CS452, Spring 2018, Problem Set # 4
Ashesi University

April 5, 2018

Due: April 19th 2018, 11:59pm

This problem set requires you to provide written answers to a few technical questions. In addition, you will need to implement some learning algorithms in Python and apply them to the enclosed data sets. The questions requiring a written answer are denoted with **W**, while the Python programming questions are marked with **P**.

Both the written answers as well as the code must be submitted electronically via Courseware.

For the written answers, you must provide a single PDF containing all your answers. Upload the PDF files trough the appropriate link in Courseware. The PDF file can be generated using editing software (e.g., Latex) or can be a scanned copy of your hand-written answers. If you opt for scanned copies, please make sure to use a scanner, *not* your phone camera. Familiarize yourself with the process of scanning documents a few days before the due date. Late submissions due to scanning problems will be treated and penalized as any other form of late submission, as per the rules stated on the Syllabus. Also, make sure your scanned submission is readable. If we cannot read it clearly, you will receive 0 points for that problem.

Your Python code must also be submitted via Courseware. We have provided Python function files with an empty body for all the functions that you need to implement. Each file specifies in detail the input and the output arguments of the function. A few of the functions include some commands to help you get started with the implementation. The function stubs include also *import* commands for all the libraries that are needed for your implementation (e.g., numpy, math). Do not add any other import command, i.e., do not use any other libraries except those already specified in the function stubs. If in doubt, please ask. We have also provided scripts that check the sizes of the inputs and outputs of your functions. Do not modify these scripts. In order to allow us to test your code, it is imperative that you do not rename the files or modify the syntax of the functions. Make sure to include all the files necessary to run your code. **If we are unable to run one of your functions, you will receive 0 points for that question.** Please place all your files (data, .png files automatically generated by the provided scripts, and function files) in a single folder (**without subfolders**). Zip up the entire folder, and upload the compressed file to Courseware. Note that we provided **two** separate links for your homework submission: one for your written answers, one for your code.

1. **[40 points]** A grayscale image can be viewed as a 2D matrix containing at entry (i, j) the intensity value of pixel (i, j). In this problem you are going to implement a simple technique for image compression, called *vector quantization*. It consists of four operations:
   1. splitting the image into small non-overlapping tiles of size, say, 8 by 8 pixels;
   2. interpreting this set of tiles as a training set of 64-dimensional input examples (each vector component being a pixel);
   3. running a clustering algorithm (e.g., K-means) on this training set;
   4. coding each tile by a code that represents the nearest cluster centroid to the tile.

   (a) [**P**, 15 points] Implement vector quantization using K-means and run it on the enclosed image "dartmouthhall.jpg". We have provided for you the function `q1_splitimgintiles.py` which splits the image in non-overlapping tiles of size 8 by 8. In order to implement vector quantization you need to fill the body of the following functions:

   - [3 points] `q1_dist2.py` calculates the *squared* Euclidean distance between two sets of points. Note that it is possible to implement this function without using any `for` or `while` loop, simply in terms of basic matrix operations. Do not call `sqrt` (which is expensive), instead output the *squared* Euclidean distance.
   - [3 points] `q1_kmeans_select_seeds.py` selects K examples that will be used as initial centroids for the K-means algorithm that you will later run. We have already implemented for you random initialization of the cluster centroids inside this function. However, in class we stressed that often better results can be obtained by choosing the initial centroids incrementally, one at a time, based on some criterion aimed at yielding a diverse set. Here we ask you to choose the i-th centroid to be the example that is furthest away from the already selected $i-1$ centroids, i.e., the example whose *minimum* distance to the first $i-1$ centroids is *maximum*. In order to make the implementation deterministic, choose the first centroid to be the first example, i.e., `X`[0,:].
   - [6 points] `q1_kmeans.py`[1] performs K-means clustering on an input set of examples (in our case the image tiles), given a set of K initial centroids. Stop K-means when the difference in distortions (i.e., the sum of squared difference errors) between two consecutive iterations is less than 10^{-6}.
   - [3 points] `q1_reconstructimgfromVQ.py` reconstructs the "compressed" image. Given the centroids and the cluster labels computed from K-means, it synthesizes an image by setting each tile to its closest centroid. We suggest that you study thoroughly the function `q1_splitimgintiles.py` that we have given to you, as this should guide your implementation of `q1_reconstructimgfromVQ.py`. In particular note that `q1_splitimgintiles.py` stores the tiles in column order, i.e., top to bottom, left to right. You should reuse the tiles in the same order to synthesize the image.

   Now you should be able to run the script `q1a.py` which will perform vector quantization using $K \in \{2, 4, 8\}$ clusters and show the results of the image compression and decompression. The script will initialize the K-means using the diverse-set strategy that

[1] Obviously, for this problem you cannot use existing K-means functions from Python libraries; you must code the method yourself.

you have coded. If you change the initialization method to "random", you would see that the reconstruction error tends to be higher as random centroids provide an inferior initialization for K-means.

(b) [**P**, 18 points] Starting from the results obtained with K-means using $K = 4$, train a Mixture of Gaussians with 4 components via the EM algorithm. You need to complete the body of the following functions:

- [2 points] `q1_gmminit.py` initializes the GMM model, given a (hard) assignment of the input examples to K clusters (e.g., computed via K-means). This function must essentially implement a single M-step using the hard associations to centroids in lieu of the component posterior probabilities.
- [5 points] `q1_logprobgauss.py` calculates the log-probability for an input example under a multivariate Gaussian model, i.e., it computes $\log(p(x; \mu, \Sigma)) = \log(\mathcal{N}(x; \mu, \Sigma))$. We ask you to do the computation in log space to avoid numerical problems. Indeed, in practical scenarios where the dimensionality of the data is high (64 for this problem), the value of a Gaussian drops very, very fast with the distance of the example from the mean causing numerical issues in the computations. The point of using log probabilities is that it allows you to decompose the product in the argument of the log function in a sum of logs. So when you have an expression like $log(a \cdot b \cdot c)$, you should expand it in your code as $log(a) + log(b) + log(c)$. The latter expression eliminates the numerical issues, while the former is prone to underflow problems. Similarly, if you have something like $log(a^b)$, you should compute it as $b \cdot log(a)$, which is much more numerically stable.
  Note that it will be sufficient to apply the Numpy `np.exp` function to the output value produced by your function in order to obtain the actual density value of the Gaussian.
- [5 points] `q1_GM_Expectation.py` executes the Expectation-step for the learning of a GMM. Your function should call `q1_logprobgauss.py`. Furthermore, remember to use the `np.exp` function to obtain the posterior probabilities.
- [3 points] `q1_GM_Maximization.py` executes the Maximization-step for the learning of a GMM. Again, you will need to use `q1_logprobgauss.py`.
- [3 points] `q1_GaussianMixture.py` learns the GMM by using the Expectation Maximization (EM) algorithm. In this function you need to call your previously implemented functions `q1_GM_Expectation.py` and `q1_GM_Maximization.py`.

After you have coded all these functions you should be able to run 10 iterations of EM training using the script `q1b.py`. The script initializes the clustering assignments using K-means, then it trains the GMM using this initialization.

(c) [**W**, 3 points] The script `q1b` performs vector quantization by replacing each tile with the mean of the Gaussian that yields the highest posterior probability for that example. However, if you have coded everything correctly you should notice that after we decompress the image, counterintuitively, the reconstruction error is higher than the one produced by the simple K-means that we used as initialization for EM! Can you explain this result?

(d) [**W**, 4 points] We asked you to implement your functions so that the script `q1b.py` can plot both the log-likelihood and the free energy computed after each E-step as functions

of the number of iterations. It also shows the log-likelihood and the free energy computed after each M-step. What is the relationship between the log-likelihood and the free energy after each E-step? And after each M step? Why? Please provide a *detailed* and *rigorous* explanation.