CS 229, Summer 2023
Problem Set #3Solutions

Alessandro Barro (`alebarro`)

Due Friday, August 11 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/41182/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 Friday, August 11 at 11:59 pm. If you submit after Friday, August 11 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. **[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 512×512 image of peppers represented in 24-bit color. This means that, for each of the 262,144 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 512×512 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.**

Answer:

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

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

Answer: In the original format, each pixel in the image was encoded using 24 bits. This allowed for a vast range of color variations and required a significant amount of storage space. The image before the compression process has a size of

$$\text{Size}_{\text{OG}} = 512 \times 512 \times 24 = 6291456 \quad \text{bits} = 786432 \quad \text{bytes} \tag{1}$$

Figure 1: Original small image

Figure 2: Original large image

Figure 3: Compresses large image

In an effort to reduce the storage requirements, the image was transformed to represent colors using only 16 distinct values. Since there are 16 options for each pixel, this can be represented using

$$\log_2 16 = 4 \tag{2}$$

bits per pixel. This is a significant reduction from the original 24 bits.

$$\text{Size}_{\text{COM}} = 512 \times 512 \times 4 = 1048576 \quad \text{bits} = 131072 \quad \text{bytes} \tag{3}$$

By reducing the number of bits required to represent each pixel from 24 to 4, the image's size was compressed by a factor of 24 divided by 4, which equals 6. In other words, the compressed image requires only one-sixth of the storage space that the original image needed, without losing the essential color information. This compression technique allows for more efficient storage and transmission of the image, potentially making it more suitable for applications where space or bandwidth is a limiting factor.

2. **[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:

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

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

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

M-step (semi-supervised)

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

(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 $\ell_{\text{semi-sup}}(\theta^{(t+1)}) \geq \ell_{\text{semi-sup}}(\theta^{(t)})$.

Answer: Let $\theta^{(t+1)}$ be the parameters obtained at exactly $t+1$ iterations of the EM algorithm. We know from hypothesis that the $\ell_{\text{semi-sup}}(\theta)$ can be expresses in the following manner

$$\ell_{\text{semi-sup}}(\theta^{(t+1)}) = \ell_{\text{unsup}}(\theta^{(t+1)}) + \alpha\ell_{\text{sup}}(\theta^{(t+1)}) \tag{4}$$

$$\sum_{i=1}^{n} \log \sum_{z^{(i)}} P(x^{(i)}; z^{(i)}; \theta)^{(t+1)} + \alpha\ell_{\text{sup}}(\theta^{(t+1)}) \tag{5}$$

Recall the definition of estimate lower-bound, given the fact that the distribution $Q_i(z^{(i)})$ coincides with the posterior distribution (E-step). From the expectation's corollary of Jensen's inequality, given f a convex function and x a random variable

$$\mathbb{E}[f(x)] \geq f(\mathbb{E}[x]) \tag{6}$$

Let's apply this concept into $\ell(\theta^{(t+1)})$. Since $\log$ is a concave function, we obtain

$$\sum_{i=1}^{n} \log \sum_{z^{(i)}} Q_i(z^{(i)}) \frac{P(x^{(i)}; z^{(i)}; \theta^{(t+1)})}{Q_i(z^{(i)})} + \alpha\ell_{\text{sup}}(\theta) \geq \sum_{i=1}^{n} \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{P(x^{(i)}; z^{(i)}; \theta^{(t+1)})}{Q_i(z^{(i)})} + \alpha\ell_{\text{sup}}(\theta^{(t+1)}) \tag{7}$$

Updating the parameters by maximizing the complete log-likelihood from the current iteration $t+1$ to a new t one (M-step)

$$\sum_{i=1}^{n} \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{P(x^{(i)}; z; \theta^{(t+1)})}{Q(z^{(i)})} + \alpha\ell_{\text{sup}}(\theta^{(t+1)}) \geq \sum_{i=1}^{n} \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{P(x^{(i)}; z; \theta^{(t)})}{Q(z^{(i)})} + \alpha\ell_{\text{sup}}(\theta^{(t)}) \tag{8}$$

From theory, we know that $Q_i(z^{(i)}) = P(z^{(i)}|x^{(i)}; \theta)$ is the posterior distribution, then $\sum_{z}^{(i)} Q_i(z^{(i)}) = 1$ is also a valid statement

$$\sum_{i=1}^{n} \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{P(x^{(i)}; z; \theta^{(t)})}{Q_i(z^{(i)})} + \alpha\ell_{\text{sup}}(\theta^{(t)}) = \sum_{i=1}^{n} \log \frac{P(x^{(i)}; z; \theta^{(t)})}{P(z^{(i)}|x^{(i)}; \theta)} + \alpha\ell_{\text{sup}}(\theta^{(t)}) \tag{9}$$

$$= \sum_{i=1}^{n} \log P(x^{(i)}; \theta^{(t)}) + \alpha\ell_{\text{sup}}(\theta^{(t)}) = \ell_{\text{unsup}}(\theta^{(t)}) + \alpha\ell_{\text{sup}}(\theta^{(t)}) = \ell_{\text{semi-sup}}(\theta^{(t)}) \tag{10}$$

We have just shown that the log-likelihood of the semi-supervised EM grows monotonically iteration by iteration, hence it will converge to locally optimal solution for the maximum likelihood estimates of the model parameters.

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 $x^{(i)}|z^{(i)} \sim \mathcal{N}(\mu_{z^{(i)}}, \Sigma_{z^{(i)}})$ 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 $\tilde{x}^{(i)}|\tilde{z}^{(i)} \sim \mathcal{N}(\mu_{\tilde{z}^{(i)}}, \Sigma_{\tilde{z}^{(i)}})$ 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.

Answer: The latent variables are the "soft-centroids" $\{z^{(1)}, ..., z^{(n)}\}_{i=1}^{n}$. Let's better define variables, data and methods taking part in the GMM algorithm

$$\begin{cases} z^{(i)} \sim \text{Multinomial}(\phi_{z^{(i)}=j}) & \text{unobserved} \\ x^{(i)}|z^{(i)} \sim N(\mu_{z^{(i)}=j}, \Sigma_{z^{(i)}=j} \\ \tilde{z}^{(i)} \in \{\tilde{z}^{(1)}, ..., \tilde{z}^{(n)}\}(\phi_{\tilde{z}^{(i)}}) & \text{observed} \\ \tilde{x}^{(i)}|\tilde{z}^{(i)} \sim N(\mu_{\tilde{z}^{(i)}}, \Sigma_{\tilde{z}^{(i)}}) \end{cases} \tag{11}$$

As the observed aspect of the data does not impact the assignment of $x^{(i)}$ to $z^{(i)}$ in any way, the E-step in the semi-supervised Gaussian Mixture Model (GMM) remains consistent with the unsupervised scenario. We can now introduce the parameters $w_j^{(i)}$ and $\tilde{w}_j^{(i)}$ as follows

$$w_j^{(i)} = Q_i(z^{(i)} = j) \tag{12}$$

$$\tilde{w}_j^{(i)} = \delta(\tilde{z}^{(i)}, j) \tag{13}$$

From theory we know that

$$= \frac{P(x^{(i)}|z^{(i)} = j; \mu, \Sigma)P(z^{(i)} = j)}{\sum_{l=1}^{k} P(x^{(i)}|z^{(i)} = j; \mu, \Sigma)P(z^{(i)} = j)} \tag{14}$$

$$= \frac{K_j e^{-\frac{1}{2}(x^{(i)} - u_j)^T \Sigma_j^{-1}(x^{(i)} - u_j)} \phi_j}{\sum_{l=1}^{k} K_l e^{-\frac{1}{2}(x^{(i)} - u_l)^T \Sigma_l^{-1}(x^{(i)} - u_l)} \phi_l} \tag{15}$$

Where the constant term $K_{j,l} = \frac{1}{(2\pi)^{\frac{d}{2}} |\Sigma_{j,l}|^{\frac{1}{2}}}$. The provided expression corresponds to the soft-assignment of a data point given the estimated $z^{(i)}$, $\forall i = 1, ..., n$.

(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.

Hint: $\phi^{(t+1)}$ must be constrained to be a probability distribution. This can be accomplished using Lagrange multipliers, as done in the course notes.

Answer: In order to derive the parameters' update rules of the M-step, we need to start from the definition of the ELBO function. For a single i-th example

$$\mathsf{ELBO}_i^{(t)}(\theta) = \mathbb{E}_{z^{(i)}\sim Q_i(z^{(i)})} \log \left[\frac{P(x^{(i)}|z^{(i)};\theta^{(t)})P(z^{(i)};\theta^{(t)})}{Q_i(z^{(i)}=j)} \right] + \alpha \left[\log P(\tilde{x}^{(i)}|\tilde{z}^{(i)};\theta^{(t)})P(\tilde{z}^{(i)};\theta^{(t)}) \right] \tag{16}$$

$$= \sum_{z^{(i)}} Q_i(z^{(i)}) \log \left[\frac{P(x^{(i)}|z^{(i)};\theta^{(t)})P(z^{(i)};\theta^{(t)})}{Q_i(z^{(i)}=j)} \right] + \alpha \left[\log P(\tilde{x}^{(i)}|\tilde{z}^{(i)};\theta^{(t)})P(\tilde{z}^{(i)};\theta^{(t)}) \right] \tag{17}$$

Let $\delta(i,j)$ be the Kronecker delta function (returns 1 if $i=j$, 0 otherwise). From the E-step, we assume that $w_j^{(i)} = Q_i(z^{(i)}=j)$ and $\tilde{w}_j^{(i)} = \delta(\tilde{z}^{(i)},j)$, with $Q_i(z^{(i)})$ being the posterior distribution $Q_i(z^{(i)}) = P(z^{(i)}|x^{(i)};\theta^{(t)})$. Therefore, we can define θ as the tuple of parameters obtained from the normal gaussian and multinomial distibutions

$$\theta^{(t)} = (\mu_j, \sigma_j, \phi_j)^{(t)} \tag{18}$$

In order to obtain closed form expression for the updated $(t+1)$ parameteres, we impose

$$\theta^{(t)} = \arg\max_\theta \ell_{\mathsf{semisup}}(\theta^{(t)}) \tag{19}$$

$$\nabla_\theta \ell_{\mathsf{semisup}} = \nabla_\theta \sum_{i=1}^{n} \mathsf{ELBO}_i^{(t)} + \alpha \ell_{\mathsf{sup}}(\theta^{(t)}) \tag{20}$$

$$= \nabla_\theta \sum_{i=1}^{n} \mathbb{E}_{z^{(i)}\sim Q_i(z^{(i)})} \log \left[\frac{P(x^{(i)}|z^{(i)};\theta^{(t)})P(z^{(i)};\theta^{(t)})}{Q_i(z^{(i)}=j)} \right] + \alpha \sum_{i=1}^{\tilde{n}} \left[\log P(\tilde{x}^{(i)}|\tilde{z}^{(i)};\theta^{(t)})P(\tilde{z}^{(i)};\theta^{(t)}) \right] \tag{21}$$

From here, let's break down the process in three steps, one for each parameter.
(1) μ_j

$$= \sum_{i=1}^{n} \mathbb{E}_{z^{(i)}\sim Q_i(z^{(i)})} \left[\nabla_{\mu_j} \log P(x^{(i)}|z^{(i)};\theta^{(t)}) \right] + \alpha \sum_{i=1}^{\tilde{n}} \left[\nabla_{\mu_j} \log P(\tilde{x}^{(i)}|\tilde{z}^{(i)};\theta^{(t)}) \right] \tag{22}$$

$$= \sum_{i=1}^{n} \mathbb{E}_{z^{(i)}\sim Q_i(z^{(i)})} \left[\nabla_{\mu_j} \left(-\frac{1}{2}(x^{(i)}-\mu_j)^T \Sigma^{-1}(x^{(i)}-\mu_j) \right) \right] + \tag{23}$$

$$+ \alpha \sum_{i=1}^{\tilde{n}} \left[\nabla_{\mu_j} \log \left(-\frac{1}{2}(\tilde{x}^{(i)}-\mu_j)^T \Sigma^{-1}(\tilde{x}^{(i)}-\mu_j) \right) \right] \tag{24}$$

$$= \sum_{i=1}^{n} w_j^{(i)} \Sigma^{-1}(x^{(i)}-\mu_j) + \sum_{i=1}^{\tilde{n}} \Sigma^{-1}(\tilde{x}^{(i)}-\mu_j) \tag{25}$$

$$= \mu_j \left(\sum_{i=1}^{n} w_j^{(i)} + \alpha \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)} \right) = \sum_{i=1}^{n} w_j^{(i)} x^{(i)} + \alpha \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)} \tilde{x}^{(i)} \tag{26}$$

$$\mu_j = \frac{\sum_{i=1}^{n} w_j^{(i)} x^{(i)} + \alpha \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)} \tilde{x}^{(i)}}{\sum_{i=1}^{n} w_j^{(i)} + \alpha \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)}} \tag{27}$$

(2) $\Sigma_j^{(t+1)}$

$$= \sum_{i=1}^{n} \mathbb{E}_{z^{(i)} \sim Q_i(z^{(i)})} \left[\nabla_{\Sigma_j} \log P(x^{(i)} | z^{(i)}; \theta^{(t)}) \right] + \alpha \sum_{i=1}^{\tilde{n}} \left[\nabla_{\Sigma_j} \log P(\tilde{x}^{(i)} | \tilde{z}^{(i)}; \theta^{(t)}) \right] \tag{28}$$

$$= \sum_{i=1}^{n} \mathbb{E}_{z^{(i)} \sim Q_i(z^{(i)})} \left[\nabla_{\Sigma_j} \left(-\frac{1}{2} (x^{(i)} - \mu_j)^T \Sigma^{-1} (x^{(i)} - \mu_j) \right) \right] + \tag{29}$$

$$+ \alpha \sum_{i=1}^{\tilde{n}} \left[\nabla_{\Sigma_j} \log \left(-\frac{1}{2} (\tilde{x}^{(i)} - \mu_j)^T \Sigma^{-1} (\tilde{x}^{(i)} - \mu_j) \right) \right] \tag{30}$$

$$= \Sigma_j \left(\sum_{i=1}^{n} w_j^{(i)} + \alpha \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)} \right) = \sum_{i=1}^{n} w_j^{(i)} (x^{(i)} - \mu_j)^T (x^{(i)} - \mu_j) + \alpha \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)} (\tilde{x}^{(i)} - \mu_j)^T (\tilde{x}^{(i)} - \mu_j) \tag{31}$$

$$\Sigma_j = \frac{\sum_{i=1}^{n} w_j^{(i)} (x^{(i)} - \mu_j)^T (x^{(i)} - \mu_j) + \alpha \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)} (\tilde{x}^{(i)} - \mu_j)^T (\tilde{x}^{(i)} - \mu_j)}{\sum_{i=1}^{n} w_j^{(i)} + \alpha \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)}} \tag{32}$$

(3) ϕ_j

$$= \sum_{i=1}^{n} \mathbb{E}_{z^{(i)} \sim Q_i(z^{(i)})} \left[\nabla_{\phi_j} \log P(z^{(i)}; \theta^{(t)}) \right] + \alpha \sum_{i=1}^{\tilde{n}} \left[\nabla_{\phi_j} \log P(\tilde{z}^{(i)}; \theta^{(t)}) \right] \tag{33}$$

Before diving into the differentiation process, we should denote the following constraint

$$\sum_{j=1}^{m} \phi_j = 1 \tag{34}$$

We are now in the position to define the Lagrangian $\mathcal{L}(\phi, \Gamma)$

$$\mathcal{L}(\phi, \Gamma) = \sum_{i=1}^{n} \sum_{j=1}^{m} w_j^{(i)} \log \phi_j + \alpha \sum_{i=1}^{\tilde{n}} \sum_{j=1}^{m} w_j^{(i)} \log \phi_j + \Gamma \left[\sum_{j=1}^{m} \phi_j - 1 \right] \tag{35}$$

Let's take the $\arg\max$ w.r.t. (*) ϕ_j and (**) Γ

$$(*) \quad \nabla_{\phi_j} \mathcal{L}(\phi, \Gamma) = \sum_{i=1}^{n} w_j^{(i)} \phi_j^{-1} + \alpha \sum_{i=1}^{\tilde{n}} \phi_j^{-1} + \Gamma = 0 \tag{36}$$

$$\phi_j = \frac{\sum_{i=1}^{n} w_j^{(i)} + \alpha \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)}}{-\Gamma} \tag{37}$$

$$(**) \quad \nabla_\Gamma \mathcal{L}(\phi, \Gamma) = \sum_{j=1}^{m} \phi_j - 1 = 0 \tag{38}$$

$$\sum_{j=1}^{m} \left[\frac{\sum_{i=1}^{n} w_j^{(i)} + \alpha \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)}}{-\Gamma} \right] = 1 \tag{39}$$

$$\Gamma = -\sum_{j=1}^{m} \left[\sum_{i=1}^{n} w_j^{(i)} + \alpha \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)} \right] \tag{40}$$

It is trivial now to fully define $\phi_{(t+1)}$

$$\phi_j = \frac{\sum_{i=1}^{n} w_j^{(i)} + \alpha \sum_{i=1}^{\tilde{n}} \delta(\tilde{z}^{(i)}, j)}{n + \alpha \tilde{n}} \tag{41}$$

In summary, the revised expressions obtained in iteration $(t+1)$ through the maximization of the log-likelihood function in the M-step of the semi-supervised GMM model, and with the substitution of the previously defined $\tilde{w}_j^{(i)}$, can be stated as follows:

$$\mu_j^{(t+1)} = \frac{\sum_{i=1}^{n} w_j^{(i)} x^{(i)} + \alpha \sum_{i=1}^{\tilde{n}} \delta(\tilde{z}^{(i)}, j) \tilde{x}^{(i)}}{\sum_{i=1}^{n} w_j^{(i)} + \alpha \sum_{i=1}^{\tilde{n}} \delta(\tilde{z}^{(i)}, j)} \tag{42}$$

$$\Sigma_j^{(t+1)} = \frac{\sum_{i=1}^{n} w_j^{(i)} (x^{(i)} - \mu_j)^T (x^{(i)} - \mu_j) + \alpha \sum_{i=1}^{\tilde{n}} \delta(\tilde{z}^{(i)}, j) (\tilde{x}^{(i)} - \mu_j)^T (\tilde{x}^{(i)} - \mu_j)}{\sum_{i=1}^{n} w_j^{(i)} + \alpha \sum_{i=1}^{\tilde{n}} \delta(\tilde{z}^{(i)}, j)} \tag{43}$$

$$\mu_j^{(t+1)} = \frac{\sum_{i=1}^{n} w_j^{(i)} x^{(i)} + \alpha \sum_{i=1}^{\tilde{n}} \delta(\tilde{z}^{(i)}, j) \tilde{x}^{(i)}}{\sum_{i=1}^{n} w_j^{(i)} + \alpha \sum_{i=1}^{\tilde{n}} \delta(\tilde{z}^{(i)}, j)} \tag{44}$$

(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.

Answer:

(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.

Answer:

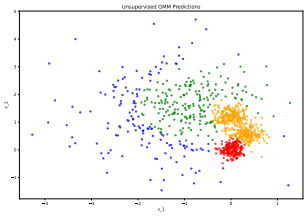


Figure 4: Unsupervised 0

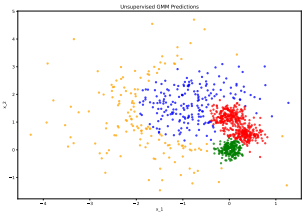


Figure 5: Unsupervised 1

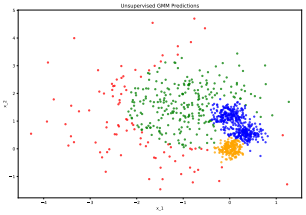


Figure 6: Unsupervised 2

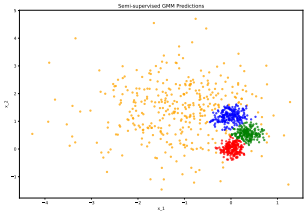


Figure 7: Semi-Supervised 0

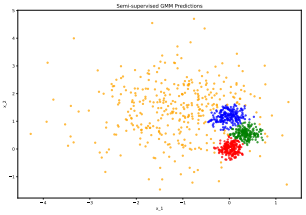


Figure 8: Semi-Supervised 1

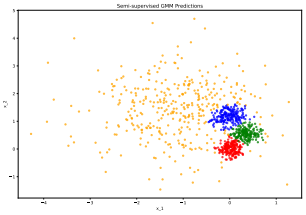


Figure 9: Semi-Supervised 2

(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.

Answer:

i. In terms of convergence, the unsupervised EM algorithm required a total of 159 iterations, while the semi-supervised version converged in just 26 iterations. This demonstrates that the semi-supervised approach is significantly more efficient in reaching convergence.

ii. Regarding stability, the assignments in the unsupervised EM showed variation with different random initializations, leading to inconsistencies. On the other hand, the semi-supervised EM provided almost identical assignments across different initializations, highlighting its greater stability.

iii. When assessing the overall quality of assignments, the unsupervised EM exhibited certain errors, such as misidentifying the presence of a single high-variance Gaussian distribution within the mixture. In contrast, the semi-supervised EM method was nearly precise in uncovering the underlying distribution, including the identification of the high-variance Gaussian component, signifying a superior quality of assignments.

3. **[10 points] PCA**

Suppose we are given a set of points $\{x^{(1)}, \ldots, x^{(n)}\}$. In class, we showed that PCA finds the "variance maximizing" directions on which to project the data:

$$u_1 \stackrel{\text{def}}{=} \arg\max_{u:\|u\|_2=1} \sum_{i=1}^{n} (u^\top x^{(i)})^2$$

In this problem, we find another interpretation of PCA.

Let us assume that we have as usual preprocessed the data to have zero-mean and unit variance in each coordinate. For a given unit-length vector u, let $f_u(x)$ be the projection of point x onto the direction given by u. I.e., if $\mathcal{V} = \{\alpha u : \alpha \in \mathbb{R}\}$, then

$$f_u(x) = \arg\min_{v \in \mathcal{V}} \|x - v\|_2^2.$$

Show that the unit-length vector u that minimizes the mean squared error between projected points and original points corresponds to the first principal component for the data. I.e., show that

$$u_1 = \arg\min_{u:\|u\|_2=1} \sum_{i=1}^{n} \|x^{(i)} - f_u(x^{(i)})\|_2^2 \ .$$

gives the first principal component.

Remark. If we are asked to find a k-dimensional subspace onto which to project the data so as to minimize the sum of squares distance between the original data and their projections, then we should choose the k-dimensional subspace spanned by the first k principal components of the data. This problem shows that this result holds for the case of $k = 1$.

Answer: Before proceeding to define a loss function to minimize and obtain a closed form expression for the principal component, we should analyze the stated projection function $f_u(x)$. Given a unit vector u and a point x, we want to find the projection of x onto the direction given by u. In other words, we need to find the value of $\alpha \in \mathbb{R}$ s.t.

$$||x - v||_2^2 = ||x - \alpha u||_2^2 \tag{45}$$

is minimized. To do so, we can proceed in this way

$$\frac{\mathrm{d}}{\mathrm{d}\alpha} ||x - \alpha u||_2^2 = 0 \tag{46}$$

$$\frac{\mathrm{d}}{\mathrm{d}\alpha} (x - \alpha u)^T (x - \alpha u) = u^T x - \alpha u^T u = 0 \tag{47}$$

$$\alpha = u^T x \in \mathbb{R} \tag{48}$$

We are now able to define a loss function $J(u)$ in the following manner

$$J(u) = \sum_{i=1}^{n} ||x^{(i)} - f_u(x^{(i)})||_2^2 = \sum_{i=1}^{n} ||x^{(i)} - (u^T x^{(i)})u||_2^2 \tag{49}$$

Our goal is to find the pricipal component by minimizing the loss function $J(u)$ w.r.t. the unit vector u

$$u_1 = \arg\min_{u:||u||_2=1} J(u) \tag{50}$$

$$= \arg\min_{u:||u||_2=1} \sum_{i=1}^{n} ||x^{(i)} - (u^T x^{(i)})u||_2^2 \tag{51}$$

$$= \arg\min_{u:||u||_2=1} \sum_{i=1}^{n} (x^{(i)} - (u^T x^{(i)})u)^T (x^{(i)} - (u^T x^{(i)})u) \tag{52}$$

$$= \arg\min_{u:||u||_2=1} \sum_{i=1}^{n} x^{(i)T} x^{(i)} - 2(u^T x^{(i)})^2 + (u^T x^{(i)})^2 u^T u \tag{53}$$

$$= \arg\min_{u:||u||_2=1} -\sum_{i=1}^{n} (u^T x^{(i)})^2 \tag{54}$$

$$= \arg\min_{u:||u||_2=1} -\left[u^T \left(\sum_{i=1}^{n} x^{(i)T} x^{(i)} \right) u \right] \tag{55}$$

Recall minimizing the argument of a negative expression is equivalent to maximizing it. It is now trivial to show that

$$u_1 = \arg\max_{u:||u||_2=1} u^T \left(\sum_{i=1}^{n} x^{(i)T} x^{(i)} \right) u \tag{56}$$

The optimal unit-length vector u that minimizes the mean squared error between the projected points and their corresponding original points corresponds to what is known as the first principal component of the data, denoted as u_1. In essence, this principal component captures the primary direction of maximum variance within the dataset. By aligning with this direction, the projected points retain as much relevant information as possible from the original data.

4. **[15 points] Independent components analysis**

While studying Independent Component Analysis (ICA) in class, we made an informal argument about why Gaussian distributed sources will not work. We also mentioned that any other distribution (except Gaussian) for the sources will work for ICA, and hence used the logistic distribution instead. In this problem, we will go deeper into understanding why Gaussian distributed sources are a problem. We will also derive ICA with the Laplace distribution, and apply it to the cocktail party problem.

Reintroducing notation, let $s \in \mathbb{R}^d$ be source data that is generated from d independent sources. Let $x \in \mathbb{R}^d$ be observed data such that $x = As$, where $A \in \mathbb{R}^{d \times d}$ is called the *mixing matrix*. We assume A is invertible, and $W = A^{-1}$ is called the *unmixing matrix*. So, $s = Wx$. The goal of ICA is to estimate W. Similar to the notes, we denote $w_j^\top$ to be the j^{th} row of W. Note that this implies that the j^{th} source can be reconstructed with w_j and x, since $s_j = w_j^\top x$. We are given a training set $\{x^{(1)}, \ldots, x^{(n)}\}$ for the following sub-questions. Let us denote the entire training set by the design matrix $X \in \mathbb{R}^{n \times d}$ where each example corresponds to a row in the matrix.

(a) [5 points] **Gaussian source**

For this sub-question, we assume sources are distributed according to a standard normal distribution, i.e. $s_j \sim \mathcal{N}(0, 1), j = \{1, \ldots, d\}$. The log-likelihood of our unmixing matrix, as described in the notes, is

$$\ell(W) = \sum_{i=1}^{n} \left(\log |W| + \sum_{j=1}^{d} \log g'(w_j^\top x^{(i)}) \right),$$

where g is the cumulative distribution function (CDF), and g' is the probability density function (PDF) of the source distribution (in this sub-question it is a standard normal distribution). Whereas in the notes we derive an update rule to train W iteratively, for the case of Gaussian distributed sources, we can analytically reason about the resulting W.

Try to derive a closed form expression for W in terms of X when g is the standard normal CDF. Deduce the relation between W and X in the simplest terms, and highlight the ambiguity (in terms of rotational invariance) in computing W.

Answer: Given the standard normal gaussian cumulative distribution function and it's first derivative (PDF)

$$g(x) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right] \tag{57}$$

$$g'(x) = \frac{1}{2\pi} e^{-\frac{x^2}{2}} \tag{58}$$

Let's proceed in order to find a closed form expression for the unmixing matrix W which highlights the relationship between this one and the design matrix X.

$$W = \arg\max_W \ell(W) \tag{59}$$

We can take the gradient of $\ell(W)$ w.r.t. W and set it equal to zero

$$\nabla_W \sum_{i=1}^{n} \left[\log |W| + \sum_{j=1}^{d} \log \frac{1}{2\pi} e^{-\frac{(w_j^T x^{(i)})^2}{2}} \right] = 0 \tag{60}$$

Moving the ∇_W operator inside the summation over the i-th examples and breaking down the derivation in two different steps we get

$$(1) \quad \nabla_W \log |W| = (W^{-1})^T \tag{61}$$

$$(2) \quad \nabla_W \sum_{j=1}^{d} \log \frac{1}{2\pi} e^{-\frac{(w_j^T x^{(i)})^2}{2}} \tag{62}$$

$$= \sum_{j=1}^{d} \nabla_W \log \frac{1}{2\pi} - \nabla_W \frac{(w_j^T x^{(i)})^2}{2} \tag{63}$$

$$= -\sum_{j=1}^{d} x^{(i)} (w_j^T x^{(i)}) \tag{64}$$

By joining the obtained expressions altogether

$$\sum_{i=1}^{n} \left[(W^{-1})^T - \sum_{j=1}^{d} (w_j^T x^{(i)}) x^{(i)T} \right] = 0 \tag{65}$$

Which can be re-formulated as follows

$$\sum_{i=1}^{n} (W^{-1})^T - \sum_{i=1}^{n} \sum_{j=1}^{d} (w_j^T x^{(i)}) x^{(i)T} = 0 \tag{66}$$

$$n(W^{-1})^T - \sum_{i=1}^{n} \sum_{j=1}^{d} w_j^T (x^{(i)} x^{(i)T}) = 0 \tag{67}$$

$$n(W^{-1})^T - W \sum_{i=1}^{n} x^{(i)} x^{(i)T} = 0 \tag{68}$$

$$n(W^{-1})^T - W X^T X = 0 \tag{69}$$

Finally coming to the below specified closed representation

$$W^T W = \left(\frac{X^T X}{n} \right)^{-1} \tag{70}$$

While this equation serves as a constraint for matrix W it does not uniquely define W itself. There exist several matrices that fulfill this constraint. Notably, if a matrix W adheres to this equation, the same holds true for any matrix produced in the form of WR where R represents any orthogonal matrix ($RR^T = I$). This insight underscores the presence of multiple valid solutions that meet the given constraint.
Let $\dot{W} = RW$, then

$$\dot{W}^T \dot{W} = W^T R^T R W = W^T W \tag{71}$$

In practical scenarios, an interesting observation emerges: the unmixing matrix W harbors an inherent ambiguity due to its susceptibility to rotation by any orthogonal matrix.

(b) [5 points] **Laplace source.**
For this sub-question, we assume sources are distributed according to a standard Laplace distribution, i.e. $s_i \sim \mathcal{L}(0,1)$. The Laplace distribution $\mathcal{L}(0,1)$ has PDF $f_{\mathcal{L}}(s) = \frac{1}{2}\exp(-|s|)$. With this assumption, derive the update rule for a single example in the form

$$W := W + \alpha(\cdots).$$

Answer: Given the i-th signal beloiging to the standard Laplace distribution, and the correlated probability density function

$$s_i \sim \mathcal{L}(0,1) \tag{72}$$

$$f_{\mathcal{L}}(s) = \frac{1}{2}e^{-|s|} \tag{73}$$

We can proceed to find a closed form expression for the unmixing matrix W, as in the previous point

$$W = \arg\max_W \ell(W) \tag{74}$$

Let's take the gradient w.r.t. W of the log-likelihood function, which will assume the following form.

$$\nabla_W \ell(W) = 0 \tag{75}$$

$$\nabla_W \sum_{i=1}^{n} \left[\log|W| + \sum_{j=1}^{d} \log \frac{1}{2} e^{-|w_j^T x^{(i)}|} \right] = 0 \tag{76}$$

Again, let's move the ∇_W operator inside the summation over the i-th elements and split the derivation process in two distinct blocks

$$(1) \quad \nabla_W \log|W| = (W^{-1})^T \tag{77}$$

$$(2) \quad \nabla_W \sum_{j=1}^{d} \log \frac{1}{2} e^{-|w_j^T x^{(i)}|} \tag{78}$$

$$= -\sum_{j=1}^{d} \nabla_W \log \frac{1}{2} + \nabla_W |w_j^T x^{(i)}| \tag{79}$$

$$= -\sum_{j=1}^{n} \begin{cases} x^(i) & \text{if} \quad w_j^T x^{(i)} > 0 \\ -x^(i) & \text{if} \quad w_j^T x^{(i)} < 0 \\ \text{not defined otherwise} \end{cases} \tag{80}$$

It is consequential to show that, under the hypothesis of the $w_j^T x^{(i)}$ inner product being always different from zero

$$w_j^T x^{(i)} \neq 0 \tag{81}$$

we can define the block (2) of the derivation process accordingly

$$-\sum_{j=1}^{d} \text{sign}[w_j^T x^{(i)}] x^{(i)} \tag{82}$$

Thus, by joining the results together, we get

$$W \leftarrow W + \alpha \nabla_W \ell(W) \tag{83}$$

$$W \leftarrow W + \alpha \sum_{i=1}^{n} \left[(W^{-1})^T - \sum_{j=1}^{d} \text{sign}[w_j^T x^{(i)}] x^{(i)} \right] \tag{84}$$

Which for a single example, corresponds to

$$W \leftarrow W + \alpha \left[(W^{-1})^T - \sum_{j=1}^{d} \text{sign}[w_j^T x^{(i)}] x^{(i)} \right] \tag{85}$$

This represents a successful derivation of a closed-form expression for the update rule in Independent Component Analysis (ICA), specifically employing the standard Laplace distribution for a singular example indexed by i.

(c) [5 points] **Cocktail Party Problem**

For this question you will implement the Bell and Sejnowski ICA algorithm, but assuming a Laplace source (as derived in part-b), instead of the Logistic distribution covered in class. The file `src/ica/mix.dat` contains the input data which consists of a matrix with 5 columns, with each column corresponding to one of the mixed signals x_i. The code for this question can be found in `src/ica/ica.py`.

Implement the `update_W` and `unmix` functions in `src/ica/ica.py`.

You can then run `ica.py` in order to split the mixed audio into its components. The mixed audio tracks are written to `mixed_i.wav` in the output folder. The split audio tracks are written to `split_i.wav` in the output folder.

To make sure your code is correct, you should listen to the resulting unmixed sources. (Some overlap or noise in the sources may be present, but the different sources should be pretty clearly separated.)

Submit the full unmixing matrix W (5×5) that you obtained, by including the W.txt the code outputs along with your code.

If you implemention is correct, your output `split_0.wav` should sound similar to the file `correct_split_0.wav` included with the source code.

Note: In our implementation, we *anneal* the learning rate α (slowly decreased it over time) to speed up learning. In addition to using the variable learning rate to speed up convergence, one thing that we also do is to choose a random permutation of the training data, and running stochastic gradient ascent visiting the training data in that order (each of the specified learning rates was then used for one full pass through the data).

Answer:
Unmixing matrix W from W.txt

$$W = \begin{bmatrix} 5.283407173865649042e+01 & 1.679520635742355594e+01 & 1.994117326518363242e+01 & -1.019863077702692244e+01 & -2.089739943202732 \\ -9.942222025724044343e+00 & -9.843907410065375618e-01 & -4.681296456213521751e+00 & 8.048384692633485216e+00 & 1.790385471427662578 \\ 8.311830803168135162e+00 & -7.476079056800498002e+00 & 1.931491020605147568e+01 & 1.517415652262788051e+01 & -1.432624582725235030 \\ -1.466676726342894810e+01 & -2.664400791263923196e+01 & 2.440627294629757138e+00 & 2.138161700821800437e+01 & -8.420724969380639635 \\ -2.689304790787783728e-01 & 1.837435608186972757e+01 & 9.312255753132744118e+00 & 9.102762073705116919e+00 & 3.059435461247027277 \end{bmatrix} \tag{86}$$

$$-2.089739943202732775e+01 \tag{87}$$

$$1.790385471427662578e+00 \tag{88}$$

$$-1.432624582725235030e+01 \tag{89}$$

$$-8.420724969380639635e+00 \tag{90}$$

$$3.059435461247027277e+01 \tag{91}$$

Source matrix S from the code's outputs

$$S = \begin{bmatrix} 52.83407174 & 16.79520636 & 19.94117327 & -10.19863078 & -20.89739943 \\ -9.94222203 & -0.98439074 & -4.68129646 & 8.04838469 & 1.79038547 \\ 8.3118308 & -7.47607906 & 19.31491021 & 15.17415652 & -14.32624583 \\ -14.66676726 & -26.64400791 & 2.44062729 & 21.38161701 & -8.42072497 \\ -0.26893048 & 18.37435608 & 9.31225575 & 9.10276207 & 30.59435461 \end{bmatrix} \tag{92}$$

`update W()` and `unmix()` functions (ica.py) are shown below

```python
def update_W(W, x, learning_rate):
    """
    Perform a gradient ascent update on W using data element x and the provided learning rate.

    This function should return the updated W.

    Args:
        W: The W matrix for ICA
        x: A single data element
        learning_rate: The learning rate to use

    Returns:
        The updated W
    """

    # * START CODE HERE *
    nabla = np.sign(np.dot(W, x))
    temp_matrix = - nabla.reshape(-1, 1) @ x.reshape(1, -1)
    updated_W = W + learning_rate * (np.linalg.inv(W.T) + temp_matrix)
    # * END CODE HERE *

    return updated_W
```

Figure 10: `update W()`

```python
def unmix(X, W):
    """
    Unmix an X matrix according to W using ICA.

    Args:
        X: The data matrix
        W: The W for ICA

    Returns:
        A numpy array S containing the split data
    """

    # * START CODE HERE *
    S = X.dot(W.T)
    # * END CODE HERE *

    return S
```

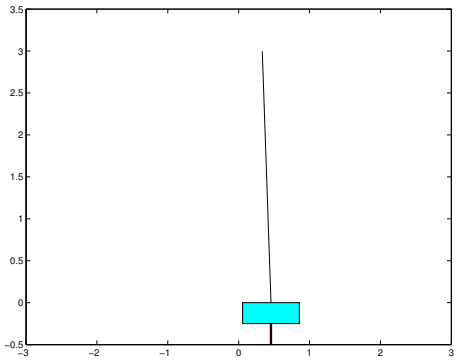
Figure 11: `unmix()`

5. **[25 points] Reinforcement Learning: The inverted pendulum**

In this problem, you will apply reinforcement learning to automatically design a policy for a difficult control task, without ever using any explicit knowledge of the dynamics of the underlying system.

The problem we will consider is the inverted pendulum or the pole-balancing problem.[3]

Consider the figure shown. A thin pole is connected via a free hinge to a cart, which can move laterally on a smooth table surface. The controller is said to have failed if either the angle of the pole deviates by more than a certain amount from the vertical position (i.e., if the pole falls over), or if the cart's position goes out of bounds (i.e., if it falls off the end of the table). Our objective is to develop a controller to balance the pole with these constraints, by appropriately having the cart accelerate left and right.



We have written a simple simulator for this problem. The simulation proceeds in discrete time cycles (steps). The state of the cart and pole at any time is completely characterized by 4 parameters: the cart position x, the cart velocity $\dot{x}$, the angle of the pole θ measured as its deviation from the vertical position, and the angular velocity of the pole $\dot{\theta}$. Since it would be simpler to consider reinforcement learning in a discrete state space, we have approximated the state space by a discretization that maps a state vector $(x, \dot{x}, \theta, \dot{\theta})$ into a number from 0 to `NUM_STATES-1`. Your learning algorithm will need to deal only with this discretized representation of the states.

At every time step, the controller must choose one of two actions - push (accelerate) the cart right, or push the cart left. (To keep the problem simple, there is no *do-nothing* action.) These are represented as actions 0 and 1 respectively in the code. When the action choice is made, the simulator updates the state parameters according to the underlying dynamics, and provides a new discretized state.

We will assume that the reward $R(s)$ is a function of the current state only. When the pole angle goes beyond a certain limit or when the cart goes too far out, a negative reward is given, and the system is reinitialized randomly. At all other times, the reward is zero. Your program must learn to balance the pole using only the state transitions and rewards observed.

The files for this problem are in `src/cartpole/` directory. Most of the the code has already been written for you, and you need to make changes only to `cartpole.py` in the places specified. This file can be run to show a display and to plot a learning curve at the end. Read the comments at the top of the file for more details on the working of the simulation.

[3]The dynamics are adapted from `http://www-anw.cs.umass.edu/rlr/domains.html`

To solve the inverted pendulum problem, you will estimate a model (i.e., transition probabilities and rewards) for the underlying MDP, solve Bellman's equations for this estimated MDP to obtain a value function, and act greedily with respect to this value function.

Briefly, you will maintain a current model of the MDP and a current estimate of the value function. Initially, each state has estimated reward zero, and the estimated transition probabilities are uniform (equally likely to end up in any other state).

During the simulation, you must choose actions at each time step according to some current policy. As the program goes along taking actions, it will gather observations on transitions and rewards, which it can use to get a better estimate of the MDP model. Since it is inefficient to update the whole estimated MDP after every observation, we will store the state transitions and reward observations each time, and update the model and value function/policy only periodically. Thus, you must maintain counts of the total number of times the transition from state s_i to state s_j using action a has been observed (similarly for the rewards). Note that the rewards at any state are deterministic, but the state transitions are not because of the discretization of the state space (several different but close configurations may map onto the same discretized state).

Each time a failure occurs (such as if the pole falls over), you should re-estimate the transition probabilities and rewards as the average of the observed values (if any). Your program must then use value iteration to solve Bellman's equations on the estimated MDP, to get the value function and new optimal policy for the new model. For value iteration, use a convergence criterion that checks if the maximum absolute change in the value function on an iteration exceeds some specified tolerance.

Finally, assume that the whole learning procedure has converged once several consecutive attempts (defined by the parameter `NO_LEARNING_THRESHOLD`) to solve Bellman's equation all converge in the first iteration. Intuitively, this indicates that the estimated model has stopped changing significantly.

The code outline for this problem is already in `cartpole.py`, and you need to write code fragments only at the places specified in the file. There are several details (convergence criteria etc.) that are also explained inside the code. Use a discount factor of $\gamma = 0.995$.

Implement the reinforcement learning algorithm as specified, and run it.

- How many trials (how many times did the pole fall over or the cart fall off) did it take before the algorithm converged? Hint: if your solution is correct, on the plot the red line indicating smoothed log num steps to failure should start to flatten out at about 60 iterations.
- Plot a learning curve showing the number of time-steps for which the pole was balanced on each trial. Python starter code already includes the code to plot. Include it in your submission.
- Find the line of code that says np.random.seed, and rerun the code with the seed set to 1, 2, and 3. What do you observe? What does this imply about the algorithm?

Answer: To make the algorithm converge, it takes about 272 trials in total (np.random.seed(0)). From the analysis of the data provided, it's evident that the algorithm's consistency starts to reach a stable point at approximately the 60th trial. This finding demonstrates that regardless of the unique initial conditions or the presence of random variations within the environment, the algorithm's performance remains steady and uniform. Even when subjected to various seeds or starting points, the algorithm behaves in a predictable manner. This stability is an indication of the algorithm's robustness, allowing it to reliably find a solution time and again. It highlights not only the algorithm's resilience to change but also its applicability across different scenarios.

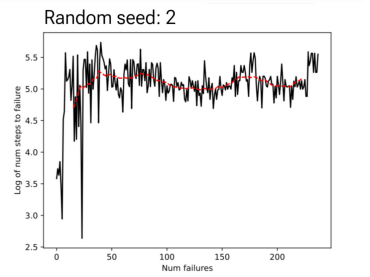


Figure 12: Cartpole w/ RS=2

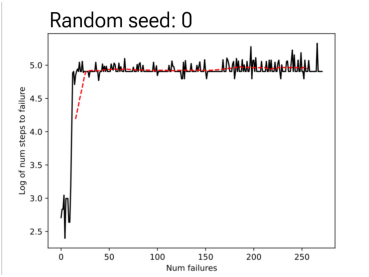


Figure 13: Cartpole w/ RS=0

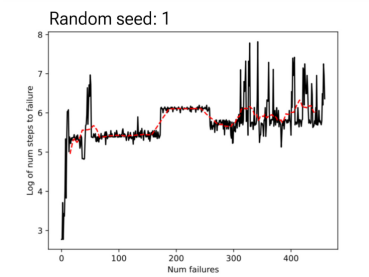


Figure 14: Cartpole w/ RS=1

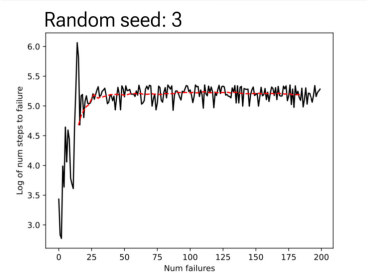


Figure 15: Cartpole w/ RS=3

Those are graphics representing the cartpole.py algorithm performance over different random seed initializations.