE HERE`.

In [18]:
```
# SOLUTION
def EM(data, init_means, init_covariances, init_weights, maxiter=1000, thresh=
1e-4):

    # Make copies of initial parameters, which we will update during each iter
ation
    means = init_means[:]
    covariances = init_covariances[:]
    weights = init_weights[:]

    # Infer dimensions of dataset and the number of clusters
    num_data = len(data)
    num_dim = len(data[0])
    num_clusters = len(means)

    # Initialize some useful variables
    resp = np.zeros((num_data, num_clusters))
    ll = loglikelihood(data, weights, means, covariances)
    ll_trace = [ll]

    for it in range(maxiter):
        if it % 5 == 0:
            print("Iteration %s" % it)

        # E-step: compute responsibilities
        resp = compute_responsibilities(data, weights, means, covariances)

        # M-step
        # Compute the total responsibility assigned to each cluster, which wil
l be useful when
        # implementing M-steps below. In the lectures this is called N^{soft}
        counts = compute_soft_counts(resp)

        # Update the weight for cluster k using the M-step update rule for the
 cluster weight, \hat{\pi}_k.
        # YOUR CODE HERE
        weights = compute_weights(counts)

        # Update means for cluster k using the M-step update rule for the mean
 variables.
        # This will assign the variable means[k] to be our estimate for \hat{\
mu}_k.
        # YOUR CODE HERE
        means = compute_means(data, resp, counts)

        # Update covariances for cluster k using the M-step update rule for co
variance variables.
        # This will assign the variable covariances[k] to be the estimate for
\hat{\Sigma}_k.
        # YOUR CODE HERE
        covariances = compute_covariances(data, resp, counts, means)

        # Compute the loglikelihood at this iteration
        # YOUR CODE HERE
        ll_latest = loglikelihood(data, weights, means, covariances)
        ll_trace.append(ll_latest)

        # Check for convergence in log-likelihood and store
        if (ll_latest - ll) < thresh and ll_latest > -np.inf:
            break
        ll = ll_latest

    if it % 5 != 0:
```

Testing the implementation on the simulated data

To help us develop and test our implementation, we will generate some observations from a mixture of Gaussians and then run our EM algorithm to discover the mixture components. We'll begin with a function to generate the data, and a quick plot to visualize its output for a 2-dimensional mixture of three Gaussians.

Now we will create a function to generate data from a mixture of Gaussians model.

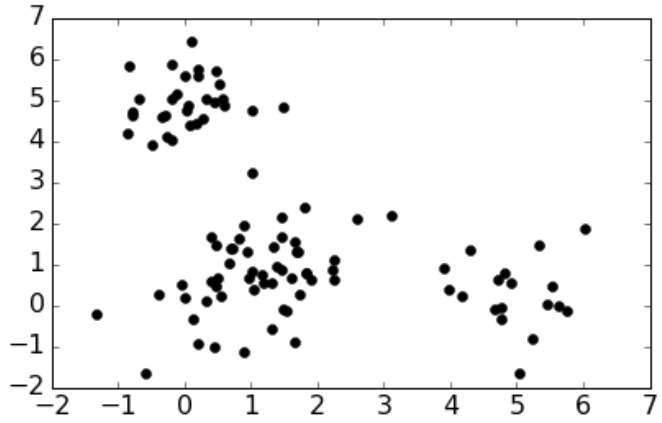
In [19]:
```
def generate_MoG_data(num_data, means, covariances, weights):
    """ Creates a list of data points """
    num_clusters = len(weights)
    data = []
    for i in range(num_data):
        #  Use np.random.choice and weights to pick a cluster id greater than
or equal to 0 and less than num_clusters.
        k = np.random.choice(len(weights), 1, p=weights)[0]

        # Use np.random.multivariate_normal to create data from this cluster
        x = np.random.multivariate_normal(means[k], covariances[k])

        data.append(x)
    return data
```

After specifying a particular set of clusters (so that the results are reproducible across assignments), we use the above function to generate a dataset.

In [20]:
```
# Model parameters
init_means = [
    [5, 0], # mean of cluster 1
    [1, 1], # mean of cluster 2
    [0, 5]  # mean of cluster 3
]
init_covariances = [
    [[.5, 0.], [0, .5]], # covariance of cluster 1
    [[.92, .38], [.38, .91]], # covariance of cluster 2
    [[.5, 0.], [0, .5]]  # covariance of cluster 3
]
init_weights = [1/4., 1/2., 1/4.]  # weights of each cluster

# Generate data
np.random.seed(4)
data = generate_MoG_data(100, init_means, init_covariances, init_weights)
```

Now plot the data you created above. The plot should be a scatterplot with 100 points that appear to roughly fall into three clusters.

In [21]:
```
plt.figure()
d = np.vstack(data)
plt.plot(d[:,0], d[:,1],'ko')
plt.rcParams.update({'font.size':16})
plt.tight_layout()
```



Now we'll fit a mixture of Gaussians to this data using our implementation of the EM algorithm. As with k-means, it is important to ask how we obtain an initial configuration of mixing weights and component parameters. In this simple case, we'll take three random points to be the initial cluster means, use the empirical covariance of the data to be the initial covariance in each cluster (a clear overestimate), and set the initial mixing weights to be uniform across clusters.

In [22]:
```
np.random.seed(4)

# Initialization of parameters
chosen = np.random.choice(len(data), 3, replace=False)
initial_means = [data[x] for x in chosen]
initial_covs = [np.cov(data, rowvar=0)] * 3
initial_weights = [1/3.] * 3

# Run EM
results = EM(data, initial_means, initial_covs, initial_weights)
```

```
Iteration 0
Iteration 5
Iteration 10
Iteration 15
Iteration 20
Iteration 22
```

Note. Like k-means, EM is prone to converging to a local optimum. In practice, you may want to run EM multiple times with different random initialization. We have omitted multiple restarts to keep the assignment reasonably short. For the purpose of this assignment, we assign a particular random seed (`seed=4`) to ensure consistent results among the students.

Checkpoint. For this particular example, the EM algorithm is expected to terminate in 23 iterations. That is, the last line of the log should say "Iteration 22". If your function stopped too early or too late, you should re-visit your code.

Our algorithm returns a dictionary with five elements:

- 'loglik': a record of the log likelihood at each iteration
- 'resp': the final responsibility matrix
- 'means': a list of K means
- 'covs': a list of K covariance matrices
- 'weights': the weights corresponding to each model component

Quiz Question: What is the weight that EM assigns to the first component after running the above codeblock?

In [24]:
```
results['weights']   # Your code here\
```

Out[24]:
```
[0.30071023006098241, 0.17993710074247016, 0.51935266919654743]
```

Quiz Question: Using the same set of results, obtain the mean that EM assigns the second component. What is the mean in the first dimension?

In [25]:
```
# Your code here
results['means']
```

Out[25]:
```
[array([ 0.02138285,  4.947729  ]),
 array([ 4.94239235,  0.31365311]),
 array([ 1.08181125,  0.73903508])]
```

Quiz Question: Using the same set of results, obtain the covariance that EM assigns the third component. What is the variance in the first dimension?

In [27]:
```
# Your code here
results['covs'][2]
```

Out[27]:
```
array([[ 0.67114992,  0.33058965],
       [ 0.33058965,  0.90429724]])
```

Plot progress of parameters

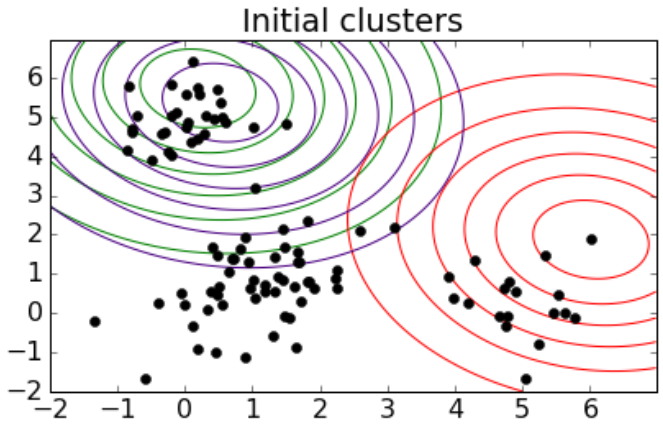
One useful feature of testing our implementation on low-dimensional simulated data is that we can easily visualize the results.

We will use the following `plot_contours` function to visualize the Gaussian components over the data at three different points in the algorithm's execution:

1. At initialization (using initial_mu, initial_cov, and initial_weights)
2. After running the algorithm to completion
3. After just 12 iterations (using parameters estimates returned when setting `maxiter=12`)

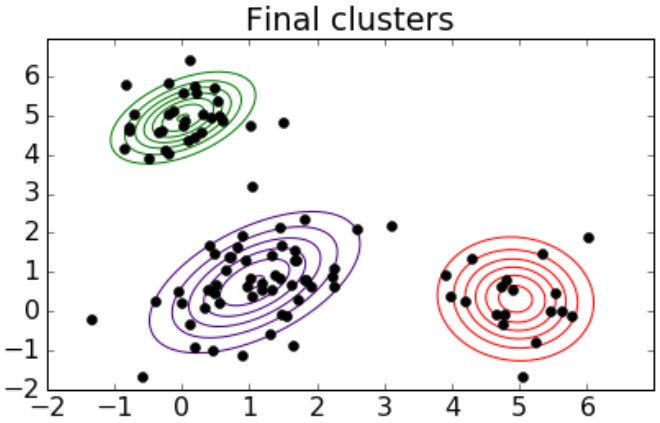
In [28]:
```
import matplotlib.mlab as mlab
def plot_contours(data, means, covs, title):
    plt.figure()
    plt.plot([x[0] for x in data], [y[1] for y in data],'ko') # data

    delta = 0.025
    k = len(means)
    x = np.arange(-2.0, 7.0, delta)
    y = np.arange(-2.0, 7.0, delta)
    X, Y = np.meshgrid(x, y)
    col = ['green', 'red', 'indigo']
    for i in range(k):
        mean = means[i]
        cov = covs[i]
        sigmax = np.sqrt(cov[0][0])
        sigmay = np.sqrt(cov[1][1])
        sigmaxy = cov[0][1]/(sigmax*sigmay)
        Z = mlab.bivariate_normal(X, Y, sigmax, sigmay, mean[0], mean[1], sigm
axy)
        plt.contour(X, Y, Z, colors = col[i])
        plt.title(title)
    plt.rcParams.update({'font.size':16})
    plt.tight_layout()
```

In [29]:
```
# Parameters after initialization
plot_contours(data, initial_means, initial_covs, 'Initial clusters')
```



In [30]:
```
# Parameters after running EM to convergence
results = EM(data, initial_means, initial_covs, initial_weights)
plot_contours(data, results['means'], results['covs'], 'Final clusters')
```

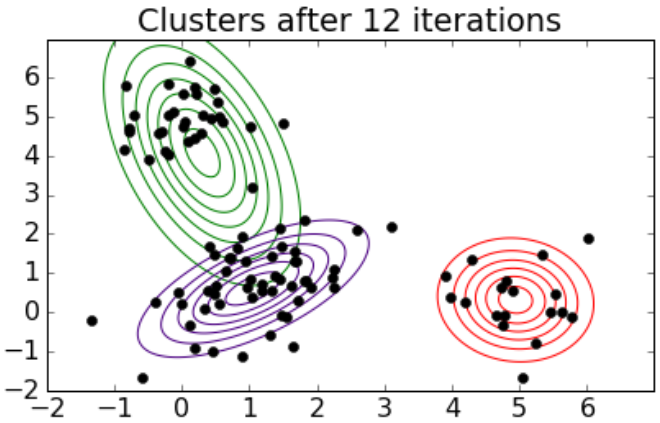
```
Iteration 0
Iteration 5
Iteration 10
Iteration 15
Iteration 20
Iteration 22
```



Fill in the following code block to visualize the set of parameters we get after running EM for 12 iterations.

In [31]:
```
# YOUR CODE HERE
results = EM(data, initial_means, initial_covs, initial_weights, maxiter = 12)

plot_contours(data, results['means'], results['covs'], 'Clusters after 12 iter
ations')
```

```
Iteration 0
Iteration 5
Iteration 10
Iteration 11
```



Quiz Question: Plot the loglikelihood that is observed at each iteration. Is the loglikelihood plot monotonically increasing, monotonically decreasing, or neither [multiple choice]?

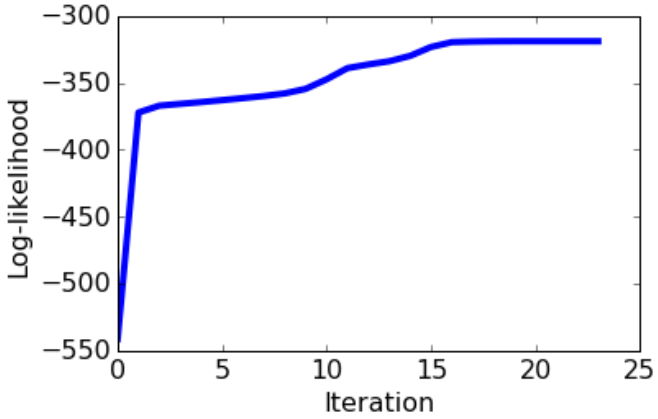
In [33]:
```
results['loglik']
```

Out[33]:
```
[-541.31612480366596,
 -372.13559278659284,
 -366.99356972063583,
 -365.65991992885012,
 -364.33486476974326,
 -362.87960460172394,
 -361.37504027778317,
 -359.78422255406787,
 -357.83486581694115,
 -354.44294078381785,
 -347.33757513783422,
 -338.95003191349019,
 -336.19269736896894]
```

In [34]:
```
results = EM(data, initial_means, initial_covs, initial_weights)

# YOUR CODE HERE
loglikelihoods = results['loglik']
```

```
Iteration 0
Iteration 5
Iteration 10
Iteration 15
Iteration 20
Iteration 22
```

In [35]:
```
plt.plot(range(len(loglikelihoods)), loglikelihoods, linewidth=4)
plt.xlabel('Iteration')
plt.ylabel('Log-likelihood')
plt.rcParams.update({'font.size':16})
plt.tight_layout()
```

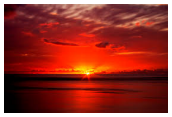


Fitting a Gaussian mixture model for image data

Now that we're confident in our implementation of the EM algorithm, we'll apply it to cluster some more interesting data. In particular, we have a set of images that come from four categories: sunsets, rivers, trees and forests, and cloudy skies. For each image we are given the average intensity of its red, green, and blue pixels, so we have a 3-dimensional representation of our data. Our goal is to find a good clustering of these images using our EM implementation; ideally our algorithm would find clusters that roughly correspond to the four image categories.

To begin with, we'll take a look at the data and get it in a form suitable for input to our algorithm. The data are provided in SFrame format:

In [36]:
```
images = gl.SFrame('images.sf')
gl.canvas.set_target('ipynb')
import array
images['rgb'] = images.pack_columns(['red', 'green', 'blue'])['X4']
images.show()
```

path	image	folder	red
dtype: **str**	dtype: **Image**	dtype: **str**	dtype:
num_unique (est.): **1 324**	First 4 images:	num_unique (est.): **4**	num_unique
num_undefined: **0**		num_undefined: **0**	num_undefi
frequent items:		frequent items:	min:
/data/coursera/ima ...		sunsets	max:
/data/coursera/ima ...		rivers	median:
/data/coursera/ima ...		trees_and_forest	mean:
/data/coursera/ima ...		cloudy_sky	std:
/data/coursera/ima ...			distribution
/data/coursera/ima ...			
/data/coursera/ima ...			
/data/coursera/ima ...			
/data/coursera/ima ...			
/data/coursera/ima ...			
/data/coursera/ima ...			

We need to come up with initial estimates for the mixture weights and component parameters. Let's take three images to be our initial cluster centers, and let's initialize the covariance matrix of each cluster to be diagonal with each element equal to the sample variance from the full data. As in our test on simulated data, we'll start by assuming each mixture component has equal weight.

This may take a few minutes to run.

In [37]:
```
np.random.seed(1)

# Initalize parameters
init_means = [images['rgb'][x] for x in np.random.choice(len(images), 4, repla
ce=False)]
cov = np.diag([images['red'].var(), images['green'].var(), images['blue'].var(
)])
init_covariances = [cov, cov, cov, cov]
init_weights = [1/4., 1/4., 1/4., 1/4.]

# Convert rgb data to numpy arrays
img_data = [np.array(i) for i in images['rgb']]

# Run our EM algorithm on the image data using the above initializations.
# This should converge in about 125 iterations
out = EM(img_data, init_means, init_covariances, init_weights)
```

```
Iteration 0
Iteration 5
Iteration 10
Iteration 15
Iteration 20
Iteration 25
Iteration 30
Iteration 35
Iteration 40
Iteration 45
Iteration 50
Iteration 55
Iteration 60
Iteration 65
Iteration 70
Iteration 75
Iteration 80
Iteration 85
Iteration 90
Iteration 95
Iteration 100
Iteration 105
Iteration 110
Iteration 115
Iteration 118
```

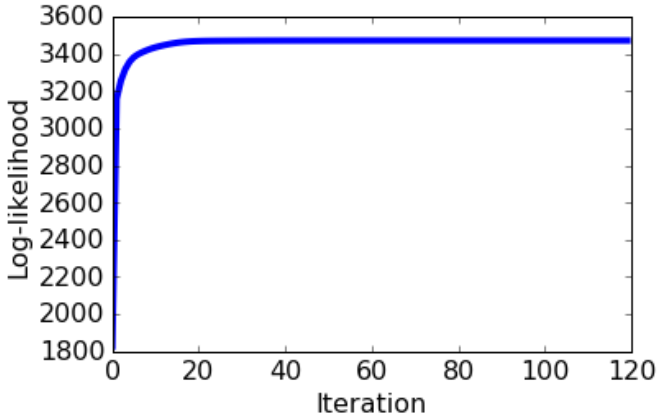
The following sections will evaluate the results by asking the following questions:

- **Convergence**: How did the log likelihood change across iterations? Did the algorithm achieve convergence?
- **Uncertainty**: How did cluster assignment and uncertainty evolve?
- **Interpretability**: Can we view some example images from each cluster? Do these clusters correspond to known image categories?

Evaluating convergence

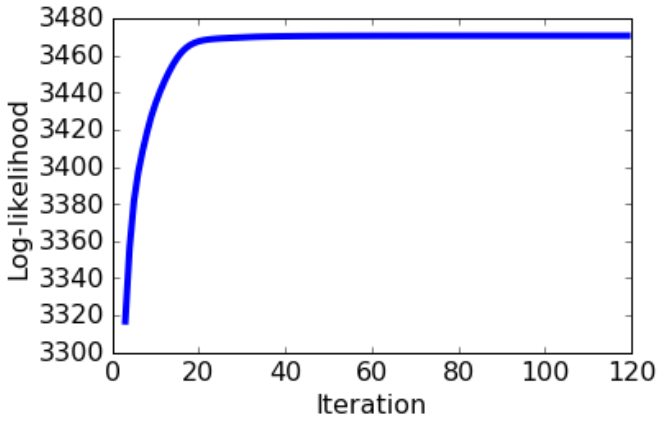
Let's start by plotting the log likelihood at each iteration - we know that the EM algorithm guarantees that the log likelihood can only increase (or stay the same) after each iteration, so if our implementation is correct then we should see an increasing function.

In [38]:
```
ll = out['loglik']
plt.plot(range(len(ll)),ll,linewidth=4)
plt.xlabel('Iteration')
plt.ylabel('Log-likelihood')
plt.rcParams.update({'font.size':16})
plt.tight_layout()
```



The log likelihood increases so quickly on the first few iterations that we can barely see the plotted line. Let's plot the log likelihood after the first three iterations to get a clearer view of what's going on:

In [39]:
```
plt.figure()
plt.plot(range(3,len(ll)),ll[3:],linewidth=4)
plt.xlabel('Iteration')
plt.ylabel('Log-likelihood')
plt.rcParams.update({'font.size':16})
plt.tight_layout()
```



Evaluating uncertainty

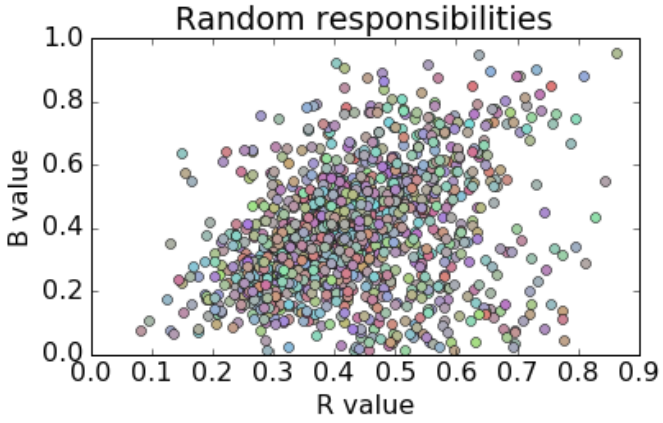
Next we'll explore the evolution of cluster assignment and uncertainty. Remember that the EM algorithm represents uncertainty about the cluster assignment of each data point through the responsibility matrix. Rather than making a 'hard' assignment of each data point to a single cluster, the algorithm computes the responsibility of each cluster for each data point, where the responsibility corresponds to our certainty that the observation came from that cluster.

We can track the evolution of the responsibilities across iterations to see how these 'soft' cluster assignments change as the algorithm fits the Gaussian mixture model to the data; one good way to do this is to plot the data and color each point according to its cluster responsibilities. Our data are three-dimensional, which can make visualization difficult, so to make things easier we will plot the data using only two dimensions, taking just the [R G], [G B] or [R B] values instead of the full [R G B] measurement for each observation.

In [40]:
```
import colorsys
def plot_responsibilities_in_RB(img, resp, title):
    N, K = resp.shape

    HSV_tuples = [(x*1.0/K, 0.5, 0.9) for x in range(K)]
    RGB_tuples = map(lambda x: colorsys.hsv_to_rgb(*x), HSV_tuples)

    R = img['red']
    B = img['blue']
    resp_by_img_int = [[resp[n][k] for k in range(K)] for n in range(N)]
    cols = [tuple(np.dot(resp_by_img_int[n], np.array(RGB_tuples))) for n in r
ange(N)]

    plt.figure()
    for n in range(len(R)):
        plt.plot(R[n], B[n], 'o', c=cols[n])
    plt.title(title)
    plt.xlabel('R value')
    plt.ylabel('B value')
    plt.rcParams.update({'font.size':16})
    plt.tight_layout()
```

To begin, we will visualize what happens when each data has random responsibilities.

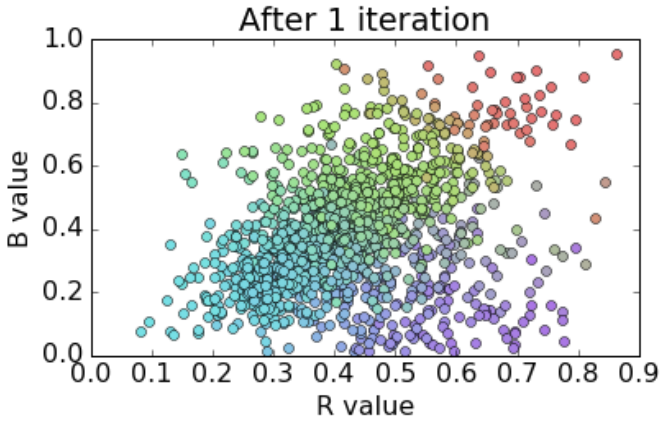
In [41]:
```
N, K = out['resp'].shape
random_resp = np.random.dirichlet(np.ones(K), N)
plot_responsibilities_in_RB(images, random_resp, 'Random responsibilities')
```



We now use the above plotting function to visualize the responsibilites after 1 iteration.

In [42]:
```
out = EM(img_data, init_means, init_covariances, init_weights, maxiter=1)
plot_responsibilities_in_RB(images, out['resp'], 'After 1 iteration')
```

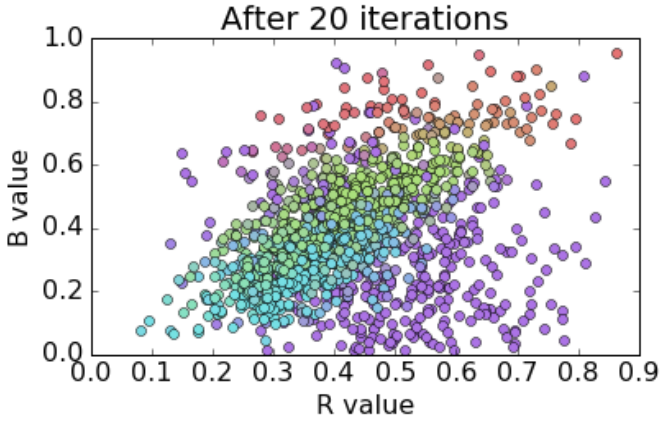
```
Iteration 0
```



We now use the above plotting function to visualize the responsibilites after 20 iterations. We will see there are fewer unique colors; this indicates that there is more certainty that each point belongs to one of the four components in the model.

In [43]:
```
out = EM(img_data, init_means, init_covariances, init_weights, maxiter=20)
plot_responsibilities_in_RB(images, out['resp'], 'After 20 iterations')
```

```
Iteration 0
Iteration 5
Iteration 10
Iteration 15
Iteration 19
```



Plotting the responsibilities over time in [R B] space shows a meaningful change in cluster assignments over the course of the algorithm's execution. While the clusters look significantly better organized at the end of the algorithm than they did at the start, it appears from our plot that they are still not very well separated. We note that this is due in part our decision to plot 3D data in a 2D space; everything that was separated along the G axis is now "squashed" down onto the flat [R B] plane. If we were to plot the data in full [R G B] space, then we would expect to see further separation of the final clusters. We'll explore the cluster interpretability more in the next section.

Interpreting each cluster

Let's dig into the clusters obtained from our EM implementation. Recall that our goal in this section is to cluster images based on their RGB values. We can evaluate the quality of our clustering by taking a look at a few images that 'belong' to each cluster. We hope to find that the clusters discovered by our EM algorithm correspond to different image categories - in this case, we know that our images came from four categories ('cloudy sky', 'rivers', 'sunsets', and 'trees and forests'), so we would expect to find that each component of our fitted mixture model roughly corresponds to one of these categories.

If we want to examine some example images from each cluster, we first need to consider how we can determine cluster assignments of the images from our algorithm output. This was easy with k-means - every data point had a 'hard' assignment to a single cluster, and all we had to do was find the cluster center closest to the data point of interest. Here, our clusters are described by probability distributions (specifically, Gaussians) rather than single points, and our model maintains some uncertainty about the cluster assignment of each observation.

One way to phrase the question of cluster assignment for mixture models is as follows: how do we calculate the distance of a point from a distribution? Note that simple Euclidean distance might not be appropriate since (non-scaled) Euclidean distance doesn't take direction into account. For example, if a Gaussian mixture component is very stretched in one direction but narrow in another, then a data point one unit away along the 'stretched' dimension has much higher probability (and so would be thought of as closer) than a data point one unit away along the 'narrow' dimension.

In fact, the correct distance metric to use in this case is known as Mahalanobis distance (https://en.wikipedia.org/wiki/Mahalanobis_distance). For a Gaussian distribution, this distance is proportional to the square root of the negative log likelihood. This makes sense intuitively - reducing the Mahalanobis distance of an observation from a cluster is equivalent to increasing that observation's probability according to the Gaussian that is used to represent the cluster. This also means that we can find the cluster assignment of an observation by taking the Gaussian component for which that observation scores highest. We'll use this fact to find the top examples that are 'closest' to each cluster.

Quiz Question: Calculate the likelihood (score) of the first image in our data set (`images[0]`) under each Gaussian component through a call to `multivariate_normal.pdf`. Given these values, what cluster assignment should we make for this image? Hint: don't forget to use the cluster weights.

Now we calculate cluster assignments for the entire image dataset using the result of running EM for 20 iterations above:

In [47]:
```
weights = out['weights']
means = out['means']
covariances = out['covs']
rgb = images['rgb']
N = len(images) # number of images
K = len(means) # number of clusters

assignments = [0]*N
probs = [0]*N

for i in range(N):
    # Compute the score of data point i under each Gaussian component:
    p = np.zeros(K)
    for k in range(K):
        p[k] = weights[k]*multivariate_normal.pdf(rgb[i], mean=means[k], cov=c
ovariances[k])

    # Compute assignments of each data point to a given cluster based on the a
bove scores:
    assignments[i] = np.argmax(p)

    # For data point i, store the corresponding score under this cluster assig
nment:
    probs[i] = np.max(p)

assignments = gl.SFrame({'assignments':assignments, 'probs':probs, 'image': im
ages['image']})
```

In [51]:
```
assignments['assignments'][0]
```

Out[51]: 3

We'll use the 'assignments' SFrame to find the top images from each cluster by sorting the datapoints within each cluster by their score under that cluster (stored in `probs`). We can plot the corresponding images in the original data using show().

Create a function that returns the top 5 images assigned to a given category in our data (HINT: use the GraphLab Create function `topk(column, k)` to find the k top values according to specified column in an SFrame).

In [52]:
```
def get_top_images(assignments, cluster, k=5):
    # YOUR CODE HERE
    images_in_cluster = assignments[assignments['assignments']==cluster]
    top_images = images_in_cluster.topk('probs', k)
    return top_images['image']
```

Use this function to show the top 5 images in each cluster.

In [53]:
```
gl.canvas.set_target('ipynb')
for component_id in range(4):
    get_top_images(assignments, component_id).show()
```

All 5 images in *<SArray>*

All 5 images in *<SArray>*

All 5 images in *<SArray>*

All 5 images in *<SArray>*

These look pretty good! Our algorithm seems to have done a good job overall at 'discovering' the four categories that from which our image data was drawn. It seems to have had the most difficulty in distinguishing between rivers and cloudy skies, probably due to the similar color profiles of images in these categories; if we wanted to achieve better performance on distinguishing between these categories, we might need a richer representation of our data than simply the average [R G B] values for each image.

Quiz Question: Which of the following images are *not* in the list of top 5 images in the first cluster?

In [ ]: