CS 229, Spring 2020
Problem Set #3

Due Wednesday, May 27 at 11:59 pm on Gradescope.

Notes: (1) These questions require thought, but do not require long answers. Please be as concise as possible. (2) If you have a question about this homework, we encourage you to post your question on our Piazza forum, at `https://piazza.com/stanford/spring2020/cs229`. (3) This quarter, Spring 2020, students may submit in pairs. If you missed the first lecture or are unfamiliar with the collaboration or honor code policy, please read the policy on the course website before starting work. (4) For the coding problems, you may not use any libraries except those defined in the provided `environment.yml` file. In particular, ML-specific libraries such as scikit-learn are not permitted. (5) To account for late days, the due date is Wednesday, May 27 at 11:59 pm. If you submit after Wednesday, May 27 at 11:59 pm, you will begin consuming your late days. If you wish to submit on time, submit before Wednesday, May 27 at 11:59 pm.

All students must submit an electronic PDF version of the written questions. We highly recommend typesetting your solutions via LaTeX. All students must also submit a zip file of their source code to Gradescope, which should be created using the `make_zip.py` script. You should make sure to (1) restrict yourself to only using libraries included in the `environment.yml` file, and (2) make sure your code runs without errors. Your submission may be evaluated by the auto-grader using a private test set, or used for verifying the outputs reported in the writeup.

1. **[25 points] A Simple Neural Network**

Let $X = \{x^{(1)}, \cdots, x^{(n)}\}$ be a dataset of n samples with 2 features, i.e $x^{(i)} \in \mathbb{R}^2$. The samples are classified into 2 categories with labels $y^{(i)} \in \{0, 1\}$. A scatter plot of the dataset is shown in Figure 1:

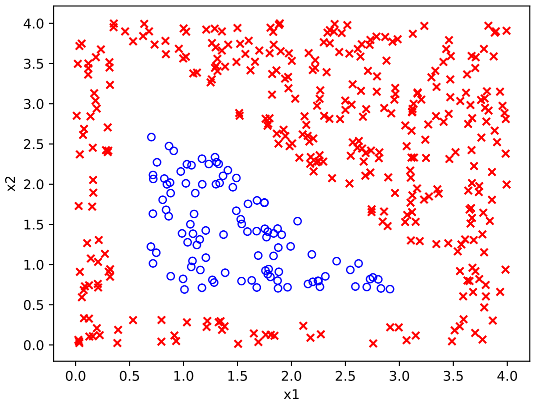

Figure 1: Plot of dataset X.

The examples in class 1 are marked as as "$\times$" and examples in class 0 are marked as "$\circ$". We want to perform binary classification using a simple neural network with the architecture shown in Figure 2:

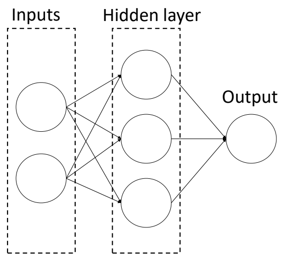

Figure 2: Architecture for our simple neural network.

Denote the two features x_1 and x_2, the three neurons in the hidden layer h_1, h_2, and h_3, and the output neuron as o. Let the weight from x_i to h_j be $w^{[1]}_{i,j}$ for $i \in \{1, 2\}, j \in \{1, 2, 3\}$, and the weight from h_j to o be $w^{[2]}_j$. Finally, denote the intercept weight for h_j as $w^{[1]}_{0,j}$, and the intercept weight for o as $w^{[2]}_0$. For the loss function, we'll use average squared loss instead of the usual negative log-likelihood:

$$l = \frac{1}{n} \sum_{i=1}^{n} \left(o^{(i)} - y^{(i)} \right)^2,$$

where $o^{(i)}$ is the result of the output neuron for example i.

(a) [5 points] Suppose we use the sigmoid function as the activation function for h_1, h_2, h_3 and o. What is the gradient descent update to $w^{[1]}_{1,2}$, assuming we use a learning rate of α? Your answer should be written in terms of $x^{(i)}$, $o^{(i)}$, $y^{(i)}$, and the weights.

(b) [10 points] Now, suppose instead of using the sigmoid function for the activation function for h_1, h_2, h_3 and o, we instead used the step function $f(x)$, defined as

$$f(x) = \begin{cases} 1, x > 0 \\ 0, x \leq 0 \end{cases}$$

Is it possible to have a set of weights that allow the neural network to classify this dataset with 100% accuracy?

If you believe it's possible, please implement your approach by completing the `optimal_step_weights` method in `src/simple_nn/simple_nn.py` and including the corresponding `step_weights.pdf` plot showing perfect prediction in your writeup.

If it is not possible, please explain your reasoning in the writeup.

Hint 1: There are three sides to a triangle, and there are three neurons in the hidden layer.

Hint 2: A solution can be found where all weight and bias parameters take values only in $\{-1, -0.5, 0, 1, 3, 4\}$. You are free to come up with other solutions as well.

(c) [10 points] Let the activation functions for h_1, h_2, h_3 be the linear function $f(x) = x$ and the activation function for o be the same step function as before.

Is it possible to have a set of weights that allow the neural network to classify this dataset with 100% accuracy?

If you believe it's possible, please implement your approach by completing the `optimal_linear_weights` method in `src/simple_nn/simple_nn.py` and including the corresponding `linear_weights.pdf` plot showing perfect prediction in your writeup.

If it is not possible, please explain your reasoning in the writeup.

Hint: The hints from the previous sub-question might or might not apply.

2. **[15 points] KL divergence and Maximum Likelihood**

The Kullback-Leibler (KL) divergence is a measure of how much one probability distribution is different from a second one. It is a concept that originated in Information Theory, but has made its way into several other fields, including Statistics, Machine Learning, Information Geometry, and many more. In Machine Learning, the KL divergence plays a crucial role, connecting various concepts that might otherwise seem unrelated.

In this problem, we will introduce KL divergence over discrete distributions, practice some simple manipulations, and see its connection to Maximum Likelihood Estimation.

The *KL divergence* between two discrete-valued distributions $P(X), Q(X)$ over the outcome space $\mathcal{X}$ is defined as follows[1]:

$$D_{KL}(P\|Q) = \sum_{x\in\mathcal{X}} P(x) \log \frac{P(x)}{Q(x)}$$

For notational convenience, we assume $P(x) > 0, \forall x$. (One other standard thing to do is to adopt the convention that "$0 \log 0 = 0$.") Sometimes, we also write the KL divergence more explicitly as $D_{KL}(P||Q) = D_{KL}(P(X)||Q(X))$.

Background on Information Theory

Before we dive deeper, we give a brief (optional) Information Theoretic background on KL divergence. While this introduction is not necessary to answer the assignment question, it may help you better understand and appreciate why we study KL divergence, and how Information Theory can be relevant to Machine Learning.

We start with the *entropy* $H(P)$ of a probability distribution $P(X)$, which is defined as

$$H(P) = -\sum_{x\in\mathcal{X}} P(x) \log P(x).$$

Intuitively, entropy measures how dispersed a probability distribution is. For example, a uniform distribution is considered to have very high entropy (i.e. a lot of uncertainty), whereas a distribution that assigns all its mass on a single point is considered to have zero entropy (i.e. no uncertainty). Notably, it can be shown that among continuous distributions over $\mathbb{R}$, the Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$ has the highest entropy (highest uncertainty) among all possible distributions that have the given mean μ and variance σ^2.

To further solidify our intuition, we present motivation from communication theory. Suppose we want to communicate from a source to a destination, and our messages are always (a sequence of) discrete symbols over space $\mathcal{X}$ (for example, $\mathcal{X}$ could be letters $\{\mathrm{a}, \mathrm{b}, \dots, \mathrm{z}\}$). We want to construct an encoding scheme for our symbols in the form of sequences of binary bits that are transmitted over the channel. Further, suppose that in the long run the frequency of occurrence of symbols follow a probability distribution $P(X)$. This means, in the long run, the fraction of times the symbol x gets transmitted is $P(x)$.

A common desire is to construct an encoding scheme such that the average number of bits per symbol transmitted remains as small as possible. Intuitively, this means we want very frequent symbols to be assigned to a bit pattern having a small number of bits. Likewise, because we are

[1]If P and Q are densities for continuous-valued random variables, then the sum is replaced by an integral, and everything stated in this problem works fine as well. But for the sake of simplicity, in this problem we'll just work with this form of KL divergence for probability mass functions/discrete-valued distributions.

interested in reducing the average number of bits per symbol in the long term, it is tolerable for infrequent words to be assigned to bit patterns having a large number of bits, since their low frequency has little effect on the long term average. The encoding scheme can be as complex as we desire, for example, a single bit could possibly represent a long sequence of multiple symbols (if that specific pattern of symbols is very common). The entropy of a probability distribution $P(X)$ is its optimal bit rate, i.e., the lowest average bits per message that can possibly be achieved if the symbols $x \in \mathcal{X}$ occur according to $P(X)$. It does not specifically tell us *how* to construct that optimal encoding scheme. It only tells us that no encoding can possibly give us a lower long term bits per message than $H(P)$.

To see a concrete example, suppose our messages have a vocabulary of $K = 32$ symbols, and each symbol has an equal probability of transmission in the long term (i.e, uniform probability distribution). An encoding scheme that would work well for this scenario would be to have $\log_2 K$ bits per symbol, and assign each symbol some unique combination of the $\log_2 K$ bits. In fact, it turns out that this is the most efficient encoding one can come up with for the uniform distribution scenario.

It may have occurred to you by now that the long term average number of bits per message depends only on the frequency of occurrence of symbols. The encoding scheme of scenario A can in theory be reused in scenario B with a different set of symbols (assume equal vocabulary size for simplicity), with the same long term efficiency, as long as the symbols of scenario B follow the same probability distribution as the symbols of scenario A. It might also have occurred to you, that reusing the encoding scheme designed to be optimal for scenario A, for messages in scenario B having a *different probability* of symbols, will always be suboptimal for scenario B. To be clear, we do not need know *what* the specific optimal schemes are in either scenarios. As long as we know the distributions of their symbols, we can say that the optimal scheme designed for scenario A will be suboptimal for scenario B if the distributions are different.

Concretely, if we reuse the optimal scheme designed for a scenario having symbol distribution $Q(X)$, into a scenario that has symbol distribution $P(X)$, the long term average number of bits per symbol achieved is called the *cross entropy*, denoted by $H(P, Q)$:

$$H(P, Q) = -\sum_{x \in \mathcal{X}} P(x) \log Q(x).$$

To recap, the entropy $H(P)$ is the best possible long term average bits per message (optimal) that can be achieved under a symbol distribution $P(X)$ by using an encoding scheme (possibly unknown) specifically designed for $P(X)$. The cross entropy $H(P, Q)$ is the long term average bits per message (suboptimal) that results under a symbol distribution $P(X)$, by reusing an encoding scheme (possibly unknown) designed to be optimal for a scenario with symbol distribution $Q(X)$.

Now, KL divergence is the penalty we pay, as measured in average number of bits, for using the optimal scheme for $Q(X)$, under the scenario where symbols are actually distributed as $P(X)$. It is straightforward to see this

$$\begin{aligned} D_{KL}(P\|Q) &= \sum_{x \in \mathcal{X}} P(x) \log \frac{P(x)}{Q(x)} \\ &= \sum_{x \in \mathcal{X}} P(x) \log P(x) - \sum_{x \in \mathcal{X}} P(x) \log Q(x) \\ &= H(P, Q) - H(P). \quad \text{(difference in average number of bits.)} \end{aligned}$$

If the cross entropy between P and Q is $H(P)$ (and hence $D_{KL}(P||Q) = 0$) then it necessarily means $P = Q$. In Machine Learning, it is a common task to find a distribution Q that is "close" to another distribution P. To achieve this, it is common to use $D_{KL}(Q||P)$ as the loss function to be optimized. As we will see in this question below, Maximum Likelihood Estimation, which is a commonly used optimization objective, turns out to be equivalent to minimizing the KL divergence between the training data (i.e. the empirical distribution over the data) and the model.

Now, we get back to showing some simple properties of KL divergence.

(a) [5 points] **Nonnegativity.**
Prove the following:

$$\forall P, Q. \quad D_{KL}(P\|Q) \geq 0$$

and

$$D_{KL}(P\|Q) = 0 \qquad \text{if and only if} \qquad P = Q.$$

[Hint: You may use the following result, called **Jensen's inequality**. If f is a convex function, and X is a random variable, then $E[f(X)] \geq f(E[X])$. Moreover, if f is strictly convex (f is convex if its Hessian satisfies $H \geq 0$; it is *strictly* convex if $H > 0$; for instance $f(x) = -\log x$ is strictly convex), then $E[f(X)] = f(E[X])$ implies that $X = E[X]$ with probability 1; i.e., X is actually a constant.]

(b) [5 points] **Chain rule for KL divergence.**
The KL divergence between 2 conditional distributions $P(X|Y), Q(X|Y)$ is defined as follows:

$$D_{KL}(P(X|Y)\|Q(X|Y)) = \sum_y P(y)\left(\sum_x P(x|y)\log\frac{P(x|y)}{Q(x|y)}\right)$$

This can be thought of as the expected KL divergence between the corresponding conditional distributions on x (that is, between $P(X|Y = y)$ and $Q(X|Y = y)$), where the expectation is taken over the random y.

Prove the following chain rule for KL divergence:

$$D_{KL}(P(X,Y)\|Q(X,Y)) = D_{KL}(P(X)\|Q(X)) + D_{KL}(P(Y|X)\|Q(Y|X)).$$

(c) [5 points] **KL and maximum likelihood.**
Consider a density estimation problem, and suppose we are given a training set $\{x^{(i)}; i = 1, \ldots, n\}$. Let the empirical distribution be $\hat{P}(x) = \frac{1}{n}\sum_{i=1}^{n} 1\{x^{(i)} = x\}$. ($\hat{P}$ is just the uniform distribution over the training set; i.e., sampling from the empirical distribution is the same as picking a random example from the training set.)

Suppose we have some family of distributions P_θ parameterized by θ. (If you like, think of $P_\theta(x)$ as an alternative notation for $P(x;\theta)$.) Prove that finding the maximum likelihood estimate for the parameter θ is equivalent to finding P_θ with minimal KL divergence from $\hat{P}$. I.e. prove:

$$\arg\min_\theta D_{KL}(\hat{P}\|P_\theta) = \arg\max_\theta \sum_{i=1}^{n} \log P_\theta(x^{(i)})$$

Remark. Consider the relationship between parts (b-c) and multi-variate Bernoulli Naive Bayes parameter estimation. In the Naive Bayes model we assumed P_θ is of the following form: $P_\theta(x, y) = p(y)\prod_{i=1}^{d} p(x_i|y)$. By the chain rule for KL divergence, we therefore have:

$$D_{KL}(\hat{P}\|P_\theta) = D_{KL}(\hat{P}(y)\|p(y)) + \sum_{i=1}^{d} D_{KL}(\hat{P}(x_i|y)\|p(x_i|y)).$$

This shows that finding the maximum likelihood/minimum KL-divergence estimate of the parameters decomposes into $2n+1$ independent optimization problems: One for the class priors $p(y)$, and one for each of the conditional distributions $p(x_i|y)$ for each feature x_i given each of the two possible labels for y. Specifically, finding the maximum likelihood estimates for each of these problems individually results in also maximizing the likelihood of the joint distribution. (If you know what Bayesian networks are, a similar remark applies to parameter estimation for them.)

3. **[8 points] Model Calibration.** In this question, we will discuss the uncertainty qualification of the machine learning models, and, in particular, a specific notion of uncertainty quantification called calibration. In many real-world applications such as policy-making or health care, prediction by machine learning models is not the end goal. Such predictions are used as the input to decision making process which needs to balances risks and rewards, both of which are often uncertain. Therefore, machine learning models not only need to produce a single prediction for each example, but also to measure the uncertainty of the prediction.

In this question we consider the binary classification setting. Let $\mathcal{X}$ be the input space and $\mathcal{Y} = \{0, 1\}$ be the label space. For simplicity, we assume that $\mathcal{X}$ is a discrete space of finite number of elements. Let X and Y be random variables denoting the input and label, respectively, over $\mathcal{X}$ and $\mathcal{Y}$, respectively. Let X, Y have a joint distribution $\mathcal{P}$. Suppose we have a model $h : \mathcal{X} \to [0, 1]$ (here we dropped the dependency on the parameters θ because the parameterization is not relevant to the discussion.) Recall that in logistical regression, we assume that there exists some model $h^\star(\cdot)$ (parameterized by a certain form, e.g., linear functions or neural networks) such that the output of the model $h^\star$ represents the conditional probability of Y being 1 given X:

$$P\left[Y = 1|X = x\right] = h^\star(x) \tag{1}$$

Under this assumption, we can derive the logistic loss to train and obtain some model $h(\cdot)$. Recall that the decision boundary is typically set to correspond to $h(x) = 1/2$, which means that the prediction of x is 1 if $h(x) \geq 1/2$, and 0 if $h(x) < 1/2$. To quantify the uncertainty of the prediction, it's tempting to just use the value of $h(x)$ — the closer $h(x)$ is to 0 or 1, the more confident we are with our prediction.

How do we know whether the learned model $h(\cdot)$ indeed outputs a reliable and truthful probability $h(x)$? We note that we shouldn't take it for granted because a) the assumption (1) may not exactly be satisfied by the data, and b) even if the assumption (1) holds perfectly, the learned model $h(x)$ may be different from the true model $h^\star$ that satisfies (1).

We will introduce a metric to evaluate how reliably the probabilities output by a model h capture the confidence of the model. In order for these probabilities to be useful as confidence measures, we would ideally like them to be *calibrated* in the following sense. Calibration intuitively requires that among all the examples for which the model predicts the value 0.7, indeed 70% of them should have label 1.

Formally, we say that a model h is perfectly calibrated if for all possible probabilities $p \in [0, 1]$ such that $P[h(X) = p] > 0$,

$$P[Y = 1 \mid h(X) = p] = p. \tag{2}$$

Recall that (X, Y) is a random draw from the (population) data distribution.

In the example in Table 1, the model h is not perfectly calibrated, because for $p = 0.3$,

$$\begin{aligned}
P[Y = 1 \mid h(X) = 0.3] &= P[Y = 1 \mid X = 0 \text{ or } 1] \\
&= \frac{P[Y = 1 \text{ and } (X = 0 \text{ or } 1)]}{P[X = 0 \text{ or } 1]} \\
&= \frac{P[Y = 1 \text{ and } X = 0] + P[Y = 1 \text{ and } X = 1]}{P[X = 0 \text{ or } 1]} \\
&= \frac{P[Y = 1 \mid X = 0]P[X = 0] + P[Y = 1 \mid X = 1]P[X = 1]}{P[X = 0 \text{ or } 1]} \\
&= \frac{0.2 \times 0.25 + 0.0 \times 0.25}{0.5} = 0.1 \neq 0.3
\end{aligned}$$

x	$P[X = x]$	$P[Y = 1 \mid X = x]$	$h(x)$	$T(x) = \mathbb{E}[Y \mid h(X) = h(x)]$
0	0.25	0.2	0.3	0.1
1	0.25	0.0	0.3	0.1
2	0.25	1.0	0.9	0.9
3	0.25	0.8	0.9	0.9

Table 1: An example of a model $h : \mathcal{X} = \{0, 1, 2, 3\} \to [0, 1]$. For calculating $T(x)$, we look at all data points with the same score $h(x)$, and compute the probability of $Y = 1$ for these data points.

(a) [3 points] Show that the perfect calibration does not necessarily imply that the model achieves perfect accuracy. Is the converse necessarily true? Justify your answers by providing either a proof or a counterexample. (Note that perfect accuracy means that $P[\mathbb{I}[h(X) \geq 0.5] = Y] = 1$ [2].)

(b) [5 points] As you showed in the last part, calibration by itself does not necessarily guarantee good accuracy. Good models must also be sharp, i.e., the probabilities output by the model should be close to 0 or 1. Mean squared error (MSE) is a common measure for evaluating the quality of a model.

$$\text{MSE}(h) = \mathbb{E}\left[(Y - h(X))^2\right] \tag{3}$$

In this part, we will show that the MSE can be decomposed to two parts, such that one part corresponds to the calibration error, and the other part corresponds to the sharpness of the model.

Formally, let $T(x) = P[Y = 1 \mid h(X) = h(x)]$ denote the true probability of $Y = 1$ given that the prediction is equal to $h(x)$. Intuitively, for a data point x, $T(x)$ is the probability of $Y = 1$ for all the data points with the same score as x. See Table 1 for an example.

Define calibration error CE to be[3]:

$$\text{CE}(h) = \mathbb{E}[(T(X) - h(X))^2] \tag{4}$$

The calibration error here is a quantitative instantiation of the notion of perfect calibration in equation (2). Indeed, zero calibration error implies perfect calibration: zero calibration means the model perfectly predicts the true probability, i.e., $h(X) = T(X)$ w.p. 1. This in turns implies that the model is perfectly calibrated because for any p such that $P[h(X) = p] > 0$, we can take some x_0 such that $h(x_0) = p$ and $h(x_0) = T(x_0)$, and conclude

$$\begin{aligned} P[Y = 1 \mid h(X) = p] &= P[Y = 1 \mid h(X) = h(x_0)] && (5) \\ &= T(x_0) = h(x_0) && \text{(by calibration error = 0)} \\ &= p && \text{(by the assumption that } h(x_0) = p) \end{aligned}$$

As explained before, we want our model prediction to be sharp as well. One way to define sharpness of a model is to look at the variance of $T(X)$. Let's define sharpness of model h as follows:

$$\text{SH}(h) = \text{Var}(T(X)) \tag{6}$$

[2] $\mathbb{I}$ is the indicator function. $\mathbb{I}[h(x) \geq 0.5]$ is equal to 1 if $h(x) \geq 0.5$ and is equal to 0 if $h(x) < 0.5$.

[3] There are other definition of calibration errors, e.g., the one introduced in the next part.

The sharpness term measures how much variation there is in the true probability across model prediction. It is very small when $T(x)$ is similar for all data points.
MSE can be decomposed as follows:

$$\text{MSE}(h) = \underbrace{\text{Var}[Y]}_{\text{Instrinsic uncertainty}} - \underbrace{\text{Var}[T(X)]}_{\text{Sharpness}} + \underbrace{\mathbb{E}\left[(T(X) - h(X))^2\right]}_{\text{Calibration error}} \tag{7}$$

This decomposition states that by minimizing MSE we try to find a sharp model with small calibration error. In other words, when we choose a model with minimum MSE it means between two models with the same calibration error we prefer the one that is sharper and between two models with the same sharpness we prefer the one with lower calibration error. Note that the uncertainty term does not depend on the model and can be mostly ignored.
Prove that the decomposition in Equation (7) **is correct.**

(c) [0 points] [**This part is optional and does not have any points**] In the previous part, we studied MSE and decomposed it to two terms corresponding to the sharpness and calibration error. But as we explained, there are other different ways to measure the calibration and sharpness of a model. In this part we focus on logistic regression models. In particular, we are going to show logistic loss can also be decomposed to two terms; where one term can be interpreted as the sharpness of the prediction (which we call log-sharpness) and the other can be interpreted as the calibration error (which we call log-calibration-error). Recall that the logistic loss (on population) of the model h is defined as:

$$\text{Log-Loss}(h) = \mathbb{E}[-Y\log(h(X)) - (1-Y)\log(1-h(X))] \tag{8}$$

Prove that logistic loss can be decomposed to two terms as follows:

$$\begin{aligned}\text{Log-Loss}(h) =& \underbrace{\mathbb{E}\left[T(X)\log\left(\frac{T(X)}{h(X)}\right) + (1-T(X))\log\left(\frac{1-T(X)}{1-h(X)}\right)\right]}_{\text{log-calibration-error}} \\ &- \underbrace{\mathbb{E}\left[T(X)\log(T(X)) + (1-T(X))\log(1-T(X))\right]}_{\text{log-sharpness}}\end{aligned} \tag{9}$$

Discuss why the log-calibration-error term in (9) is a meaningful term for measuring the calibration error and specify when it attains its minimum. Similarly discuss why log-sharpness term in (9) is a meaningful term for measuring the sharpness of a model and specify when it attains its maximum.

Hint. For showing that log-calibration-error and log-sharpness are meaningful terms, you should use your information theory knowledge (there is a section about information theory in the previous question). For each data point x, both model prediction ($h(x)$) and underlying probability ($T(x)$) define a distribution over the label set $\mathcal{Y} = \{0, 1\}$. In particular, define distribution P_1 on $\mathcal{Y} = \{0, 1\}$ as follows: $P_1(Y = 1) = T(x)$ and $P_1(Y = 0) = 1 - T(x)$. Similarly, define distribution P_2 on $\mathcal{Y} = \{0, 1\}$ as follows: $P_2(Y = 1) = h(x)$ and $P_2(Y = 0) = 1 - h(x)$. You can interpret the log-calibration-error in (9) as KL-divergence distance between these two distributions. Recall that the KL divergence distance between these two distributions is:

$$D_{\text{KL}}(P_1 \parallel P_2) = P_1(Y=0)\log\left(\frac{P_1(Y=0)}{P_2(Y=0)}\right) + P_1(Y=1)\log\left(\frac{P_1(Y=1)}{P_2(Y=1)}\right) \tag{10}$$

The log-sharpness term in (9) can be expressed as the negative entropy of the distribution corresponding to $T(x)$. Recall that entropy of distribution P_1 is:

$$\mathrm{H}(P_1) = -P_1(Y=0)\log(P_1(Y=0)) - P_1(Y=1)\log(P_1(Y=1)) \tag{11}$$

Remark: The decomposition suggests that minimizing the logistic loss (on the population) tends to minimize the calibration error as well, since the calibration error is upper bounded by the logistic loss. In practice, when the train and test sets are from the same distribution and when the model has not overfit or underfit, logistic regression tends to be well calibrated on the test data as well. In contrast, modern large-scale deep learning models trained with the logistic loss are typically not well-calibrated, likely since the population logistic loss suffers from overfitting (the test loss is much higher than the train loss), even when there is little overfitting in terms of the accuracy. As such, often people use other recalibration methods to adjust the outputs of a deep learning model to be better calibrated.

4. **[20 points] K-means for compression**

In this problem, we will apply the K-means algorithm to lossy image compression, by reducing the number of colors used in an image.

We will be using the files `src/k_means/peppers-small.tiff` and `src/k_means/peppers-large.tiff`.

The `peppers-large.tiff` file contains a 512x512 image of peppers represented in 24-bit color. This means that, for each of the 262144 pixels in the image, there are three 8-bit numbers (each ranging from 0 to 255) that represent the red, green, and blue intensity values for that pixel. The straightforward representation of this image therefore takes about $262144 \times 3 = 786432$ bytes (a byte being 8 bits). To compress the image, we will use K-means to reduce the image to $k = 16$ colors. More specifically, each pixel in the image is considered a point in the three-dimensional (r, g, b)-space. To compress the image, we will cluster these points in color-space into 16 clusters, and replace each pixel with the closest cluster centroid.

Follow the instructions below. Be warned that some of these operations can take a while (several minutes even on a fast computer)!

(a) [15 points] **[Coding Problem] K-Means Compression Implementation.** First let us *look* at our data. From the `src/k_means/` directory, open an interactive Python prompt, and type

```
from matplotlib.image import imread; import matplotlib.pyplot as plt;
```

and run `A = imread('peppers-large.tiff')`. Now, `A` is a "three dimensional matrix," and `A[:,:,0]`, `A[:,:,1]` and `A[:,:,2]` are 512x512 arrays that respectively contain the red, green, and blue values for each pixel. Enter `plt.imshow(A); plt.show()` to display the image.

Since the large image has 262,144 pixels and would take a while to cluster, we will instead run vector quantization on a smaller image. Repeat (a) with `peppers-small.tiff`.

Next we will implement image compression in the file `src/k_means/k_means.py` which has some starter code. Treating each pixel's (r, g, b) values as an element of $\mathbb{R}^3$, implement K-means with 16 clusters on the pixel data from this smaller image, iterating (preferably) to convergence, but in no case for less than 30 iterations. For initialization, set each cluster centroid to the (r, g, b)-values of a randomly chosen pixel in the image.

Take the image of `peppers-large.tiff`, and replace each pixel's (r, g, b) values with the value of the closest cluster centroid from the set of centroids computed with `peppers-small.tiff`. Visually compare it to the original image to verify that your implementation is reasonable. **Include in your write-up a copy of this compressed image alongside the original image.**

(b) [5 points] **Compression Factor.**

If we represent the image with these reduced (16) colors, by (approximately) what factor have we compressed the image?

5. **[35 points] Semi-supervised EM**

Expectation Maximization (EM) is a classical algorithm for unsupervised learning (*i.e.,* learning with hidden or latent variables). In this problem we will explore one of the ways in which EM algorithm can be adapted to the semi-supervised setting, where we have some labeled examples along with unlabeled examples.

In the standard unsupervised setting, we have $n \in \mathbb{N}$ unlabeled examples $\{x^{(1)}, \ldots, x^{(n)}\}$. We wish to learn the parameters of $p(x, z; \theta)$ from the data, but $z^{(i)}$'s are not observed. The classical EM algorithm is designed for this very purpose, where we maximize the intractable $p(x; \theta)$ indirectly by iteratively performing the E-step and M-step, each time maximizing a tractable lower bound of $p(x; \theta)$. Our objective can be concretely written as:

$$\begin{aligned}\ell_{\text{unsup}}(\theta) &= \sum_{i=1}^{n} \log p(x^{(i)}; \theta) \\ &= \sum_{i=1}^{n} \log \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}; \theta)\end{aligned}$$

Now, we will attempt to construct an extension of EM to the semi-supervised setting. Let us suppose we have an *additional* $\tilde{n} \in \mathbb{N}$ labeled examples $\{(\tilde{x}^{(1)}, \tilde{z}^{(1)}), \ldots, (\tilde{x}^{(\tilde{n})}, \tilde{z}^{(\tilde{n})})\}$ where both x and z are observed. We want to simultaneously maximize the marginal likelihood of the parameters using the unlabeled examples, and full likelihood of the parameters using the labeled examples, by optimizing their weighted sum (with some hyperparameter α). More concretely, our semi-supervised objective $\ell_{\text{semi-sup}}(\theta)$ can be written as:

$$\begin{aligned}\ell_{\text{sup}}(\theta) &= \sum_{i=1}^{\tilde{n}} \log p(\tilde{x}^{(i)}, \tilde{z}^{(i)}; \theta) \\ \ell_{\text{semi-sup}}(\theta) &= \ell_{\text{unsup}}(\theta) + \alpha \ell_{\text{sup}}(\theta)\end{aligned}$$

We can derive the EM steps for the semi-supervised setting using the same approach and steps as before. You are *strongly encouraged* to show to yourself (no need to include in the write-up) that we end up with:

E-step (semi-supervised)

For each $i \in \{1, \ldots, n\}$, set

$$Q_i^{(t)}(z^{(i)}) := p(z^{(i)} | x^{(i)}; \theta^{(t)})$$

M-step (semi-supervised)

$$\theta^{(t+1)} := \arg\max_{\theta} \left[\sum_{i=1}^{n} \left(\sum_{z^{(i)}} Q_i^{(t)}(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i^{(t)}(z^{(i)})} \right) + \alpha \left(\sum_{i=1}^{\tilde{n}} \log p(\tilde{x}^{(i)}, \tilde{z}^{(i)}; \theta) \right) \right]$$

(a) [5 points] **Convergence.** First we will show that this algorithm eventually converges. In order to prove this, it is sufficient to show that our semi-supervised objective $\ell_{\text{semi-sup}}(\theta)$ monotonically increases with each iteration of E and M step. Specifically, let $\theta^{(t)}$ be the parameters obtained at the end of t EM-steps. Show that $\ell_{\text{semi-sup}}(\theta^{(t+1)}) \geq \ell_{\text{semi-sup}}(\theta^{(t)})$.

Semi-supervised GMM

Now we will revisit the Gaussian Mixture Model (GMM), to apply our semi-supervised EM algorithm. Let us consider a scenario where data is generated from $k \in \mathbb{N}$ Gaussian distributions, with unknown means $\mu_j \in \mathbb{R}^d$ and covariances $\Sigma_j \in \mathbb{S}_+^d$ where $j \in \{1, \ldots, k\}$. We have n data points $x^{(i)} \in \mathbb{R}^d, i \in \{1, \ldots, n\}$, and each data point has a corresponding latent (hidden/unknown) variable $z^{(i)} \in \{1, \ldots, k\}$ indicating which distribution $x^{(i)}$ belongs to. Specifically, $z^{(i)} \sim \text{Multinomial}(\phi)$, such that $\sum_{j=1}^{k} \phi_j = 1$ and $\phi_j \geq 0$ for all j, and $x^{(i)}|z^{(i)} \sim \mathcal{N}\left(\mu_{z^{(i)}}, \Sigma_{z^{(i)}}\right)$ i.i.d. So, μ, Σ, and ϕ are the model parameters.

We also have additional $\tilde{n}$ data points $\tilde{x}^{(i)} \in \mathbb{R}^d, i \in \{1, \ldots, \tilde{n}\}$, and an associated *observed* variable $\tilde{z}^{(i)} \in \{1, \ldots, k\}$ indicating the distribution $\tilde{x}^{(i)}$ belongs to. Note that $\tilde{z}^{(i)}$ are known constants (in contrast to $z^{(i)}$ which are unknown *random* variables). As before, we assume $\tilde{x}^{(i)}|\tilde{z}^{(i)} \sim \mathcal{N}\left(\mu_{\tilde{z}^{(i)}}, \Sigma_{\tilde{z}^{(i)}}\right)$ i.i.d.

In summary we have $n + \tilde{n}$ examples, of which n are unlabeled data points x's with unobserved z's, and $\tilde{n}$ are labeled data points $\tilde{x}^{(i)}$ with corresponding observed labels $\tilde{z}^{(i)}$. The traditional EM algorithm is designed to take only the n unlabeled examples as input, and learn the model parameters μ, Σ, and ϕ.

Our task now will be to apply the semi-supervised EM algorithm to GMMs in order to also leverage the additional $\tilde{n}$ labeled examples, and come up with semi-supervised E-step and M-step update rules specific to GMMs. Whenever required, you can cite the lecture notes for derivations and steps.

(b) [5 points] **Semi-supervised E-Step.** Clearly state which are all the latent variables that need to be re-estimated in the E-step. Derive the E-step to re-estimate all the stated latent variables. Your final E-step expression must only involve x, z, μ, Σ, ϕ and universal constants.

(c) [10 points] **Semi-supervised M-Step.** Clearly state which are all the parameters that need to be re-estimated in the M-step. Derive the M-step to re-estimate all the stated parameters. Specifically, derive closed form expressions for the parameter update rules for $\mu^{(t+1)}$, $\Sigma^{(t+1)}$ and $\phi^{(t+1)}$ based on the semi-supervised objective.

(d) [5 points] **Classical (Unsupervised) EM Implementation.** For this sub-question, we are only going to consider the n unlabelled examples. Follow the instructions in `src/semi_supervised_em/gmm.py` to implement the traditional EM algorithm, and run it on the unlabelled data-set until convergence.

Run three trials and use the provided plotting function to construct a scatter plot of the resulting assignments to clusters (one plot for each trial). Your plot should indicate cluster assignments with colors they got assigned to (*i.e.,* the cluster which had the highest probability in the final E-step).

Submit the three plots obtained above in your write-up.

(e) [7 points] **Semi-supervised EM Implementation.** Now we will consider both the labelled and unlabelled examples (a total of $n + \tilde{n}$), with 5 labelled examples per cluster. We have provided starter code for splitting the dataset into matrices `x` and `x_tilde` of unlabelled and labelled examples respectively. Add to your code in `src/semi_supervised_em/gmm.py` to implement the modified EM algorithm, and run it on the dataset until convergence.

Create a plot for each trial, as done in the previous sub-question.

Submit the three plots obtained above in your write-up.

(f) [3 points] **Comparison of Unsupervised and Semi-supervised EM.** Briefly describe the differences you saw in unsupervised *vs.* semi-supervised EM for each of the following:

   i. Number of iterations taken to converge.
   ii. Stability (*i.e.,* how much did assignments change with different random initializations?)
   iii. Overall quality of assignments.

   Note: The dataset was sampled from a mixture of three low-variance Gaussian distributions, and a fourth, high-variance Gaussian distribution. This should be useful in determining the overall quality of the assignments that were found by the two algorithms.