Clustering text data with Gaussian mixtures

In a previous assignment, we explored K-means clustering for a high-dimensional Wikipedia dataset. We can also model this data with a mixture of Gaussians, though with increasing dimension we run into several important problems associated with using a full covariance matrix for each component.

In this section, we will use an EM implementation to fit a Gaussian mixture model with **diagonal** covariances to a subset of the Wikipedia dataset.

If you are using GraphLab Create

An IPython Notebook has been provided below to you for this assignment. This notebook contains the instructions, quiz questions and partially-completed code for you to use as well as some cells to test your code.

- Download the Wikipedia people dataset in SFrame format: people_wiki.gl.zip
- Download the companion IPython notebook: 4_em-with-text-data_blank.ipynb
- Download a collection of helper functions: em_utilities.py
- Save all the files in the same directory (where you are calling IPython notebook from) and unzip the data file.

Open the companion IPython notebook and follow the instructions in the notebook. The instructions below do not apply to users of GraphLab Create.

If you are not using GraphLab Create

It is possible to complete this assignment without using GraphLab Create. The instructions below are geared towards Python users, but you are free to adapt them to your specific environment.

Disclaimer. We have tested the assessment using the standard Python installation (with access to scikit-learn). However, the assessment may not be compatible with other tools (e.g. Matlab, R).

Download the dataset

- Download the Wikipedia people dataset in SFrame format: people_wiki.gl.zip. (Those experimenting with other tools, get people_wiki.csv.zip instead.)

- Download the mapping between words and integer indices: 4_map_index_to_word.gl.zip (or alternatively, 4_map_index_to_word.json.zip)
- Download the pre-processed set of TF-IDF scores: 4_tf_idf.npz

Overview

In a previous assignment, we explored k-means clustering for a high-dimensional Wikipedia dataset. We can also model this data with a mixture of Gaussians, though with increasing dimension we run into two important issues associated with using a full covariance matrix for each component.

- Computational cost becomes prohibitive in high dimensions: score calculations have complexity cubic in the number of dimensions M if the Gaussian has a full covariance matrix.
- A model with many parameters require more data: bserve that a full covariance matrix for an M-dimensional Gaussian will have M(M+1)/2 parameters to fit. With the number of parameters growing roughly as the square of the dimension, it may quickly become impossible to find a sufficient amount of data to make good inferences.

Both of these issues are avoided if we require the covariance matrix of each component to be diagonal, as then it has only M parameters to fit and the score computation decomposes into M univariate score calculations. Recall from the lecture that the M-step for the full covariance is:

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

Note that this is a square matrix with M rows and M columns, and the above equation implies that the (v, w) element is computed by

$$\hat{\Sigma}_{k,v,w} = \frac{1}{N_k^{soft}} \sum_{i=1}^{N} r_{ik}(x_{iv} - \hat{\mu}_{kv})(x_{iw} - \hat{\mu}_{kw})$$

When we assume that this is a diagonal matrix, then non-diagonal elements are assumed to be zero and we only need to compute each of the M elements along the diagonal independently using the following equation.

$$\hat{\sigma}_{k,v}^2 = \hat{\Sigma}_{k,v,v} = \frac{1}{N_k^{soft}} \sum_{i=1}^{N} r_{ik}(x_{iv} - \hat{\mu}_{kv})^2$$

In this section, we will use an EM implementation to fit a Gaussian mixture model with **diagonal** covariances to a subset of the Wikipedia dataset. The implementation uses the above equation to compute each variance term.

We'll begin by importing the dataset and coming up with a useful representation for each article. After running our algorithm on the data, we will explore the output to see whether we can give a meaningful interpretation to the fitted parameters in our model.

Import packages

```
import sframe                                          # see below for install instruction
import numpy as np
from scipy.sparse import csr_matrix
from scipy.sparse import spdiags
from scipy.stats import multivariate_normal
from copy import deepcopy
from sklearn.metrics import pairwise_distances
from sklearn.preprocessing import normalize
```

About SFrame. SFrame is a dataframe library Dato has released free-of-charge. Its source code is availablehere. You may install SFrame via pip:

```
pip install --upgrade sframe
```

Load Wikipedia data and extract TF-IDF features

Load Wikipedia data and transform each of the first 5000 document into a TF-IDF representation.

```
wiki = sframe.SFrame('people_wiki.gl/').head(5000)
```

As in the previous assignment, we extract the TF-IDF vector of each document.

For your convenience, we extracted the TF-IDF vectors from the dataset. The vectors are packaged in a sparse matrix, where the i-th row gives the TF-IDF vectors for the i-th document. Each column corresponds to a unique word appearing in the dataset.

To load in the TF-IDF vectors, run

```
def load_sparse_csr(filename):
    loader = np.load(filename)
    data = loader['data']
    indices = loader['indices']
    indptr = loader['indptr']
    shape = loader['shape']

    return csr_matrix( (data, indices, indptr), shape)

tf_idf = load_sparse_csr('4_tf_idf.npz')  # NOT people_wiki_tf_idf.npz
map_index_to_word = sframe.SFrame('4_map_index_to_word.gl/')  # NOT
    people_wiki_map_index_to_word.gl
```

As in the previous assignment, we will normalize each document's TF-IDF vector to be a unit vector.

```
tf_idf = normalize(tf_idf)
```

(Optional) Extracting TF-IDF vectors yourself. We provide the pre-computed TF-IDF vectors to minimize potential compatibility issues. You are free to experiment with other tools to compute the TF-IDF vectors yourself. A good place to start is sklearn.TfidfVectorizer. Note. Due to variations in tokenization and other factors, your TF-IDF vectors may differ from the ones we provide. For the purpose the assessment, we ask you to use the vectors from 4_tf_idf.npz.

EM in high dimensions

EM for high-dimensional data requires some special treatment:

- E step and M step must be vectorized as much as possible, as explicit loops are dreadfully slow in Python.
- All operations must be cast in terms of sparse matrix operations, to take advantage of computational savings enabled by sparsity of data.
- Initially, some words may be entirely absent from a cluster, causing the M step to produce zero mean and variance for those words. This means any data point with one of those words will have 0 probability of being assigned to that cluster since the cluster allows for no variability (0 variance) around that count being 0 (0 mean). Since there is a small chance for those words to later appear in the cluster, we instead assign a small positive variance (~1e-10). Doing so also prevents numerical overflow.

Due to complexity in implementation, we provide the complete implementation here. You are expected to answer some quiz questions using the results of clustering.

Log probability function for diagonal covariance Gaussian.

```
def diag(array):
    n = len(array)
    return spdiags(array, 0, n, n)

def logpdf_diagonal_gaussian(x, mean, cov):
    '''
    Compute logpdf of a multivariate Gaussian distribution with diagonal covariance at a given
        point x.
    A multivariate Gaussian distribution with a diagonal covariance is equivalent
    to a collection of independent Gaussian random variables.

    x should be a sparse matrix. The logpdf will be computed for each row of x.
    mean and cov should be given as 1D numpy arrays
    mean[i] : mean of i-th variable
    cov[i] : variance of i-th variable'''

    n = x.shape[0]
    dim = x.shape[1]
    assert(dim == len(mean) and dim == len(cov))

    # multiply each i-th column of x by (1/(2*sigma_i)), where sigma_i is sqrt of variance of i
        -th variable.
    scaled_x = x.dot( diag(1./(2*np.sqrt(cov))) )
    # multiply each i-th entry of mean by (1/(2*sigma_i))
    scaled_mean = mean/(2*np.sqrt(cov))

    # sum of pairwise squared Eulidean distances gives SUM[(x_i - mean_i)^2/(2*sigma_i^2)]
    return -np.sum(np.log(np.sqrt(2*np.pi*cov))) - pairwise_distances(scaled_x, [scaled_mean],
        'euclidean').flatten()**2
```

EM algorithm for sparse data.

```
def log_sum_exp(x, axis):
    '''Compute the log of a sum of exponentials'''
    x_max = np.max(x, axis=axis)
    if axis == 1:
        return x_max + np.log( np.sum(np.exp(x-x_max[:,np.newaxis]), axis=1) )
    else:
        return x_max + np.log( np.sum(np.exp(x-x_max), axis=0) )

def EM_for_high_dimension(data, means, covs, weights, cov_smoothing=1e-5, maxiter=int(1e3),
    thresh=1e-4, verbose=False):
    # cov_smoothing: specifies the default variance assigned to absent features in a cluster.
    #                If we were to assign zero variances to absent features, we would be
    overconfient,
    #                as we hastily conclude that those featurese would NEVER appear in the
    cluster.
    #                We'd like to leave a little bit of possibility for absent features to show
    up later.
    n = data.shape[0]
    dim = data.shape[1]
    mu = deepcopy(means)
    Sigma = deepcopy(covs)
    K = len(mu)
    weights = np.array(weights)

    ll = None
    ll_trace = []

    for i in range(maxiter):
        # E-step: compute responsibilities
        logresp = np.zeros((n,K))
        for k in xrange(K):
            logresp[:,k] = np.log(weights[k]) + logpdf_diagonal_gaussian(data, mu[k], Sigma[k])
        ll_new = np.sum(log_sum_exp(logresp, axis=1))
        if verbose:
            print(ll_new)
        logresp -= np.vstack(log_sum_exp(logresp, axis=1))
        resp = np.exp(logresp)
        counts = np.sum(resp, axis=0)

        # M-step: update weights, means, covariances
        weights = counts / np.sum(counts)
        for k in range(K):
            mu[k] = (diag(resp[:,k]).dot(data)).sum(axis=0)/counts[k]
            mu[k] = mu[k].A1

```

```
            Sigma[k] = diag(resp[:,k]).dot( data.power(2)-2*data.dot(diag(mu[k])) ).sum(axis=0)
                \
                       + (mu[k]**2)*counts[k]
            Sigma[k] = Sigma[k].A1 / counts[k] + cov_smoothing*np.ones(dim)

        # check for convergence in log-likelihood
        ll_trace.append(ll_new)
        if ll is not None and (ll_new-ll) < thresh and ll_new > -np.inf:
            ll = ll_new
            break
        else:
            ll = ll_new

    out = {'weights':weights,'means':mu,'covs':Sigma,'loglik':ll_trace,'resp':resp}

    return out
```

Initializing mean parameters using k-means. Recall from the lectures that EM for Gaussian mixtures is very sensitive to the choice of initial means. With a bad initial set of means, EM may produce clusters that span a large area and are mostly overlapping. To eliminate such bad outcomes, we first produce a suitable set of initial means by using the cluster centers from running k-means. That is, we first run k-means and then take the final set of means from the converged solution as the initial means in our EM algorithm.

```
from sklearn.cluster import KMeans

np.random.seed(5)
num_clusters = 25

# Use scikit-learn's k-means to simplify workflow
kmeans_model = KMeans(n_clusters=num_clusters, n_init=5, max_iter=400, random_state=1, n_jobs=-1
    )
kmeans_model.fit(tf_idf)
centroids, cluster_assignment = kmeans_model.cluster_centers_, kmeans_model.labels_

means = [centroid for centroid in centroids]
```

Initializing cluster weights. We will initialize each cluster weight to be the proportion of documents assigned to that cluster by k-means above.

```
num_docs = tf_idf.shape[0]
weights = []
for i in xrange(num_clusters):
    # Compute the number of data points assigned to cluster i:
    num_assigned = ... # YOUR CODE HERE
    w = float(num_assigned) / num_docs
    weights.append(w)**Initializing covariances.** To initialize our covariance parameters, we
        compute $\hat{\sigma}_{k, j}^2 = \sum_{i=1}^{N}(x_{i,j} - \hat{\mu}_{k, j})^2$ for each
        feature $j$.  For features with really tiny variances, we assign 1e-8 instead to prevent
        numerical instability. We do this computation in a vectorized fashion in the following
        code block.
```

Initializing covariances. To initialize our covariance parameters, we compute $\hat{\sigma}_{k,j}^2 = \sum_{i=1}^{N}(x_{i,j} - \hat{\mu}_{k,j})^2$ for each feature j. For features with really tiny variances, we assign 1e-8 instead to prevent numerical instability. We do this computation in a vectorized fashion in the following code block.

```
covs = []
for i in xrange(num_clusters):
    member_rows = tf_idf[cluster_assignment==i]
    cov = (member_rows.power(2) - 2*member_rows.dot(diag(means[i]))).sum(axis=0).A1 / member_rows
        .shape[0] \
          + means[i]**2
    cov[cov < 1e-8] = 1e-8
    covs.append(cov)
```

Running EM. Now that we have initialized all of our parameters, run EM.

```
out = EM_for_high_dimension(tf_idf, means, covs, weights, cov_smoothing=1e-10)
print out['loglik'] # print history of log-likelihood over time
```

Interpret clusters

In contrast to k-means, EM is able to explicitly model clusters of varying sizes and proportions. The relative magnitude of variances in the word dimensions tell us much about the nature of the clusters.

Write yourself a cluster visualizer as follows. Examining each cluster's mean vector, list the 5 words with the largest mean values (5 most common words in the cluster). For each word, also include the associated variance parameter (diagonal element of the covariance matrix).

A sample output may be:

```
============================================================
Cluster 0: Largest mean parameters in cluster

Word        Mean        Variance
football    1.08e-01    8.64e-03
season      5.80e-02    2.93e-03
club        4.48e-02    1.99e-03
league      3.94e-02    1.08e-03
played      3.83e-02    8.45e-04
...
```

```
# Fill in the blanks
def visualize_EM_clusters(tf_idf, means, covs, map_index_to_word):
    print('')
    print('============================================================')

    num_clusters = len(means)
    for c in xrange(num_clusters):
        print('Cluster {0:d}: Largest mean parameters in cluster '.format(c))
        print('\n{0: <12}{1: <12}{2: <12}'.format('Word', 'Mean', 'Variance'))

        # The k'th element of sorted_word_ids should be the index of the word
        # that has the k'th-largest value in the cluster mean. Hint: Use np.argsort().
        sorted_word_ids = ...  # YOUR CODE HERE

        for i in sorted_word_ids[:5]:
            print '{0: <12}{1:<10.2e}{2:10.2e}'.format(map_index_to_word['category'][i],
                                                       means[c][i],
                                                       covs[c][i])
        print '\n============================================================Quiz Question. Select all
            the topics that have a cluster in the model created above. [multiple choice]====='

```

Quiz Question. Select all the topics that have a cluster in the model created above. [multiple choice]

```
'''By EM'''
visualize_EM_clusters(tf_idf, out['means'], out['covs'], map_index_to_word)
```

Comparing to random initialization

Create variables for randomly initializing the EM algorithm. Complete the following code block.

```
np.random.seed(5)
num_clusters = len(means)
num_docs, num_words = tf_idf.shape

random_means = []
random_covs = []
random_weights = []

for k in range(num_clusters):

    # Create a numpy array of length num_words with random normally distributed values.
    # Use the standard univariate normal distribution (mean 0, variance 1).
    # YOUR CODE HERE
    mean = ...

    # Create a numpy array of length num_words with random values uniformly distributed between
        1 and 5.
    # YOUR CODE HERE
    cov = ...

    # Initially give each cluster equal weight.
    # YOUR CODE HERE
    weight = ...

    random_means.append(mean)
    random_covs.append(cov)
    random_weights.append(weight)
```

Quiz Question: Try fitting EM with the random initial parameters you created above. (Use cov_smoothing=1e-5.) Store the result to out_random_init. What is the final loglikelihood that the algorithm converges to?

Quiz Question: Is the final loglikelihood larger or smaller than the final loglikelihood we obtained above when initializing EM with the results from running k-means?

Quiz Question: For the above model, out_random_init, use the visualize_EM_clustersmethod you created above. Are the clusters more or less interpretable than the ones found after initializing using k-means?

Takeaway

In this assignment we were able to apply the EM algorithm to a mixture of Gaussians model of text data. This was made possible by modifying the model to assume a diagonal covariance for each cluster, and by modifying the implementation to use a sparse matrix representation. In the second part you explored the role of k-means initialization on the convergence of the model as well as the interpretability of the clusters.