CS 229, Fall 2018
Problem Set #3: Deep Learning & Unsupervised learning

Due Wednesday, Nov 14 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 `http://piazza.com/stanford/fall2018/cs229`. (3) If you missed the first lecture or are unfamiliar with the collaboration or honor code policy, please read the policy on Handout #1 (available from 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 listed on Gradescope is Nov 17 at 11:59 pm. If you submit after Nov 14, you will begin consuming your late days. If you wish to submit on time, submit before Nov 14 at 11:59 pm.

All students must submit an electronic PDF version of the written questions. We highly recommend typesetting your solutions via LaTeX. If you are scanning your document by cell phone, please check the Piazza forum for recommended scanning apps and best practices. All students must also submit a zip file of their source code to Gradescope, which should be created using the `make_zip.py` script. In order to pass the auto-grader tests, 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 when running `p04_gmm.py` and `p05_kmeans.py`. Your submission will be evaluated by the auto-grader using a private test set.

1. **[20 points] A Simple Neural Network**

Let $X = \{x^{(1)}, \cdots, x^{(m)}\}$ be a dataset of m 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:

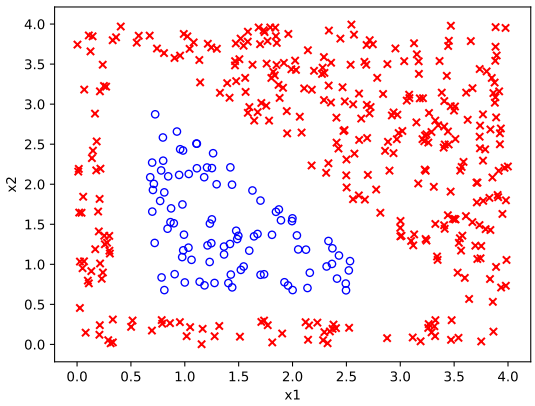

Figure 1: Plot of dataset X.

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

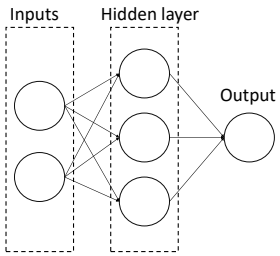

Figure 2: Architecture for our simple neural network.

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 $w_{i,j}^{[1]}$ for $i \in \{1, 2\}, j \in \{1, 2, 3\}$, and the weight from h_j to o be $w_j^{[2]}$. Finally, denote the intercept weight for h_j as $w_{0,j}^{[1]}$, and the intercept weight for o as $w_0^{[2]}$. For the loss function, we'll use average squared loss instead of the usual negative log-likelihood:

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

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 $w_{1,2}^{[1]}$, 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

$$f(x) = \begin{cases} 1, x \geq 0 \\ 0, x < 0 \end{cases}$$

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

If it is possible, please provide a set of weights that enable 100% accuracy by completing `optimal_step_weights` within `src/p01_nn.py` and explain your reasoning for those weights in your PDF.

If it is not possible, please explain your reasoning in your PDF. (There is no need to modify `optimal_step_weights` if it is not possible.)

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

(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 it is possible, please provide a set of weights that enable 100% accuracy by completing `optimal_linear_weights` within `src/p01_nn.py` and explain your reasoning for those weights in your PDF.

If it is not possible, please explain your reasoning in your PDF. (There is no need to modify `optimal_linear_weights` if it is not possible.)

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]:

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

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 $D_{\mathrm{KL}}(P||Q) = D_{\mathrm{KL}}(P(X)||Q(X))$.

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

[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 occured 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 achived 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

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

If the cross entropy between P and Q is zero (and hence $D_{\mathrm{KL}}(P||Q) = 0$) 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, we use $D_{\mathrm{KL}}(Q||P)$ to be 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 minimizing 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:

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

and

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

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:

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

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:

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

(c) [5 points] **KL and maximum likelihood.** Consider a density estimation problem, and suppose we are given a training set $\{x^{(i)}; i = 1, \ldots, m\}$. Let the empirical distribution be $\hat{P}(x) = \frac{1}{m} \sum_{i=1}^{m} 1\{x^{(i)} = x\}$. ($\hat{P}$ 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:

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

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: $P_\theta(x, y) = p(y) \prod_{i=1}^{n} p(x_i|y)$. By the chain rule for KL divergence, we therefore have:

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

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. **[25 points] KL Divergence, Fisher Information, and the Natural Gradient**

As seen before, the Kullback-Leibler divergence between two distributions is an asymmetric measure of how different two distributions are. Consider two distributions over the same space given by densities $p(x)$ and $q(x)$. The KL divergence between two continuous distributions, q and p is defined as,

$$\begin{aligned} D_{\mathrm{KL}}(p||q) &= \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx \\ &= \int_{-\infty}^{\infty} p(x) \log p(x) dx - \int_{-\infty}^{\infty} p(x) \log q(x) dx \\ &= \mathbb{E}_{x \sim p(x)}[\log p(x)] - \mathbb{E}_{x \sim p(x)}[\log q(x)]. \end{aligned}$$

A nice property of KL divergence is that it invariant to parametrization. This means, KL divergence evaluates to the same value no matter how we parametrize the distributions P and Q. For e.g, if P and Q are in the exponential family, the KL divergence between them is the same whether we are using natural parameters, or canonical parameters, or any arbitrary reparametrization.

Now we consider the problem of fitting model parameters using gradient descent (or stochastic gradient descent). As seen previously, fitting model parameters using Maximum Likelihood is equivalent to minimizing the KL divergence between the data and the model. While KL divergence is invariant to parametrization, the gradient w.r.t the model parameters (i.e, direction of steepest descent) is *not invariant to parametrization.* To see its implication, suppose we are at a particular value of parameters (either randomly initialized, or mid-way through the optimization process). The value of the parameters correspond to some probability distribution (and in case of regression, a conditional probability distribution). If we follow the direction of steepest descent from the current parameter, take a small step along that direction to a new parameter, we end up with a new distribution corresponding to the new parameters. The non-invariance to reparametrization means, a step of fixed size in the parameter space could end up in a distribution that could either be extremely far away in D_{KL} from the previous distribution, or on the other hand not move very much at all w.r.t D_{KL} from the previous distributions.

This is where the *natural gradient* comes into picture. It is best introduced in contrast with the usual gradient descent. In the usual gradient descent, we *first choose the direction* by calculating the gradient of the MLE objective w.r.t the parameters, and then move a magnitude of step size (where size is measured in the *parameter space*) along that direction. Whereas in natural gradient, we *first choose a divergence* amount by which we would like to move, in the D_{KL} sense. This effectively gives us a perimeter around the current parameters (of some arbitrary shape), such that points along this perimeter correspond to distributions which are at an equal D_{KL}-distance away from the current parameter. Among the set of all distributions along this perimeter, we move to the distribution that maximizes the objective (i.e minimize D_{KL} between data and itself) the most. This approach makes the optimization process invariant to parametrization. That means, even if we chose a new arbitrary reparametrization, by starting from a particular distribution, we always descend down the same sequence of distributions towards the optimum.

In the rest of this problem, we will construct and derive the natural gradient update rule. For that, we will break down the process into smaller sub-problems, and give you hints to answer them. Along the way, we will encounter important statistical concepts such as the *score function* and *Fisher Information* (which play a prominant role in Statistical Learning Theory as well). Finally, we will see how this new natural gradient based optimization is actually equivalent to Newton's method for Generalized Linear Models.

Let the distribution of a random variable Y parameterized by $\theta \in \mathbb{R}^n$ be $p(y;\theta)$.

(a) [3 points] **Score function**

The score function associated with $p(y;\theta)$ is defined as $\nabla_\theta \log p(y;\theta)$, which signifies the sensitivity of the likelihood function with respect to the parameters. Note that the score function is actually a vector since it's the gradient of a scalar quantity with respect to the vector θ.

Recall that $\mathbb{E}_{y\sim p(y)}[g(y)] = \int_{-\infty}^{\infty} p(y)g(y)dy$. Using this fact, show that the expected value of the score is 0, i.e.

$$\mathbb{E}_{y\sim p(y;\theta)}[\nabla_{\theta'} \log p(y;\theta')|_{\theta'=\theta}] = 0$$

(b) [2 points] **Fisher Information**

Let us now introduce a quantity known as the Fisher information. It is defined as the covariance matrix of the score function,

$$\mathcal{I}(\theta) = \mathrm{Cov}_{y\sim p(y;\theta)}[\nabla_{\theta'} \log p(y;\theta')|_{\theta'=\theta}]$$

Intuitively, the Fisher information represents the amount of information that a random variable Y carries about a parameter θ of interest. When the parameter of interest is a vector (as in our case, since $\theta \in \mathbb{R}^n$), this information becomes a matrix. Show that the Fisher information can equivalently be given by

$$\mathcal{I}(\theta) = \mathbb{E}_{y\sim p(y;\theta)}[\nabla_{\theta'} \log p(y;\theta') \nabla_{\theta'} \log p(y;\theta')^T |_{\theta'=\theta}]$$

Note that the Fisher Information is a function of the parameter. The parameter of the Fisher information is both a) the parameter value at which the score function is evaluated, and b) the parameter of the distribution with respect to which the expectation and variance is calculated.

(c) [5 points] **Fisher Information (alternate form)**

It turns out that the Fisher Information can not only be defined as the covariance of the score function, but in most situations it can also be represented as the expected negative Hessian of the log-likelihood.

Show that $\mathbb{E}_{y\sim p(y;\theta)}[-\nabla^2_{\theta'} \log p(y;\theta')|_{\theta'=\theta}] = \mathcal{I}(\theta)$.

Remark. The Hessian represents the curvature of a function at a point. This shows that the expected curvature of the log-likelihood function is also equal to the Fisher information matrix. If the curvature of the log-likelihood at a parameter is very steep (i.e, Fisher Information is very high), this generally means you need fewer number of data samples to a estimate that parameter well (assuming data was generated from the distribution with those parameters), and vice versa. The Fisher information matrix associated with a statistical model parameterized by θ is extremely important in determining how a model behaves as a function of the number of training set examples.

(d) [5 points] **Approximating D_{KL} with Fisher Information**

As we explained at the start of this problem, we are interested in the set of all distributions that are at a small fixed D_{KL} distance away from the current distribution. In order to calculate D_{KL} between $p(y;\theta)$ and $p(y;\theta+d)$, where $d \in \mathbb{R}^n$ is a small magnitude "delta" vector, we approximate it using the Fisher Information at θ. Eventually d will be the natural gradient update we will add to θ. To approximate the KL-divergence with Fisher

Infomration, we will start with the Taylor Series expansion of D_{KL} and see that the Fisher Information pops up in the expansion.

Show that $D_{\mathrm{KL}}(p_\theta||p_{\theta+d}) \approx \frac{1}{2} d^T \mathcal{I}(\theta) d$.

Hint: Start with the Taylor Series expansion of $D_{\mathrm{KL}}(p_\theta||p_{\tilde{\theta}})$ where θ is a constant and $\tilde{\theta}$ is a variable. Later set $\tilde{\theta} = \theta + d$. Recall that the Taylor Series allows us to approximate a scalar function $f(\tilde{\theta})$ near θ by:

$$f(\tilde{\theta}) \approx f(\theta) + (\tilde{\theta} - \theta)^T \nabla_{\theta'} f(\theta')|_{\theta'=\theta} + \frac{1}{2}(\tilde{\theta} - \theta)^T \left(\nabla^2_{\theta'} f(\theta')|_{\theta'=\theta}\right)(\tilde{\theta} - \theta)$$

(e) [8 points] **Natural Gradient**

Now we move on to calculating the natural gradient. Recall that we want to maximize the log-likelihood by moving only by a fixed D_{KL} distance from the current position. In the previous sub-question we came up with a way to approximate D_{KL} distance with Fisher Information. Now we will set up the constrained optimization problem that will yield the natural gradient update d. Let the log-likelihood objective be $\ell(\theta) = \log p(y; \theta)$. Let the D_{KL} distance we want to move by, be some small positive constant c. The natural gradient update d^* is

$$d^* = \arg\max_d \ell(\theta + d) \quad \text{subject to} \quad D_{\mathrm{KL}}(p_\theta||p_{\theta+d}) = c \tag{1}$$

First we note that we can use Taylor approximation on $\ell(\theta + d) \approx \ell(\theta) + d^T \nabla_{\theta'} \ell(\theta')|_{\theta'=\theta}$. Also note that we calculated the Taylor approximation $D_{\mathrm{KL}}(p_\theta||p_{\theta+d})$ in the previous sub-problem. We shall substitute both these approximations into the above constrainted optimization problem.

In order to solve this constrained optimization problem, we employ the *method of Lagrange multipliers*. If you are familiar with Lagrange multipliers, you can proceed directly to solve for d^*. If you are not familiar with Lagrange multipliers, here is a simplified introduction. (You may also refer to a slightly more comprehensive introduction in the Convex Optimization section notes, but for the purposes of this problem, the simplified introduction provided here should suffice).

Consider the following constrained optimization problem

$$d^* = \arg\max_d f(d) \quad \text{subject to} \quad g(d) = c$$

The function f is the objective function and g is the constraint. We instead optimize the *Lagrangian* $\mathcal{L}(d, \lambda)$, which is defined as

$$\mathcal{L}(d, \lambda) = f(d) - \lambda[g(d) - c]$$

with respect to both d and λ. Here $\lambda \in \mathbb{R}_+$ is called the Lagrange multiplier. In order to optimize the above, we construct the following system of equations:

$$\nabla_d \mathcal{L}(d, \lambda) = 0, \tag{a}$$
$$\nabla_\lambda \mathcal{L}(d, \lambda) = 0. \tag{b}$$

So we have two equations (a and b above) with two unknowns (d and λ), which can be sometimes be solved analytically (in our case, we can).

The following steps guide you through solving the constrained optimization problem:

- Construct the Lagrangian for the constrained optimization problem (1) with the Taylor approximations substituted in for both the objective and the constraint.
- Then construct the system of linear equations (like (a) and (b)) from the Lagrangian you obtained.
- From (a), come up with an expression for d that *involves* λ.
  At this stage we have already found the "direction" of the natural gradient d, since λ is only a positive scaling constant. For most practical purposes, the solution we obtain here is sufficient. This is because we almost always include a learning rate hyperparameter in our optimization algorithms, or perform some kind of a line search for algorithmic stability. This can make the exact calculation of λ less critical. Let's call this expression $\tilde{d}$ (involving λ) as the *unscaled natural gradient*. Clearly state what is $\tilde{d}$ as a function of λ.
  The remaining steps are to figure out the value of the scaling constant λ along the direction of d, for completeness.
- Plugin that expression for d into (b). Now we have an equation that has λ but not d. Come up with an expression for λ that does *not include* d.
- Plug that expression for λ (without d) back into (a). Now we have an equation that has d but not λ. Come up with an expression for d that does *not include* λ.

The expression of d obtained this way will be the desired natural gradient update d^*. Clearly state and highlight your final expression for d^*. This expression cannot include λ.

(f) [2 points] **Relation to Newton's Method**

After going through all these steps to calculate the natural gradient, you might wonder if this is something used in practice. We will now see that the familiar Newton's method that we studied earlier, when applied to Generalized Linear Models, is equivalent to natural gradient on Generalized Linear Models. While the two methods (Netwon's and natural gradient) agree on GLMs, in general they need not be equivalent.

Show that the direction of update of Newton's method, and the direction of natural gradient, are exactly the same for Generalized Linear Models. You may want to recall and cite the results you derived in problem set 1 question 4 (Convexity of GLMs). For the natural gradient, it is sufficient to use $\tilde{d}$, the unscaled natural gradient.

4. **[30 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 labelled examples along with unlabelled examples.

In the standard unsupervised setting, we have $m \in \mathbb{N}$ unlabelled examples $\{x^{(1)}, \ldots, x^{(m)}\}$. 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:

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

Now, we will attempt to construct an extension of EM to the semi-supervised setting. Let us suppose we have an *additional* $\tilde{m} \in \mathbb{N}$ labelled examples $\{(x^{(1)}, z^{(1)}), \ldots, (x^{(\tilde{m})}, z^{(\tilde{m})})\}$ where both x and z are observed. We want to simultaneously maximize the marginal likelihood of the parameters using the unlabelled examples, and full likelihood of the parameters using the labelled 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:

$$\ell_{\text{sup}}(\theta) = \sum_{i=1}^{\tilde{m}} \log p(\tilde{x}^{(i)}, \tilde{z}^{(i)}; \theta)$$

$$\ell_{\text{semi-sup}}(\theta) = \ell_{\text{unsup}}(\theta) + \alpha \ell_{\text{sup}}(\theta)$$

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 $i \in \{1, \ldots, m\}$, set

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

M-step (semi-supervised)

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

(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 m data points $x^{(i)} \in \mathbb{R}^d, i \in \{1, \ldots, m\}$, 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 $x^{(i)}|z^{(i)} \sim \mathcal{N}(\mu_{z^{(i)}}, \Sigma_{z^{(i)}})$ i.i.d. So, μ, Σ, and ϕ are the model parameters.

We also have an additional $\tilde{m}$ data points $\tilde{x}^{(i)} \in \mathbb{R}^d, i \in \{1, \ldots, \tilde{m}\}$, and an associated *observed* variable $\tilde{z} \in \{1, \ldots, k\}$ 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 $\tilde{x}^{(i)}|\tilde{z}^{(i)} \sim \mathcal{N}(\mu_{\tilde{z}^{(i)}}, \Sigma_{\tilde{z}^{(i)}})$ i.i.d.

In summary we have $m + \tilde{m}$ examples, of which m are unlabelled data points x's with unobserved z's, and $\tilde{m}$ are labelled data points $\tilde{x}^{(i)}$ with corresponding observed labels $\tilde{z}^{(i)}$. The traditional EM algorithm is designed to take only the m unlabelled 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 leverage the additional $\tilde{m}$ labelled 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) [5 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] **[Coding Problem] Classical (Unsupervised) EM Implementation.** For this sub-question, we are only going to consider the m unlabelled examples. Follow the instructions in `src/p04_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).

Note: You only need to submit the three plots in your write-up. Your code will not be autograded.

(e) [7 points] **[Coding Problem] Semi-supervised EM Implementation.** Now we will consider both the labelled and unlabelled examples (a total of $m + \tilde{m}$), with 5 labelled examples per cluster. We have provided starter code for splitting the dataset into a matrices `x` of labelled examples and `x_tilde` of unlabelled examples. Add to your code in `src/p04_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.

Note: You only need to submit the three plots in your write-up. Your code will not be autograded.

(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.

5. **[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 `data/peppers-small.tiff` and `data/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.** From the `data` 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 262144 pixels and would take a while to cluster, we will instead run vector quantization on a smaller image. Repeat (a) with `peppers-small.tiff`. Treating each pixel's (r, g, b) values as an element of $\mathbb{R}^3$, run K-means[2] 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 matrix `A` from `peppers-large.tiff`, and replace each pixel's (r, g, b) values with the value of the closest cluster centroid. Display the new image, and compare it visually to the original image. **Include in your write-up all your code and a copy of your compressed 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?

[2] Please implement K-means yourself, rather than using built-in functions.