CS 229, Fall 2022
Problem Set #3

Due Thursday, November 10 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 Ed forum, at `https://edstem.org/us/courses/30055/discussion/`.

(3) 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) The due date is Thursday, November 10 at 11:59 pm. If you submit after Thursday, November 10 at 11:59 pm, you will begin consuming your late days. The late day policy can be found in the course website: Course Logistics and FAQ.

All students must submit an electronic PDF version of the written question including plots generated from the codes. 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. Please make sure that your PDF file and zip file are submitted to the corresponding Gradescope assignments respectively. We reserve the right to not give any points to the written solutions if the associated code is not submitted.

Honor code: We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solution independently, and without referring to written notes from the joint session. Each student must understand the solution well enough in order to reconstruct it by him/herself. It is an honor code violation to copy, refer to, or look at written or code solutions from a previous year, including but not limited to: official solutions from a previous year, solutions posted online, and solutions you or someone else may have written up in a previous year. Furthermore, it is an honor code violation to post your assignment solutions online, such as on a public git repo. We run plagiarism-detection software on your code against past solutions as well as student submissions from previous years. Please take the time to familiarize yourself with the Stanford Honor Code[1] and the Stanford Honor Code[2] as it pertains to CS courses.

[1] https://communitystandards.stanford.edu/policies-and-guidance/honor-code
[2] https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1164/handouts/honor-code.pdf

1. **[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[3]:

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

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 to 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 $\{\mathrm{a}, \mathrm{b}, \ldots, \mathrm{z}\}$). 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

[3] 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

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

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 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.)

2. **[35 points] Implicit Regularization**

Recall that in the overparameterized regime (where the number of parameters is larger than the number of samples), typically there are infinitely many solutions that can fit the training dataset perfectly, and many of them cannot generalize well (that is, they have large validation errors). However, in many cases, the particular optimizer we use (e.g., GD, SGD with particular learning rates, batch sizes, noise, etc.) tends to find solutions that generalize well. This phenomenon is called implicit regularization effect (also known as algorithmic regularization or implicit bias).

In this problem, we will look at the implicit regularization effect on two toy examples in the overparameterized regime: linear regression and a quadratically parameterized model. For linear regression, we will show that gradient descent with zero initialization will always find the minimum norm solution (instead of an arbitrary solution that fits the training data), and in practice, the minimum norm solution tends to generalize well. For a quadratically parameterized model, we will show that initialization and batch size also affect generalization.

(a) [3 points] Suppose we have a dataset $\{(x^{(i)}, y^{(i)}); i = 1, \cdots, n\}$ where $x^{(i)} \in \mathbb{R}^d$ and $y^{(i)} \in \mathbb{R}$ for all $1 \leq i \leq n$. We assume the dataset is generated by a linear model without noise. That is, there is a vector $\beta^\star \in \mathbb{R}^d$ such that $y^{(i)} = (\beta^\star)^\top x^{(i)}$ for all $1 \leq i \leq n$. Let $X \in \mathbb{R}^{n \times d}$ be the matrix representing the inputs (i.e., the i-th row of X corresponds to $x^{(i)}$)) and $\vec{y} \in \mathbb{R}^n$ the vector representing the labels (i.e., the i-th row of $\vec{y}$ corresponds to $y^{(i)}$)):

$$X = \begin{bmatrix} - & x^{(1)} & - \\ - & x^{(2)} & - \\ \vdots & \vdots & \vdots \\ - & x^{(n)} & - \end{bmatrix}, \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}.$$

Then in matrix form, we can write $\vec{y} = X\beta^\star$. We assume that the number of examples is less than the number of parameters (that is, $n < d$).

We use the least-squares cost function to train a linear model:

$$J(\beta) = \frac{1}{2n}\|X\beta - \vec{y}\|_2^2. \tag{1}$$

In this sub-question, we characterize the family of global minimizers to Eq. (1). We assume that $XX^\top \in \mathbb{R}^{n \times n}$ is an invertible matrix. **Prove that** β achieves zero cost in Eq. (1) if and only if

$$\beta = X^\top (XX^\top)^{-1}\vec{y} + \zeta \tag{2}$$

for some ζ in the subspace orthogonal to all the data (that is, for some ζ such that $\zeta^\top x^{(i)} = 0, \forall 1 \leq i \leq n$.)

Note that this implies that there is an infinite number of β's such that Eq. (1) is minimized. We also note that $X^\top (XX^\top)^{-1}$ is the pseudo-inverse of X, but you don't necessarily need this fact for the proof.

(b) [3 points] We still work with the setup of part (a). Among the infinitely many optimal solutions of Eq. (1), we consider the *minimum norm* solution. Let $\rho = X^\top (XX^\top)^{-1}\vec{y}$. In the setting of (a), **prove that** for any β such that $J(\beta) = 0$, $\|\rho\|_2 \leq \|\beta\|_2$. In other words, ρ is the minimum norm solution.

Hint: As a intermediate step, you can prove that for any β in the form of Eq. (2),

$$\|\beta\|_2^2 = \|\rho\|_2^2 + \|\zeta\|_2^2.$$

(c) [5 points] **Coding question: minimum norm solution generalizes well**
For this sub-question, we still work with the setup of parts (a) and (b). We use the following datasets:

`src/implicitreg/ds1_train.csv,ds1_valid.csv`

Each file contains $d+1$ columns. The first d columns in the i-th row represents $x^{(i)}$, and the last column represents $y^{(i)}$. In this sub-question, we use $d = 200$ and $n = 40$.
Using the formula in sub-question (b), **compute** the minimum norm solution using the training dataset. Then, **generate** three other different solutions with zero costs and different norms using the formula in sub-question (a). The starter code is in `src/implicitreg/linear.py`. **Plot** the validation error of these solutions (including the minimum norm solution) in a scatter plot. Use the norm of the solutions as x-axis, and the validation error as y-axis. For your convenience, the plotting function is provided as the method `generate_plot` in the starter code. Your plot is expected to demonstrate that the minimum norm solution generalizes well.

(d) [5 points] For this sub-question, we work with the setup of part (a) and (b). In this sub-question, you will prove that the gradient descent algorithm with *zero initialization* always converges to the minimum norm solution. Let $\beta^{(t)}$ be the parameters found by the GD algorithm at time step t. Recall that at step t, the gradient descent algorithm update the parameters in the following way

$$\beta^{(t)} = \beta^{(t-1)} - \eta \nabla J(\beta^{(t-1)}) = \beta^{(t-1)} - \frac{\eta}{n} X^\top (X\beta^{(t-1)} - \vec{y}). \tag{3}$$

As in sub-question (a), we also assume $XX^\top$ is an invertible matrix. **Prove** that if the GD algorithm with zero initialization converges to a solution $\hat{\beta}$ satisfying $J(\hat{\beta}) = 0$, then $\hat{\beta} = X^\top (XX^\top)^{-1}\vec{y} = \rho,$, that is, $\hat{\beta}$ is the minimum norm solution.
Hint: As a first step, you can prove by induction that if we start with zero initialization, $\beta^{(t)}$ will always be a linear combination of $\{x^{(1)}, x^{(2)}, \cdots, x^{(n)}\}$ for any $t \geq 0$. Then, for any $t \geq 0$, you can write $\beta^{(t)} = X^\top v^{(t)}$ for some $v^{(t)} \in \mathbb{R}^n$. As a second step, you can prove that if $\hat{\beta} = X^\top v^{(t)}$ for some $v^{(t)}$ and $J(\hat{\beta}) = 0$, then we have $\hat{\beta} = \rho$.
You don't necessarily have to follow the steps in this hint. But if you use the hint, you need to prove the statements in the hint.

(e) [3 points] In the following sub-questions, we consider a slightly more complicated model called quadratically parameterized model. A quadratically parameterized model has two sets of parameters $\theta, \phi \in \mathbb{R}^d$. Given a d-dimensional input $x \in \mathbb{R}^d$, the output of the model is

$$f_{\theta,\phi}(x) = \sum_{k=1}^{d} \theta_k^2 x_k - \sum_{k=1}^{d} \phi_k^2 x_k. \tag{4}$$

Note that $f_{\theta,\phi}(x)$ is linear in its input x, but non-linear in its parameters θ, ϕ. Thus, if the goal was to learn the function, one should simply just re-parameterize it with a linear model and use linear regression. However, here we insist on using the parameterization above in Eq. (4) in order to study the implicit regularization effect in models that are nonlinear in the parameters.
Notations: To simplify the equations, we define the following notations. For a vector $v \in \mathbb{R}^d$, let $v^{\odot 2}$ be its element-wise square (that is, $v^{\odot 2}$ is the vector $[v_1^2, v_2^2, \cdots, v_d^2] \in \mathbb{R}^d$.) For two

vectors $v, w \in \mathbb{R}^d$, let $v \odot w$ be their element-wise product (that is, $v \odot w$ is the vector $[v_1 w_1, v_2 w_2, \cdots, v_d w_d] \in \mathbb{R}^d$.) Then our model can be written as

$$f_{\theta,\phi}(x) = x^\top(\theta^{\odot 2} - \phi^{\odot 2}). \tag{5}$$

Suppose we have a dataset $\{(x^{(i)}, y^{(i)}); i = 1, \cdots, n\}$ where $x^{(i)} \in \mathbb{R}^d$ and $y^{(i)} \in \mathbb{R}$ for all $1 \le i \le n$, and

$$y^{(i)} = (x^{(i)})^\top((\theta^\star)^{\odot 2} - (\phi^\star)^{\odot 2})$$

for some $\theta^\star, \phi^\star \in \mathbb{R}^d$. Similarly, we use $X \in \mathbb{R}^{n \times d}$ and $\vec{y} \in \mathbb{R}^n$ to denote the matrix/vector representing the inputs/labels respectively:

$$X = \begin{bmatrix} - & x^{(1)} & - \\ - & x^{(2)} & - \\ \vdots & \vdots & \vdots \\ - & x^{(n)} & - \end{bmatrix}, \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}.$$

Let $J(\theta, \phi) = \frac{1}{4n}\sum_{i=1}^n (f_{\theta,\phi}(x^{(i)}) - y^{(i)})^2$ be the cost function.

First, when $n < d$ and $XX^\top$ is invertible, **prove** that there exists infinitely many optimal solutions with zero cost.

Hint: Find a mapping between the parameter β in linear model and the parameter θ, ϕ in quadratically parameterized model. Then use the conclusion in sub-question (a).

(f) [10 points] **Coding question: implicit regularization of initialization**

We still work with the setup in part (e). For this sub-question, we use the following datasets:

`src/implicitreg/ds2_train.csv,ds2_valid.csv`

Each file contains $d + 1$ columns. The first d columns in the i-th row represents $x^{(i)}$, and the last column represents $y^{(i)}$. In this sub-question, we use $d = 200$ and $n = 40$.

First of all, the gradient of the loss has the following form:

$$\nabla_\theta J(\theta, \phi) = \frac{1}{n}\sum_{i=1}^n ((x^{(i)})^\top(\theta^{\odot 2} - \phi^{\odot 2}) - y^{(i)})(\theta \odot x^{(i)}), \tag{6}$$

$$\nabla_\phi J(\theta, \phi) = -\frac{1}{n}\sum_{i=1}^n ((x^{(i)})^\top(\theta^{\odot 2} - \phi^{\odot 2}) - y^{(i)})(\phi \odot x^{(i)}). \tag{7}$$

You don't need to prove these two equations. They can be verified directly using the chain rule.

Using the formula above, run gradient descent with initialization $\theta = \alpha\mathbf{1}, \phi = \alpha\mathbf{1}$ with $\alpha \in \{0.1, 0.03, 0.01\}$ (where $\mathbf{1} = [1, 1, \cdots, 1] \in \mathbb{R}^d$ is the all-1's vector) and learning rate 0.08. We provide the starter code in `src/implicitreg/qp.py`. **Plot** the curve of training error and validation error with different α. Use the number of gradient steps as x-axis, and training/validation error as y-axis. Include your plot in the writeup and **answer** the following two questions based on your plot: which models can fit the training set? Which initialization achieves the best validation error?

Remark: Your plot is expected to demonstrate that the initialization plays an important role in the generalization performance—different initialization can lead to different global minimizers with different generalization performance. In other words, the initialization has an implicit regularization effect.

(g) [6 points] **Coding question: implicit regularization of batch size**

We still work with the setup in part (e). For this sub-question, we use the same dataset and starter code as in sub-question (f). We will show that the noise in the training process also induces implicit regularization. In particular, the noise introduced by *stochastic* gradient descent in this case helps generalization. **Implement** the SGD algorithm, and **plot** the training and validation errors with batch size $\{1, 5, 40\}$, learning rate 0.08, and initialization $\alpha = 0.1$. Similarly, use the number of gradient steps as x-axis, and training/validation error as y-axis. For simplicity, the code for selecting a batch of examples is already provided in the starter code. **Compare** the results with those in sub-question (g) with the same initialization. Does SGD find a better solution?

Your plot is expected to show that the stochasticity in the training process is also an important factor in the generalization performance — in our setting, SGD finds a solution that generalizes better. In fact, a conjecture is that stochasticity in the optimization process (such as the noise introduced by a small batch size) helps the optimizer to find a solution that generalizes better. This conjecture can be proved in some simplified cases, such as the quadratically parameterized model in this sub-question (adapted from the paper HaoChen et al., 2020), and can be observed empirically in many other cases.

3. **[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.

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?