EECS 189 Introduction to Machine Learning
Fall 2020

HW10

This homework is due **Wednesday, November 11 at 11:59 p.m.**

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 the appropriate assignment on Gradescope. 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.
3. If there is a test set, submit your test set evaluation results.

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 inadvertently cheats.

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions nor have I looked at any online solutions to any of these problems. I have credited all external sources in this write up.

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

Note that this problem appears lengthy. Don't worry! Several parts are of a tutorial nature and several others are of a demo nature, where we have tried to provide ample explanations. This leads to the lengthy appearance of the overall problem, even though it is not actually that long. 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:

$$Z \sim \begin{cases} 1 & \text{w.p. } \frac{1}{2} \\ 2 & \text{w.p. } \frac{1}{2} \end{cases}$$
$$X|Z = 1 \sim \mathcal{N}(\mu_1, \sigma_1^2), \quad \text{and}$$
$$X|Z = 2 \sim \mathcal{N}(\mu_2, \sigma_2^2).$$

Z denotes the hidden label of the Gaussian distribution from which X is drawn. $p(Z = 1) = 0.5, p(Z = 2) = 0.5$ — the hidden label is a fair coin toss. X has two different Gaussian distributions based on what Z is. In other words, we have an equally weighted 2-mixture of Gaussians, where the variances and means for both elements of the mixture are unknown. (Note that the mixture weights are given to you. In general, even those would be unknown.)

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 presumed to consist of i.i.d. samples of only $(x_i, i = 1, \ldots, n)$ (and no labels z_i), our goal is to iteratively 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 an i.i.d. 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** $\ell(X_1 = x_1, \ldots, X_n = x_n; \boldsymbol{\theta})$ **of a given dataset** $\{x_i\}_{i=1}^n$**.**

(c) Let q denote a possible 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 $z_i \in \{1, 2\}$, 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:**

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

©UCB EECS 189, Fall 2020. All Rights Reserved. This may not be publicly shared without explicit permission.

Hint: Jensen's inequality (for concave-$\cap$ functions f, we know $f(\mathbb{E}[X]) \geq \mathbb{E}(f(X))$ since a line joining two points is below the function), the concave-$\cap$ nature of the log, and reviewing your notes from lecture 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[1] 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.

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

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

[1] 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.)

©UCB EECS 189, Fall 2020. All Rights Reserved. This may not be publicly shared without explicit permission.

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, it is easy to show (make sure you know how to do this) 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

$$\frac{\partial \mathcal{L}}{\partial \mu_1} = -\frac{\sum_{i=1}^n q_i^{t+1}(\mu_1 - x_i)}{\sigma_1^2}, \qquad \frac{\partial \mathcal{L}}{\partial \mu_2} = -\frac{\sum_{i=1}^n (1 - q_i^{t+1})(\mu_2 - x_i)}{\sigma_2^2},$$

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

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 these gradients to zero, **conclude that the M-step updates are given by**

$$\mu_1^{t+1} = \frac{\sum_{i=1}^n q_i^{t+1} x_i}{\sum_{i=1}^n q_i^{t+1}}, \qquad \mu_2^{t+1} = \frac{\sum_{i=1}^n (1 - q_i^{t+1}) x_i}{\sum_{i=1}^n (1 - q_i^{t+1})},$$

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

Explain intuitively why these updates make sense.

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

$$Z \sim \begin{cases} 1 & \text{w.p. } \frac{1}{2} \\ 2 & \text{w.p. } \frac{1}{2} \end{cases}$$

$$X|Z = 1 \sim \mathcal{N}(\mu, 1), \quad \text{and}$$
$$X|Z = 2 \sim \mathcal{N}(-\mu, 1),$$

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

(i) 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)) and part (e), **show that the E-step updates simplify to**

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

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).

(j) 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

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

(k) 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 natural here since the log-likelihood is a sum.). **Compute the gradient $\frac{d}{d\mu}(\ell(\{x_i\}_{i=1}^n; \mu)/n)$ and show that it is equal to**

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

©UCB EECS 189, Fall 2020. All Rights Reserved. This may not be publicly shared without explicit permission.

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

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

(l) **Comment on the similarity or dissimilarity between the EM and gradient ascent (GA) updates derived in the previous two parts.** Refer to the corresponding jupyter notebook. You can run the two algorithms for the simpler one-dimensional mixture with a single unknown parameter μ. Run corresponding part in the notebook first. 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 (attach 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?**

(m) Suppose we decided to use the simplest algorithm to estimate the parameter μ: K Means! Because the parameter is shared, assuming $\mu > 0$, we can estimate $\hat{\mu} = \frac{1}{n_1+n_2}\left|n_1\hat{\mu}_1 - n_2\hat{\mu}_2\right|$, where n_1, n_2 denote the size of the two clusters at the time of update. $\hat{\mu}_1$ 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 of corresponding part in jupyter notebook.** 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 (report 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.

3 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 $[K] = \{1, \ldots, K\}$.

©UCB EECS 189, Fall 2020. All Rights Reserved. This may not be publicly shared without explicit permission.

The joint-distribution of the pair $(\mathbf{X}, Z)$ is given to us as follows:

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

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. (The EM updates will also pull out an estimate for $\boldsymbol{\pi}$ because the prior probability of a point being in each cluster is important for the probabilistic model, but K-Means style algorithms don't care about that.) 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 aim to 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 relevant distance to the mean has to be relative to the norm induced by the inverse covariance matrix). The objective at the assignment-step would be given by

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

HW10, ©UCB EECS 189, Fall 2020. All Rights Reserved. This may not be publicly shared without explicit permission.

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 since each point is assigned to exactly one cluster.

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 in jupyter notebook for all the above 3 algorithms. 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 (as written in the corresponding part):

```
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. Attach the two plots for this case.**

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:** (as written in the corresponding part)

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

and attach the two plots for this case as well.

4 Implicit Bias of Gradient Descent for Logistic Regression

You have seen in an earlier homework that gradient descent on squared-loss will tend to the minimum-norm solution if initialized at zero. We focused in on the special case where you could get to a perfect solution (perfectly interpolate all the training data) where you get the lowest-possible squared-loss, namely zero. Here, we study what happens for logistic regression in the counterpart case.

In this problem, we will consider the effects of the method used to train a model on the final outcome. Specifically, consider a classification problem with two *linearly separable* classes of points $\{\mathbf{a}_i\}$ and $\{\mathbf{b}_i\}$. (This condition is the logistic regression counterpart of having a perfect solution possible — we can perfectly separate the training points.) Let the first class of points have label 0, and the second class of points have label $+1$. (We used -1 and $+1$ earlier at times, but here we will use 0 and $+1$ as the labels.)

Recall that logistic regression associates a point $\mathbf{x}$ with a real number from 0 to 1 by computing:

$$f(\mathbf{x}) = \frac{1}{1 + \exp\{-\mathbf{w}^T\mathbf{x}\}},$$

This number can be interpreted as the estimated probability for the point $\mathbf{x}$ having a true label of $+1$. Since this number is $\frac{1}{2}$ when $\mathbf{w}^T\mathbf{x} = 0$, the sign of $\mathbf{w}^T\mathbf{x}$ is what predicts the label of the test point $\mathbf{x}$.

The loss function is defined to be

$$\sum_i y_i \ln\left(\frac{1}{f(\mathbf{x}_i)}\right) + (1 - y_i) \ln\left(\frac{1}{1 - f(\mathbf{x}_i)}\right), \tag{5}$$

where the label of the ith point $\mathbf{x}_i$ is y_i. This form illustrates the "log-loss" nature of logistic loss. Notice that when $y_i = +1$, the second term inside the sum is gone and all we want is $\ln(f(\mathbf{x}_i))$ to be close to 0 which will happen if $f(\mathbf{x}_i)$ is close to 1, which happens if $\mathbf{w}^T\mathbf{x}_i$ is a huge positive number. A similar story for $y_i = 0$ makes us want $\mathbf{w}^T\mathbf{x}_i$ to be a huge negative number.

In the two-class example defined above, it can be rewritten as

$$\sum_i \ln\left(1 + e^{\mathbf{w}^T\mathbf{a}_i}\right) + \sum_i \ln\left(1 + e^{-\mathbf{w}^T\mathbf{b}_i}\right), \tag{6}$$

summing over the points in each class separately.

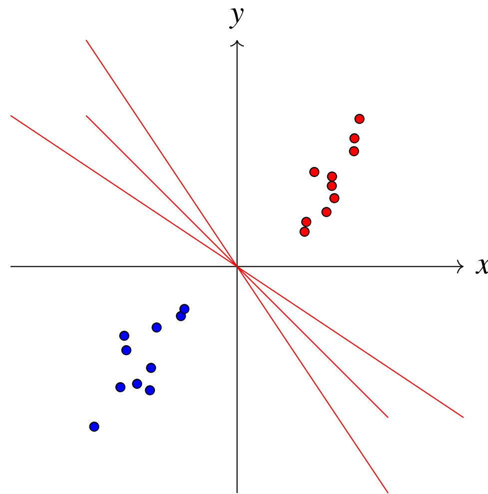
(a) Consider trying to minimize (6) when our sets of points are linearly separable (i.e. there exists a $\mathbf{w}$ such that $\mathbf{w}^T\mathbf{a}_i < 0$ and $\mathbf{w}^T\mathbf{b}_j > 0$ for all valid i.) **Does a single "optimal" solution exist for minimizing (6)? How low can the loss go?**

 ©UCB EECS 189, Fall 2020. All Rights Reserved. This may not be publicly shared without explicit permission.

(Hint: If $\mathbf{w}$ is such that it properly separates the two classes of points, what happens to the loss in (6) if we doubled $\mathbf{w}$? Also, look at the individual terms in (6) — can any of those terms be negative?)

(b) In the previous part, you should have seen that *any* $\mathbf{w}$ that correctly separates our training data by class (i.e. $\text{sign}(\mathbf{w}^T\mathbf{x})$ has zero classification error on the training data) can be used to produce a set of weights $\mathbf{w}'$ with an arbitrarily low loss on the training set when evaluating (6).

In most cases of separable training data, infinitely many qualitatively distinct $\mathbf{w}$s can exist that separate the training data, as shown in the below figure:



Yet when we try to fit a model by minimizing (6) using standard techniques such as gradient descent, it turns out that we tend to converge to a particular classifier, even though infinitely many exist with arbitrarily small loss.

To understand this effect, we will consider a very simple one-dimensional example where the points are scalars. To be able to place the separator anywhere on the line, we will introduce a bias term into our model so that there are two parameters. In particular, here we consider

$$f(x) = \frac{1}{1 + \exp\{-m(x-c)\}}$$

with decision boundary $x = c$. (All the numbers bigger than c are deemed to belong to the class 1 and all the points smaller than c are deemed to belong to the class 0.) For any c that separates our training data, it is easy to see that as $m \to \infty$, the training loss in (5) goes to zero if we assume that all the +1-labeled training points are indeed bigger than c and all the 0-labeled training points are indeed smaller than c.

Explicitly write down the loss function for logistic regression in this example, assuming that we have the points $a_1, \ldots, a_{n_1}$ in class 0, and $b_1, \ldots, b_{n_2}$ in class 1 following the example of (6) so that the dependence on the two scalar parameters m and c is clear.

(c) **Compute the gradient descent update step with step size γ.** Give updates for both m and c.

(d) Now, you can imagine that based on where we are initialized and based on where the points are, that gradient descent will take some path as it minimizes. Here, we are going to assume that our training data is separable, so $\max(\{a_i\}) < \min(\{b_i\})$. It is important to note that initially, it is possible that c does *not* separate our data i.e. there may exist an i such that $c < a_i$, or $c > b_i$. We can start with some points on the wrong side of c.

It turns out that logisitic loss is nicely convex and so with a sufficiently small step size γ, gradient-descent will take us towards a global minimizer. (Strictly speaking, a global infimizer.) Throughout this problem, you may use the fact that gradient descent, when applied to minimize our loss function, can drive the loss down to $L_{min} + \varepsilon$ for any $\varepsilon > 0$, where L_{min} is the minimum possible loss (0, in the separable case).

Show that gradient descent will move c to a value that does separate the training data.

(Hint: Suppose that there was a point that was on the wrong side of c. How low could your loss be if that were true?)

(e) You have now shown that, after some time, gradient descent will be forced to drive c strictly between the two classes in the training data. For a sufficiently small choice of learning rate, it is also known that gradient descent updates only strictly decrease the loss being optimized by gradient descent. Thus, we have in fact shown that c will continue separate to our training data by class as we take further steps in gradient descent.

At this point, there are two conceptual steps to understand why c must converge to the max-margin choice. First, we are going to show that m must grow unboundedly. Then, we will leverage the idea that m has gotten very large to allow us to focus on the class members on the boundary and their impact on gradient descent.

Here, we are asking you to do the first part: after c separates our training data by class, **show that m grows unboundedly towards infinity.**

For simplicity, assume that $m^{(t)} > 0$ when c first separates our training points by class.

If you want, you can use approximation or heuristic-based reasoning to understand the rate of growth, as a function of the number of gradient descent steps. To do this, you may also assume a sufficiently small step size γ, that is so small that you can approximate m as a continuous function of t, with

$$m^{(t+1)} - m^{(t)} \approx \frac{dm}{dt}.$$

This will permit you to use a differential equation to understand the growth of m.

(Hint: You want to show that the gradient is going to be pushing $m(t)$ to be bigger. How small can this gradient be? That will give you a lower bound of the form $\frac{\kappa}{1+\exp\{m(t)\beta\}}$ where κ can be made as small as you want by setting the step-size, and β is something that has to do with the training data. You could similarly upper-bound this gradient if you wanted to, even though we didn't ask you to. Notice that this bound by itself can allow you to prove that $m(t)$ cannot stay bounded. Why? Suppose that $m(t)$ stayed bounded. Then the gradient would also stay bounded and positive. Why does that lead to a contradiction?

You can also apply the approximation that $\frac{dm}{dt}$ = your lower-bound for the gradient and look at that differential equation. The gradient is positive and so it is growing. Solve this differential

©UCB EECS 189, Fall 2020. All Rights Reserved. This may not be publicly shared without explicit permission.

equation by guessing a solution using separation-of-variables if you want to get the asymptotic scaling.

(f) Now that we have shown that m tends towards infinity, let's see what is eventually going to happen to c. This will tell us which of the classifiers separating our training data gradient descent converges towards.

Assume without loss of generality that

$$a_{n_1} < a_{n_1-1} < \cdots < a_2 < a_1 < b_1 < b_2 < \cdots < b_{n_2},$$

so that the two points a_1 and b_1 define the boundaries of the training data.

Our goal is to show that gradient descent eventually pushes c to $c^* = \frac{a_1+b_1}{2}$.

Suppose that we are at c that is somewhere between $\frac{\delta}{2}$ and δ away from c^*, **show that when m is large enough, the gradient (with respect to c) of the loss is dominated by the contributions from the closer point to c among a_1 and b_1 and that gradient descent is pushing c towards c^*.**

(g) The final thing to do is to make sure that the force of gradient descent can actually move c enough to close the gap by $\frac{\delta}{2}$ in a finite amount of time if c were to stay in the region between $\frac{\delta}{2}$ and δ away from c^*. **Argue why this must be true based on the fact that $m(t)$ must grow unboundedly by comparing the gradient pushing c to the one that is pushing m.**

Once you've done that, you have shown that c has to close the gap to $\frac{\delta}{2}$ away in a finite time, and since δ was arbitary, this means that we have to close the gap to $\frac{\delta}{4}$ and then to $\frac{\delta}{8}$ and so on all the way as c converges to c^* itself. It cannot converge anywhere else.

Technically speaking, we also would have to rule out the prospect of gradient descent with a constant step size constantly jumping too far over across c^* to result in an infinite oscillation. However, you can also show this cannot happen forever based on the fact that $m(t) \to \infty$ implies that the gradients themselves are getting smaller and smaller. No finite jump could be sustained.

(h) Run the associated Jupyter notebook to observe this effect, and its generalization to d-dimensions, numerically. **Comment on what you observe in the notebook.**

5 Linear Methods on Fruits and Veggies

In previous semesters that were not struck by the pandemic and shelter-at-home restrictions, students were asked to collectively collect a dataset for the class. Unfortunately, this is hard to do this semester and so we are going to have to trust that you had enough experience collecting data for learning in 16AB's labs, or if you are a graduate student, in the appropriate lab-based course for your field. In this problem, we will use a dataset of fruits and vegetables that was collected in previous semesters.

The goal is to accurately classify the produce in the image. Instead of operating on the raw pixel values (because that can be slow), 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 $y \in \{0, \dots, 24\}$.

Better features would of course give better results, but we chose color spectra for an initial problem for ease of interpretation. As a historical note, early image search engines also used color histograms to group images. HSV is a more perceptually relevant color-space than RGB, and that is why we are using those features.

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 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, $A \in \mathbb{R}^{2\times729}$ where each element A_{ij} is sampled independently from a normal distribution (i.e. $A_{ij} \sim N(0, 1)$).

Run the Part (a) in Jupyter Notebook 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 $U\Lambda V^\top = \Sigma_{XX}$. Collecting the first two column vectors of U in the matrix U_2, we can project our data as follows:

$$\bar{x} = U_2^\top x.$$

Run the Part (b) in Jupyter Notebook and report the projected 2D points figure. You do not need to write any code.

(c) Finally, we will use what is called **Canonical Correlation Analysis (CCA)** to do dimensionality reduction. As PCA provided us with a dimensionality-reduction approach that didn't use the labels y in any way, what would happen to PCA if the data by nature was "contaminated" with a strong correlated latent signal that had nothing to do with the labels? PCA would select

projections with the greatest variation and keep them (even though they are irrelevant to predicting the labels), in effect throwing away those dimensions where we could actually hope to get information relevant for predicting the label.

We saw in an earlier homework how we can use auxiliary labels to help us find the relevant directions in the inputs that correspond to latent variables that are useful in predicting the auxiliary labels as a collective. In this problem, we are going to exploit the fact that we have more than one class. This means, we can use one-hot encoding to get vector-valued $\bar{y} \in \{0, 1\}^J$ where J is the number of class labels, which is $J = 3$ for this part. Each of these dimensions can function as an auxiliary label.

In the earlier homework, we saw that the object whose SVD is desired is $E[\mathbf{x}\mathbf{z}^\top]$. In that problem however, we had made sure that that the input data $\mathbf{x}$ by its nature did not have any strong directions in it. If we want to suppress any strong directions in our $\mathbf{x}$ inputs, we can apply a normalization to our data set that is called "whitening" — whereby we change coordinates to that our $\mathbf{X}$ matrix has all of its nonzero singular values being constant. This can be done by applying a change of coordinates that is given by the square-root of the PSD matrix that represents the inverse of the empirical covariance of the $\mathbf{x}$ in the data.

In an earlier homework, you also saw that there can be issues that arise from having class imbalances in our data set. Such class imbalances manifest in one-hot encodings by having some the "label-features" (banana or not-banana) having a higher norm than others — notice that under one-hot encoding, the empirical "label-feature" vectors are guaranteed to be orthogonal to begin with. Reweighting data-points to achieve class balance is spiritually akin to whitening the $\mathbf{y}$ one-hot-encodings of the labels.

CCA computes the singular value decomposition $U\Lambda V^\top = \Sigma_{XX}^{-\frac{1}{2}}\Sigma_{XY}\Sigma_{YY}^{-\frac{1}{2}}$ to basically work with such whitened $\mathbf{x}$ and $\mathbf{y}$ data. We do not want to be biased by strong directions in $\mathbf{x}$, nor by strong directions in $\mathbf{y}$ — we just want to understand which directions in a whitened $\mathbf{x}$ help us predict a whitened $\mathbf{y}$.

We can then project to the canonical variates by using the first k columns in U, or U_k together with the whitening. The complete lower-dimensional projection can be written as follows:

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

In this problem, however, you can just treat CCA as a black-box tool for dimensionality-reduction that takes label information into account.

It is interesting to note that although theoretically, once you have picked the first J singular vectors, there is no further label-targeted information in the singular vectors beyond that point. However, there is still information about the inputs there and so formally, we do not have to stop at J when it comes to reducing dimensionality.

Run the Part (c) in Jupyter Notebook and 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?

©UCB EECS 189, Fall 2020. All Rights Reserved. This may not be publicly shared without explicit permission.

(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:

$$\min_W \sum_{n=1}^{N} \|\bar{y} - W^\top x_n\|_2^2 + \lambda \|W\|_F^2.$$

Then we will make predictions with the following function:

$$y = \operatorname*{argmax}_{j \in 0,..,J-1} (W^\top x)_j,$$

where $(W^\top x)_j$ considers the j-th coordinate of the predicted vector.

Run the Part (d) in Jupyter Notebook. You do not need to write any code. 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.

$$y = \operatorname*{argmin}_{j \in 0,...,J-1} (x - \mu_j)^\top \Sigma^{-1} (x - \mu_j).$$

Fill in the Part (e) in Jupyter Notebook. 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

$$y = \operatorname*{argmin}_{j \in 0,..,J-1} (x - \mu_j)^\top \Sigma_j^{-1} (x - \mu_j) + \ln\Big(\det|\Sigma_j|\Big).$$

Fill in the Part (f) in Jupyter Notebook. Report the QDA confusion matrix for the training and validation data.

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

©UCB EECS 189, Fall 2020. All Rights Reserved. This may not be publicly shared without explicit permission.

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

(g) Let us try the Linear SVM, which fits a hyperplane to separate the dimensionality-reduced data. The regularization parameter of SVM (the squared ℓ_2 regularization in SVM) is inversely proportional to $C > 0$ in the Jupyter Notebook. *You can treat this C as a tuning hyperparameter in this problem.*

Run the Part (g) in Jupyter Notebook. Report the Linear SVM confusion matrix for the training data and validation data. You do not need to write any code.

(h) Let us try logistic regression.

Run the Part (h) in Jupyter Notebook. 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 Part (i) in Jupyter Notebook. 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.

Part (k) in Jupyter Notebook 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 the Part (k) in Jupyter Notebook, 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.

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.

Contributors:

- Alex Tsigler
- Anant Sahai
- Michael Laskey
- Peter Wang
- Raaz Dwivedi
- Rahul Arya
- Ruta Jawale
- Stella Yu
- Yaodong Yu

©UCB EECS 189, Fall 2020. All Rights Reserved. This may not be publicly shared without explicit permission.