CS 189 Introduction to Machine Learning
Spring 2018 HW10

1 Getting Started

Read through this page carefully. You may typeset your homework in latex or submit neatly handwritten/scanned solutions. Please start each question on a new page. Deliverables:

1. Submit a PDF of your writeup, **with an appendix for your code**, to assignment on Gradescope, "HW10 Write-Up". If there are graphs, include those graphs in the correct sections. Do not simply reference your appendix.
2. If there is code, submit all code needed to reproduce your results, "HW10 Code".
3. If there is a test set, submit your test set evaluation results, "HW10 Test Set".

After you've submitted your homework, watch out for the self-grade form.

(a) Who else did you work with on this homework? In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

(b) Please copy the following statement and sign next to it. We just want to make it *extra* clear so that no one inadverdently cheats.

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up.

2 Regularized and Kernel k-Means

Recall that in k-means clustering we are attempting to minimize an objective defined as follows:

$$\min_{C_1,C_2,\ldots,C_k} \sum_{i=1}^{k} \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2, \text{ where}$$

$$\mu_i = \text{argmin}_{\mu_i \in \mathbb{R}^d} \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2 = \frac{1}{|C_i|} \sum_{x_j \in C_i} x_j, \quad i = 1, 2, \ldots, k.$$

The samples are $\{x_1, \ldots, x_n\}$, where $x_j \in \mathbb{R}^d$, and C_i is the set of samples assigned to cluster i. Each sample is assigned to exactly one cluster, and clusters are non-empty.

(a) **What is the minimum value of the objective when $k = n$ (the number of clusters equals the number of samples)?**

(b) (Regularized k-means) Suppose we add a regularization term to the above objective. That is, the objective now becomes

$$\sum_{i=1}^{k} \left(\lambda \|\mu_i\|_2^2 + \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2 \right).$$

Show that the optimum of

$$\min_{\mu_i \in \mathbb{R}^d} \lambda \|\mu_i\|_2^2 + \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2$$

is obtained at $\mu_i = \frac{1}{|C_i| + \lambda} \sum_{x_j \in C_i} x_j$.

(c) Here is an example where we would want to regularize clusters. Suppose there are n students who live in a $\mathbb{R}^2$ Euclidean world and who wish to share rides efficiently to Berkeley for their final exam in CS189. The university permits k vehicles which may be used for shuttling students to the exam location. The students need to figure out k good locations to meet up. The students will then each *drive* to the closest meet up point and then the shuttles will deliver them to the exam location. Define x_j to be the location of student j, and let the exam location be at $(0, 0)$. Assume that we can drive as the crow flies, i.e. by taking the shortest paths between two points. **Write down an appropriate objective function to solve this ridesharing problem and minimize the total distance that all vehicles need to travel.** Your result should be similar to the regularized k-means.

(d) (Kernel k-means) Suppose we have a dataset $\{\mathbf{x}_i\}_{i=1}^n, \mathbf{x}_i \in \mathbb{R}^\ell$ that we want to split into k clusters, i.e., finding the best k-means clustering *without the regularization*. Furthermore, suppose we know *a priori* that this data is best clustered in an impractically high-dimensional feature space $\mathbb{R}^m$ with an appropriate metric. Fortunately, instead of having to deal with the

(implicit) feature map $\phi : \mathbb{R}^\ell \to \mathbb{R}^m$ and (implicit) distance metric[1], we have a kernel function $\kappa(\mathbf{x}_1, \mathbf{x}_2) = \langle \phi(\mathbf{x}_1), \phi(\mathbf{x}_2) \rangle$ that we can compute easily on the raw samples. How should we perform the kernelized counterpart of k-means clustering?

Derive the underlined portion of this algorithm.

Algorithm 1 Kernel K-means

Require: Data matrix $X \in \mathbb{R}^{n \times \ell}$; Number of clusters k; kernel function $\kappa(\mathbf{x}_1, \mathbf{x}_2)$
Ensure: Cluster id $i(j)$ for each sample x_j.
  function KERNEL-K-MEANS(X, k)
    Randomly initialize $i(j)$ to be an integer in $1, 2, \ldots, k$ for each x_j.
    while not converged **do**
      for $i \leftarrow 1$ **to** k **do**
        Set $C_i = \{j \in \{1, 2, \ldots, n\} : i(j) = i\}$.
      end for
      for $j \leftarrow 1$ **to** n **do**
        Set $i(j) = \text{argmin}_i$ _____
      end for
    end while
    Return C_i for $i = 1, 2, \ldots, k$.
  end function

(Hint: there will be no explicit representation of the "means" $\boldsymbol{\mu}_\mathbf{i}$, instead each cluster's membership itself will implicitly define the relevant quantity, in keeping with the general spirit of kernelization that we've seen elsewhere as well.)

3 Linear Methods on Fruits and Veggies

In this problem, we will use the dataset of fruits and vegetables that was collected in HW5. The goal is to accurately classify the produce in the image. Instead of operating on the raw pixel values, we operate on extracted HSV colorspace histogram features from the image. HSV histogram features extract the color spectrum of an image, so we expect these features to serve well for distinguishing produce like bananas from apples. Denote the input state $x \in \mathbb{R}^{729}$, which is an HSV histogram generated from an RGB image with a fruit centered in it. Each data point will have a corresponding class label, which corresponds to their matching produce. Given 25 classes, we can denote the label as $y \in \{0, ..., 24\}$.

Better features would of course give better results, but we chose color spectra for an initial problem for ease of interpretation.

Classification here is still a hard problem because the state space is much larger than the amount of data we obtained in the class – we are trying to perform classification in a 729 dimensional space with only a few hundred data points from each of the 25 classes. In order to obtain higher accuracy, we will examine how to perform hyper-parameter optimization and dimensionality reduction. We

[1]Just as how the interpretation of kernels in kernelized ridge regression involves an implicit prior/regularizer as well as an implicit feature space, we can think of kernels as generally inducing an implicit distance metric as well. Think of how you would represent the squared distance between two points in terms of pairwise inner products and operations on them.

will first build out each component and test on a smaller dataset of just 3 categories: apple, banana, eggplant. Then we will combine the components to perform a search over the entire dataset.

Note all python packages needed for the project, will be imported already. **DO NOT import new Python libraries.**

(a) Before we classify our data, we will study how to reduce the dimensionality of our data. We will project some of the dataset into 2D to visualize how effective different dimensionality reduction procedures are. The first method we consider is a random projection, where a matrix is randomly created and the data is linearly projected along it.

For random projections, it produces a matrix, $A \in \mathbb{R}^{2\times 729}$ where each element A_{ij} is sampled independently from a normal distribution (i.e. $A_{ij} \sim N(0,1)$). **Run the code projection.py and report the resulting plot of the projection. You do not need to write any code.**

(b) We will next examine how well PCA performs as a dimensionality-reduction tool. PCA projects the data into the subspace with the most variance, which is determined via the covariance matrix Σ_{XX}. We can compute the principal components via the singular value decomposition $U\Lambda V^T = \Sigma_{XX}$. Collecting the first two column vectors of U in the matrix U_2, we can project our data as follows:

$$\bar{x} = U_2^T x$$

Report the projected 2D points figure. You do not need to write any code.

(c) Finally, we will project our data into the Canonical Variates. In order to perform CCA, we must first turn our labels y into a one-hot encoding vector $\bar{y} \in \{0,1\}^J$, where each element corresponds to the label. Note J is the number of class labels, which is $J = 3$ for this part. Next we need to compute the canonical correlation matrix $\Sigma_{XX}^{-\frac{1}{2}}\Sigma_{XY}\Sigma_{YY}^{-\frac{1}{2}}$ and compute the singular value decomposition $U\Lambda V^T = \Sigma_{XX}^{-\frac{1}{2}}\Sigma_{XY}\Sigma_{YY}^{-\frac{1}{2}}$.

We can then project to the canonical variates by using the first k columns in U, or U_k. The projection can be written as follows:

$$\bar{x} = U_k^T \Sigma_{XX}^{-\frac{1}{2}} x$$

Report the resulting plot for CCA. You do not need to write any code. Among the dimension reduction methods we have tried, i.e. random projection, PCA and CCA, which is the best for separation among classes? Which is the worst? Why do you think this happens?

(d) We will now examine ways to perform classification using the smaller projected space from CCA as our features. One technique is to regress to the class labels and then greedily choose the model's best guess. In this problem, we will use ridge regression to learn a mapping from the HSV histogram features to the one-hot encoding $\bar{y}$ described in the previous problem. Solve the following Ridge Regression problem:

$$\min_{w} \sum_{n=1}^{N} ||\bar{y} - w^T x_n||_2^2 + \lambda ||w||_F^2$$

Then we will make predictions with the following function:

$$y = \underset{j \in 0,..,J-1}{\text{argmax}} \ (w^T x)_j$$

where $(w^T x)_j$ considers the j-th coordinate of the predicted vector. **You do not need to write any code.**

Run linear_classification.py. It will output a confusion matrix, a matrix that compares the actual label to the predicted label of the model. The higher the numerical value on the diagonal, the higher the percentage of correct predictions made by the model, thus the better model. **Report the Ridge Regression confusion matrix for the training data and validation data**.

(e) Instead of performing regression, we can potentially obtain better performance by using algorithms that are more tailored for classification problems. LDA (Linear Discriminant Analysis) approaches the problem by assuming each $p(x|y = j)$ is a normal distribution with mean μ_j and covariance Σ. Notice that the covariance matrix is assumed to be the same for all the class labels.

LDA works by fitting μ_j and Σ on the dimensionality-reduced dataset. During prediction, the class with the highest likelihood is chosen.

$$y = \underset{j \in 0,...,J-1}{\text{argmin}} \ (x - \mu_j)^T \Sigma^{-1} (x - \mu_j)$$

Fill in the class LDA_Model. Then **run linear_classification.py and report the LDA confusion matrix for the training and validation data**.

(f) LDA makes an assumption that all classes have the same covariance matrix. We can relax this assumption with QDA. In QDA, we will now parametrize each conditional distribution (still normal) by μ_j and Σ_j. The prediction function is then computed[2] as

$$y = \underset{j \in 0,..,J-1}{\text{argmin}} \ (x - \mu_j)^T \Sigma_j^{-1} (x - \mu_j) + \ln\big(\det|\Sigma_j|\big)$$

Fill in the class QDA_Model. Then **run linear_classification.py and report the QDA confusion matrix for the training data and validation data**.

(g) Let us try the Linear SVM, which fits a hyperplane to separate the dimensionality-reduced data. **Run linear_classification.py and report the Linear SVM confusion matrix for the training data and validation data. You do not need to write any code.**

[2] You should verify for yourself why this is indeed the maximum likelihood way to pick a class.

	label=1	label=0
prediction=1	True Positive	False Positive
prediction=0	False Negative	True Negative

(h) Let us try logistic regression. **Run linear_classification.py and report the logistic regression confusion matrix for the training data and validation data. You do not need to write any code.**

(i) In this part, we look at the Receiver Operating Characteristic (ROC), another approach to understanding a classifier's performance. In a two class classification problem, we can compare the prediction to the labels. Specifically, we count the number of True Positives (TP), False Positives (FP), False Negatives (FN) and True Negatives (TN). The true positive rate (TPR) measures how many positive examples out of all positive examples have been detected, concretely, TPR = TP/(TP+FN). The false positive rate (FPR) on the other hand, measures the proportion of negative examples that are mistakenly classified as positive, concretely, FPR = FP/(FP + TN).

A perfect classifier would have TPR = 1.0 and FPR = 0.0. However, in the real world, we usually do not have such a classifier. Requiring a higher TPR usually incurs the cost of a higher FPR, since that makes the classifier predict more positive outcomes. An ROC plots the trade off between TPR and FPR in a graphical manner. One way to get an ROC curve is to first obtain a set of scalar prediction scores, one for each of the validation samples. A higher score means that the classifier believes the sample is more likely to be a positive example. For each of the possible thresholds for the classifier, we can compute the TPR and FPR. After getting all (TPR, FPR) pairs, we can plot the ROC curve. **Finish the ROC function in roc.py. Run roc.py and report the ROC curve for an SVM with different regularization weight C. Which C is better and why?**

(j) If you multiply the scores output by the classifier by a factor of 10.0, how the ROC curve would change?

(k) We will finally train on the full dataset and compare the different models. We will perform a grid search over the following hyperparameters:

(a) The regularization term λ in Ridge Regression.

(b) The weighting on slack variables, C in the linear SVM.

(c) The number of dimensions, k we project to using CCA.

The file hyper_search.py contains the parameters that will be swept over. If the code is correctly implemented in the previous steps, this code should perform a sweep over all parameters and give the best model. **Run hyper_search.py, report the model parameters chosen, report the plot of the models's validation error, and report the best model's confusion matrix for validation data.** WARNING: This can take up to an hour to run.

4 Expectation Maximization (EM) Algorithm: A closer look!

Hello All! Happy Spring Break! We agree this problem *APPEARS* lengthy. Don't worry! Several parts are of tutorial nature and several other are demo in nature, where we have tried to provide ample explanation leading to the lengthy appearance of the overall problem. Hopefully you enjoy working on this homework and learn several concepts in greater detail.

The discussion will also be going over EM using examples similar to this homework.

In this problem, we will work on different aspects of the EM algorithm to reinforce your understanding of this very important algorithm.

For the first few parts, we work with the following one-dimensional mixture model:

$$\begin{aligned} Z &\sim \text{Bernoulli}(0.5) + 1 \\ X|Z=1 &\sim \mathcal{N}(\mu_1, \sigma_1^2), \quad \text{and} \\ X|Z=2 &\sim \mathcal{N}(\mu_2, \sigma_2^2), \end{aligned}$$

i.e., Z denotes the label of the Gaussian distribution from which X is drawn. In other words, we have an equal weighted 2-mixture (since Z takes value 1 or 2 with probability 0.5 each) of Gaussians, where the variances and means for both mixtures are unknown. Note that the mixture weights are given to you. (So there is one less parameter to worry about!) For a given set of parameters, we represent the likelihood of (X, Z) by $p(X, Z; \boldsymbol{\theta})$ and its log-likelihood by $\ell(X, Z; \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is used to denote the set of all unknown parameters $\boldsymbol{\theta} = \{\mu_1, \mu_2, \sigma_1, \sigma_2\}$.

Given a dataset consisting of only $(x_i, i = 1, \ldots, n)$ (and no labels z_i), our goal is to approximate the maximum-likelihood estimate of the parameters $\boldsymbol{\theta}$. In the first few parts, we walk you through one way of achieving this, namely the EM algorithm.

(a) **Write down the expression for the joint likelihood $p(X = x, Z = 1; \boldsymbol{\theta})$ and $p(X = x, Z = 2; \boldsymbol{\theta})$. What is the marginal likelihood $p(X = x; \boldsymbol{\theta})$ and the log-likelihood $\ell(X = x; \boldsymbol{\theta})$?**

(b) Now we are given a dataset where the label values Z are unobserved, i.e., we are given a set of n data points $\{x_i, i = 1, \ldots, n\}$. **Derive the expression for the log-likelihood $\ell(X_1 = x_1, \ldots, X_n = x_n)$ of a given dataset $\{x_i\}_{i=1}^n$.**

(c) Let q denote a distribution on the (hidden) labels $\{Z_i\}_{i=1}^n$ given by

$$q(Z_1 = z_1, \ldots, Z_n = z_n) = \prod_{i=1}^{n} q_i(Z_i = z_i). \tag{1}$$

Note that since $Z \in \{1, 2\}$, q has n parameters, namely $\{q_i(Z_i = 1), i = 1, \ldots, n\}$. More generally if Z took values in $\{1, \ldots, K\}$, q would have $n(K - 1)$ parameters. To simplify notation, from now on, we use the notation $\ell(x; \boldsymbol{\theta}) := \ell(X = x; \boldsymbol{\theta})$ and $p(x, k; \boldsymbol{\theta}) := p(X = x, Z = k; \boldsymbol{\theta})$. **Show that for a given point x_i, we have**

$$\ell(x_i; \boldsymbol{\theta}) \geq \mathcal{F}_i(\boldsymbol{\theta}; q_i) := \underbrace{\sum_{k=1}^{2} q_i(k) \log p(x_i, k; \boldsymbol{\theta})}_{\mathcal{L}(x_i; \boldsymbol{\theta}, q_i)} + \underbrace{\sum_{k=1}^{2} q_i(k) \log \left(\frac{1}{q_i(k)} \right)}_{H(q_i)}, \tag{2}$$

where $H(q_i)$ denotes the Shannon-entropy of the distribution q_i. Thus **conclude that we obtain the following lower bound on the log-likelihood:**

$$\ell(\{x_i\}_{i=1}^n; \boldsymbol{\theta}) \geq \mathcal{F}(\boldsymbol{\theta}; q) := \sum_{i=1}^n \mathcal{F}_i(\boldsymbol{\theta}; q_i). \tag{3}$$

Hint: Jensen's inequality, the concave-$\cap$ nature of the log, and reviewing lecture notes might be useful.

Notice that the right hand side of the bound depends on q while the left hand side does not.

(d) The EM algorithm can be considered a coordinate-ascent[3] algorithm on the lower bound $\mathcal{F}(\boldsymbol{\theta}; q)$ derived in the previous part, where we ascend with respect to $\boldsymbol{\theta}$ and q in an alternating fashion. More precisely, one iteration of the EM algorithm is made up of 2-steps:

$$q^{t+1} = \arg\max_q \mathcal{F}(\boldsymbol{\theta}^t; q) \qquad \text{(E-step)}$$
$$\boldsymbol{\theta}^{t+1} \in \arg\max_{\boldsymbol{\theta}} \mathcal{F}(\boldsymbol{\theta}; q^{t+1}). \qquad \text{(M-step)}$$

Given an estimate $\boldsymbol{\theta}^t$, the previous part tells us that $\ell(\{x_i\}_{i=1}^n; \boldsymbol{\theta}^t) \geq \mathcal{F}(\boldsymbol{\theta}^t; q)$. **Verify that equality holds in this bound if we plug in** $q(Z_1 = z_1, \ldots, Z_n = z_n) = \prod_{i=1}^n p(Z = z_i | X = x_i; \boldsymbol{\theta}^t)$ **and hence we can conclude that**

$$q^{t+1}(Z_1 = z_1, \ldots, Z_n = z_n) = \prod_{i=1}^n p(Z = z_i | X = x_i; \boldsymbol{\theta}^t). \tag{4}$$

is a valid maximizer for the problem $\max_q \mathcal{F}(\boldsymbol{\theta}^t; q)$ **and hence a valid E-step update.**

(e) Using equation (4) from above and the relation (1), we find that the E-step updates can be re-written as

$$q_i^{t+1}(Z_i = k) = p(Z = k | X = x_i; \boldsymbol{\theta}^t).$$

Using this relation, show that the E-step updates for the 2**-mixture case are given by**

$$q_i^{t+1}(Z_i = 1) = \frac{\frac{1}{\sigma_1} \exp\left(-\frac{(x_i - \mu_1)^2}{2\sigma_1^2}\right)}{\frac{1}{\sigma_1} \exp\left(-\frac{(x_i - \mu_1)^2}{2\sigma_1^2}\right) + \frac{1}{\sigma_2} \exp\left(-\frac{(x_i - \mu_2)^2}{2\sigma_2^2}\right)}, \quad \textbf{and}$$

$$q_i^{t+1}(Z_i = 2) = \frac{\frac{1}{\sigma_2} \exp\left(-\frac{(x_i - \mu_2)^2}{2\sigma_2^2}\right)}{\frac{1}{\sigma_1} \exp\left(-\frac{(x_i - \mu_1)^2}{2\sigma_1^2}\right) + \frac{1}{\sigma_2} \exp\left(-\frac{(x_i - \mu_2)^2}{2\sigma_2^2}\right)} = 1 - q_i^{t+1}(Z_i = 1).$$

Explain intuitively why these updates make sense.

[3] A coordinate-ascent algorithm is just one that fixes some coordinates and maximizes the function with respect to the others as a way of taking iterative improvement steps. (By contrast, gradient-descent algorithms tend to change all the coordinates in each step, just by a little bit.) We will discuss coordinate-ascent and coordinate-descent in more detail later.

(f) We now discuss the M-step. Using the definitions from equations (2) and (3), we have that

$$\mathcal{F}(\boldsymbol{\theta}; q^{t+1}) = \sum_{i=1}^{n} (\mathcal{L}(x_i; \boldsymbol{\theta}, q_i^{t+1}) + H(q_i)) = H(q^{t+1}) + \sum_{i=1}^{n} \mathcal{L}(\mathbf{x}_i; \boldsymbol{\theta}, q_i^{t+1}),$$

where we have used the fact that entropy in this case is given by $H(q^{t+1}) = \sum_{i=1}^{n} H(q_i^{t+1})$. Notice that although (as computed in previous part), q^{t+1} depends on $\boldsymbol{\theta}^t$, the M-step only involves maximizing $\mathcal{F}(\boldsymbol{\theta}; q^{t+1})$ with respect to just the parameter $\boldsymbol{\theta}$ while keeping the parameter q^{t+1} fixed. Now, noting that the entropy term $H(q^{t+1})$ does not depend on the parameter $\boldsymbol{\theta}$, we conclude that the M-step simplifies to solving for

$$\arg\max_{\boldsymbol{\theta}} \underbrace{\sum_{i=1}^{n} \mathcal{L}(\mathbf{x}_i; \boldsymbol{\theta}, q_i^{t+1})}_{=:\mathcal{L}(\boldsymbol{\theta}; q^{t+1})}.$$

For this and the next few parts, we use the simplified notation

$$q_i^{t+1} := q_i^{t+1}(Z_i = 1) \quad \text{and} \quad 1 - q_i^{t+1} := q^{t+1}(Z_i = 2)$$

and recall that $\boldsymbol{\theta} = (\mu_1, \mu_2, \sigma_1, \sigma_2)$. **Show that the expression for $\mathcal{L}(\boldsymbol{\theta}; q^{t+1})$ for the 2-mixture case is given by**

$$\begin{aligned}
&\mathcal{L}((\mu_1, \mu_2, \sigma_1, \sigma_2); q^{t+1}) \\
&\quad = C - \sum_{i=1}^{n} \left[q_i^{t+1} \left(\frac{(x_i - \mu_1)^2}{2\sigma_1^2} + \log \sigma_1 \right) + (1 - q_i^{t+1}) \left(\frac{(x_i - \mu_2)^2}{2\sigma_2^2} + \log \sigma_2 \right) \right],
\end{aligned}$$

where C is a constant that does not depend on $\boldsymbol{\theta}$ or q^{t+1}.

(g) Using the expression from the previous part, **show that the gradients of $\mathcal{L}(\boldsymbol{\theta}; q^{t+1})$ with respect to $\mu_1, \mu_2 \sigma_1, \sigma_2$ are given by**

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mu_1} &= -\frac{\sum_{i=1}^{n} q_i^{t+1}(\mu_1 - x_i)}{\sigma_1^2}, & \frac{\partial \mathcal{L}}{\partial \mu_2} &= -\frac{\sum_{i=1}^{n} (1 - q_i^{t+1})(\mu_2 - x_i)}{\sigma_2^2}, \\
\frac{\partial \mathcal{L}}{\partial \sigma_1} &= \frac{\sum_{i=1}^{n} q_i^{t+1}(x_i - \mu_1)^2}{\sigma_1^3} - \frac{\sum_{i=1}^{n} q_i^{t+1}}{\sigma_1}, & \frac{\partial \mathcal{L}}{\partial \sigma_2} &= \frac{\sum_{i=1}^{n} (1 - q_i^{t+1})(x_i - \mu_2)^2}{\sigma_2^2} - \frac{\sum_{i=1}^{n} (1 - q_i^{t+1})}{\sigma_2}.
\end{aligned}$$

(h) Typically, the M-step updates are computed using the stationary points for $F(\boldsymbol{\theta}; q^{t+1})$. Using the expressions from previous parts and setting the gradients to zero, **conclude that the M-step updates are given by**

$$\begin{aligned}
\mu_1^{t+1} &= \frac{\sum_{i=1}^{n} q_i^{t+1} x_i}{\sum_{i=1}^{n} q_i^{t+1}}, & \mu_2^{t+1} &= \frac{\sum_{i=1}^{n} (1 - q_i^{t+1}) x_i}{\sum_{i=1}^{n} (1 - q_i^{t+1})}, \\
(\sigma_1^2)^{(t+1)} &= \frac{\sum_{i=1}^{n} q_i^{t+1}(x_i - \mu_1^{t+1})^2}{\sum_{i=1}^{n} q_i^{t+1}}, & (\sigma_2^2)^{(t+1)} &= \frac{\sum_{i=1}^{n} (1 - q_i^{t+1})(x_i - \mu_2^{t+1})^2}{\sum_{i=1}^{n} (1 - q_i^{t+1})}
\end{aligned}$$

Explain intuitively why these updates make sense.

(i) For the next few parts, we simplify the mixture model. We work with the following simpler one-dimensional mixture model that has only a single unknown parameter:

$$\begin{aligned} Z &\sim \text{Bernoulli}(0.5) + 1 \\ X|Z = 1 &\sim \mathcal{N}(\mu, 1), \quad \text{and} \\ X|Z = 2 &\sim \mathcal{N}(-\mu, 1), \end{aligned}$$

where Z denotes the label of the Gaussian from which X is drawn. Given a set of observations only for X (i.e., the labels are unobserved), our goal is to infer the maximum-likelihood parameter μ. Using computations similar to part (a), we can conclude that the likelihood function in this simpler set-up is given by

$$p(X = x; \mu) = \frac{1}{2}\frac{e^{-\frac{1}{2}(x-\mu)^2}}{\sqrt{2\pi}} + \frac{1}{2}\frac{e^{-\frac{1}{2}(x+\mu)^2}}{\sqrt{2\pi}}.$$

For a given dataset $\{x_i, i = 1, \ldots, n\}$, **what is the log-likelihood $\ell(\{x_i\}_{i=1}^n; \mu)$ as a function of μ?**

(j) We now discuss EM updates for the set up introduced in the previous part. Let μ_t denote the estimate for μ at time t. First, we derive the E-step updates. Using part (d) (equation (4)), **show that the E-step updates simplify to**

$$q_i^{t+1}(Z_i = 1) = \frac{\exp\big(-(x_i - \mu_t)^2/2\big)}{\exp\big(-(x_i - \mu_t)^2/2\big) + \exp\big(-(x_i + \mu_t)^2/2\big)}.$$

Notice that these updates can also be derived by plugging in $\mu_1 = \mu$ and $\mu_2 = -\mu$ in the updates given in part (e).

(k) Next, we derive the M-step update. Note that we can NOT simply plug in $\mu_1 = \mu$ in the updates obtained in part (h), because the parameters are shared between the two mixtures. However, we can still make use of some of our previous computations for this simpler case. Plugging in $\mu_1 = \mu$ and $\mu_2 = -\mu$, $\sigma_1 = \sigma_2 = 1$ in part (f), **show that the objective for the M-step is given by**

$$\mathcal{L}(\mu; q^{t+1}) = C - \sum_{i=1}^{n} \left(q_i^{t+1} \frac{(x_i - \mu)^2}{2} + (1 - q_i^{t+1}) \frac{(x_i + \mu)^2}{2} \right).$$

where C is a constant independent of μ. Compute the expression for the gradient $\frac{d}{d\mu}(\mathcal{L}(\mu; q^{t+1}))$. And by setting the gradient to zero, conclude that the M-step update at time $t+1$ is given by

$$\mu_{t+1} = \frac{\sum_{i=1}^n (2q_i^{t+1} - 1)x_i}{\sum_{i=1}^n (2q_i^{t+1} - 1)} = \frac{1}{n}\sum_{i=1}^{n} (2q_i^{t+1} - 1)x_i.$$

(l) Let us now consider a direct optimization method to estimate the MLE for μ: Doing a gradient ascent algorithm directly on the complete log-likelihood function $\ell(\{x_i\}_{i=1}^n;\mu)/n$ (scaling with n is a bit natural here!). **Compute the gradient $\frac{d}{d\mu}(\ell(\{x_i\}_{i=1}^n;\mu)/n)$ and show that it is equal to**

$$\frac{d}{d\mu}\left(\frac{1}{n}\ell(\{x_i\}_{i=1}^n;\mu)\right) = \left[\frac{1}{n}\sum_{i=1}^n(2w_i-1)x_i\right] - \mu, \quad \textbf{where} \quad w_i(\mu) = \frac{e^{-\frac{(x_i-\mu)^2}{2}}}{e^{-\frac{(x_i-\mu)^2}{2}}) + e^{-\frac{(x_i+\mu)^2}{2}})}.$$

Finally conclude that the gradient ascent scheme with step size α is given by

$$\begin{aligned}\mu_{t+1}^{\mathbf{GA}} &= \mu_t^{\mathbf{GA}} + \alpha\frac{d}{d\mu}\ell(\{x_i\}_{i=1}^n;\mu)\Big|_{\mu=\mu_t^{\mathbf{GA}}} \\ &= (1-\alpha)\mu_t^{\mathbf{GA}} + \alpha\left[\frac{1}{n}\sum_{i=1}^n(2w_i(\mu_t^{\mathbf{GA}})-1)x_i\right].\end{aligned}$$

(m) **Comment on the similarity or dissimilarity between the EM and gradient ascent (GA) updates derived in the previous two parts.** You are given a code `em_gd_km.py` to run the two algorithms for the simpler one-dimensional mixture with a single unknown parameter μ. **Set part_m=True in the code and run it.** The code first generates a dataset of size 100 for two cases $\mu_{\text{true}} = 0.5$ (i.e., when the two mixtures are close) and the case $\mu_{\text{true}} = 3$ (i.e., when the two mixtures are far). We also plot the labeled dataset to help you visualize (but note that labels are not available to estimate μ_{true} and hence we use EM and GA to obtain estimates). Starting at $\mu_0 = 0.1$, the code then computes EM updates and GA updates with step size 0.05 for the dataset. **Comment on the convergence plots for the two algorithms. Do the observations match well with the similarity/dissimilarity observed in the updates derived in the previous parts?**

(n) Suppose we decided to use the simplest algorithm to estimate the parameter μ: K Means! Because the parameter is shared, we can estimate $\mu = \frac{1}{2}|\hat{\mu}_1 - \hat{\mu}_2|$ (assuming $\mu > 0$) where $\hat{\mu}_1$ and $\hat{\mu}_2$ denote the cluster centroids determined by the K means. **Do you think this strategy will work well? Does your answer depend on whether μ is large or small?** To help you answer the question, we have given a numerical implementation for this part as well. **Run the code `em_gd_km.py` with part_n=True.** The code then plots a few data-points where we also plot the hidden labels to help you understand the dataset. Also code provides the final estimates of μ by EM, Gradient Ascent (GA) and K Means. **Use the plots and the final answers to provide an intuitive argument for the questions asked above in this part. Do not spend time on being mathematically rigorous.**

Hopefully you are able to learn the following take away messages: For the simple one-dimensional mixture model, we have that

- EM works well: It converges to a good estimate of μ pretty quickly.
- Gradient ascent is a weighted version of EM: It converges to a good estimate of μ, but is slower than EM.

- K Means: Because of the hard thresholding, it converges to a biased estimate of μ if the two distributions overlap.

5 Expectation Maximization (EM) Algorithm: In Action!

Suppose we have the following general mixture of Gaussians. We describe the model by a pair of random variables $(\mathbf{X}, Z)$ where $\mathbf{X}$ takes values in $\mathbb{R}^d$ and Z takes value in the set $[K] = \{1, \ldots, K\}$. The joint-distribution of the pair $(\mathbf{X}, Z)$ is given to us as follows:

$$Z \sim \text{Multinomial}(\boldsymbol{\pi}),$$
$$(\mathbf{X}|Z = k) \sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k), \quad k \in [K],$$

where $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_K)^\top$ and $\sum_{k=1}^K \pi_k = 1$. Note that we can also write

$$\mathbf{X} \sim \sum_{k=1}^K \pi_k \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k).$$

Suppose we are given a dataset $\{\mathbf{x}_i\}_{i=1}^n$ without their labels. Our goal is to identify the K-clusters of the data. To do this, we are going to estimate the parameters $\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k$ using this dataset. We are going to use the following three algorithms for this clustering task.

K-Means: For each data-point for iteration t we find its cluster by computing:

$$y_i^{(t)} = \arg\min_{j \in [K]} \|\mathbf{x}_i - \boldsymbol{\mu}_j^{(t-1)}\|^2$$
$$C_j^{(t)} = \{\mathbf{x}_i : y_i^{(t)} == j\}_{i=1}^n$$

where $\boldsymbol{\mu}_j^{(t-1)}$ denotes the mean of $C_j^{(t-1)}$, the j-th cluster in iteration $t-1$. The cluster means are then recomputed as:

$$\boldsymbol{\mu}_j^{(t)} = \frac{1}{|C_j^{(t)}|} \sum_{\mathbf{x}_i \in C_j^{(t)}} \mathbf{x}_i.$$

We can run K-means till convergence (that is we stop when the cluster memberships do not change anymore). Let us denote the final iteration as T, then the estimate of the covariances Σ_k from the final clusters can be computed as:

$$\boldsymbol{\Sigma}_j = \frac{1}{|C_j^{(T)}|} \sum_{\mathbf{x}_i \in C_j^{(T)}} (\mathbf{x}_i - \boldsymbol{\mu}_j^{(T)})(\mathbf{x}_i - \boldsymbol{\mu}_j^{(T)})^\top.$$

Notice that this method can be viewed as a "hard" version of EM.

K-QDA: Given that we also estimate the covariance, we may consider a QDA version of K-means where the covariances keep getting updated at every iteration and also play a role in determining cluster membership. The objective at the assignment-step would be given by

$$y_i^{(t)} = \arg\min_{j \in [K]} (\mathbf{x}_i - \boldsymbol{\mu}_j^{(t-1)})^\top (\boldsymbol{\Sigma}_j^{(t-1)})^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_j^{(t-1)}).$$

$$C_j^{(t)} = \{x_i : y_i^{(t)} == j\}_{i=1}^n$$

We could then use $C_j^{(t)}$ to recompute the parameters as follows:

$$\boldsymbol{\mu}_j^{(t)} = \frac{1}{|C_j^{(t)}|} \sum_{\mathbf{x}_i \in C_j^{(t)}} \mathbf{x}_i, \quad \text{and}$$

$$\boldsymbol{\Sigma}_j^{(t)} = \frac{1}{|C_j^{(t)}|} \sum_{\mathbf{x}_i \in C_j^{(t)}} (\mathbf{x}_i - \boldsymbol{\mu}_j^{(t)})(\mathbf{x}_i - \boldsymbol{\mu}_j^{(t)})^\top.$$

We again run K-QDA until convergence (that is we stop when the cluster memberships do not change anymore). Notice that, again, this method can be viewed as another variant for the "hard" EM method.

EM: The EM updates are given by

- E-step: For $k = 1, \ldots, K$ and $i = 1 \ldots, n$, we have

$$q_i^{(t)}(Z_i = k) = p(Z = k | \mathbf{X} = \mathbf{x}_i; \boldsymbol{\theta}^{(t-1)}).$$

- M-step: For $k = 1, \ldots, K$, we have

$$\pi_k^{(t)} = \frac{1}{n} \sum_{i=1}^n q_i^{(t)}(Z_i = k) = \frac{1}{n} \sum_{i=1}^n p(Z = k | \mathbf{X} = \mathbf{x}_i; \boldsymbol{\theta}^{(t-1)}),$$

$$\boldsymbol{\mu}_k^{(t)} = \frac{\sum_{i=1}^n q_i^{(t)}(Z_i = k)\mathbf{x}_i}{\sum_{i=1}^n q_i^{(t)}(Z_i = k)}, \quad \text{and}$$

$$\boldsymbol{\Sigma}_k{}^{(t)} = \frac{\sum_{i=1}^n q_i^{(t)}(Z_i = k)(\mathbf{x}_i - \boldsymbol{\mu}_k^{(t)})(\mathbf{x}_i - \boldsymbol{\mu}_k^{(t)})^\top}{\sum_{i=1}^n q_i^{(t)}(Z_i = k)}.$$

Notice that unlike previous two methods, in the EM updates, each data point contributes in determining the mean and covariance for each cluster.

We now see the three methods in action. You are provided with a code for all the above 3 algorithms (`gmm_em_kmean.py`). You can run it by calling the following function from main:

```
experiments(seed, factor, num_samples, num_clusters)
```

We assume that $\mathbf{x} \in \mathbb{R}^2$, and the default settings are number of samples is 500 ($n = 500$), and the number of clusters is 3 ($K = 3$). Notice that `seed` will determine the randomness and `factor` will determine how far apart are the clusters.

(a) Run the following setting:

```
experiments(seed=11, factor=1, num_samples=500, num_clusters=3)
```

Observe the initial guesses for the means and the plots for the 3 algorithms on convergence. **Comment on your observations.** Note that the colors are used to indicate that the points that belong to different clusters, to help you visualize the data and understand the results.

(b) **Comment on the results obtained for the following setting:**

```
experiments(seed=63, factor=10, num_samples=500, num_clusters=3)
```

6 Your Own Question

Write your own question, and provide a thorough solution.

Writing your own problems is a very important way to really learn the material. The famous "Bloom's Taxonomy" that lists the levels of learning is: Remember, Understand, Apply, Analyze, Evaluate, and Create. Using what you know to create is the top-level. We rarely ask you any HW questions about the lowest level of straight-up remembering, expecting you to be able to do that yourself. (e.g. make yourself flashcards) But we don't want the same to be true about the highest level.

As a practical matter, having some practice at trying to create problems helps you study for exams much better than simply counting on solving existing practice problems. This is because thinking about how to create an interesting problem forces you to really look at the material from the perspective of those who are going to create the exams.

Besides, this is fun. If you want to make a boring problem, go ahead. That is your prerogative. But it is more fun to really engage with the material, discover something interesting, and then come up with a problem that walks others down a journey that lets them share your discovery. You don't have to achieve this every week. But unless you try every week, it probably won't happen ever.