CS 229 Problem Set 3 Solutions

Justina Žurauskienė
SUNet:

2023 Summer

Problem 1: 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

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:

These are the results:

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:

Before compression image of peppers was represented as 24 bits (2^{24} different colours/tones); after compression we have 4 bits ($16 = 2^4$ different colours) representation. Therefore, here we have reduced image representation approximately by a factor of $24/4 = 6$.

Reference source.

Problem 2: 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 the 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:

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

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 the full likelihood of the parameters using the labeled examples by optimizing their weighted sum (with some hyperparameter α). More concretely, our semi-supervised objective $l_{\text{semi-sup}}(\theta)$ can be written as:

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

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

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 $i \in \{1, \ldots, n\}$, 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:

Using definition from above, we have

$$l_{\text{semi-sup}}(\theta^{(t+1)}) = l_{\text{unsup}}(\theta^{(t+1)}) + \alpha \cdot l_{\text{sup}}(\theta^{(t+1)})$$

$$l_{\text{semi-sup}}(\theta^{(t+1)}) = \left[\sum_{i=1}^{n} \left(\sum_{z} Q_i^{(t+1)}(z) \log \frac{p(x^{(i)}, z; \theta^{(t+1)})}{Q_i^{(t+1)}(z)} \right) + \alpha \left(\sum_{i=1}^{\tilde{n}} \log p(\tilde{x}^{(i)}, \tilde{z}^{(i)}; \theta^{(t+1)}) \right) \right]$$

Therefore, from definition of argmax we know that $\text{ELBO}(\theta^{(t+1)}) \geq \text{ELBO}(\theta^{(t)})$, giving us

$$l_{\text{semi-sup}}(\theta^{(t+1)}) \geq \left[\sum_{i=1}^{n} \left(\sum_{z} Q_i^{(t)}(z) \log \frac{p(x^{(i)}, z; \theta^{(t)})}{Q_i^{(t)}(z)} \right) + \alpha \left(\sum_{i=1}^{\tilde{n}} \log p(\tilde{x}^{(i)}, \tilde{z}^{(i)}; \theta^{(t+1)}) \right) \right] =$$

Assuming that at each M-step we are jointly optimising parameters to increase the likelihood using both – unsupervised and supervised parts, therefore it cannot decrease as to not contradict

optimisation process. Thus we can rewrite the above as,

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

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

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

$$= l_{\text{unsup}}(\theta^{(t)}) + \alpha \cdot l_{\text{sup}}(\theta^{(t)}).$$

This shows that semi-supervised likelihood is non-decreasing at each iteration, meaning it will converge.

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 $j \in \{1, ..., k\}$. We have n data points $x^{(i)} \in \mathbb{R}^d$, $i \in \{1, ..., n\}$, and each data point has a corresponding latent (hidden/unknown) variable $z^{(i)} \in \{1, ..., k\}$ 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 $\tilde{x}^{(i)} \in \mathbb{R}^d$, $i \in \{1, ..., \tilde{n}\}$, and an associated observed variable $\tilde{z}^{(i)} \in \{1, ..., k\}$ indicating the distribution $\tilde{x}^{(i)}$ belongs to. These 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 $z^{(i)}$. 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:

In the E-step of the EM algorithm for our semi-supervised Gaussian Mixture Model problem, we need to re-estimate the responsibilities (or weights) $w_j^{(i)}$ for the unsupervised part of the dataset, which represent the posterior probabilities of each latent variable $z^{(i)}$ as

$$w_j^{(i)} = Q_i(z^{(i)} = j) = P(z^{(i)} = j|x^{(i)}; \phi, \mu, \Sigma).$$

In this problem $z^{(i)}$ are given multinomial (categorical) prior distribution; whereas posterior can be expressed in a general way using Bayesian rule (see main lecture notes eq. (11.8):) giving us the following expression,

$$Q(z) = p(z|x) = \frac{p(x, z; \theta)}{\sum_z p(x, z; \theta)} = \frac{p(z = j)p(x|z = j)}{\sum_l p(z = l)p(x|z = l)}$$

In our case these terms correspond to:

- $p(z = j) \equiv \phi_j$ prior, given by the categorical distribution;
- $p(x|z = j) \equiv \mathcal{N}(x^{(i)}|\mu_j, \Sigma_j)$ is specific mixture component that was responsible for generating an observation
- $\sum_l p(z = l)p(x|z = l) \equiv \sum_{l=1}^k \phi_l \mathcal{N}(x^{(i)}|\mu_l, \Sigma_l)$ is normalising term

Therefore, the update formula will become

$$\boxed{w_j^{(i)} = \frac{\phi_j \mathcal{N}(x^{(i)}|\mu_j, \Sigma_j)}{\sum_{l=1}^k \phi_l \mathcal{N}(x^{(i)}|\mu_l, \Sigma_l)}.}$$

(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: We will be using the following expressions for multivariate Gaussian and its *log*-version,

$$\mathcal{N}(\tilde{x}^{(i)}; \mu_j, \Sigma_j) = \frac{1}{(2\pi)^{d/2}|\Sigma_j|^{1/2}} \exp\left(-\frac{1}{2}(\tilde{x}^{(i)} - \mu_j)^T \Sigma_j^{-1}(\tilde{x}^{(i)} - \mu_j)\right)$$

and

$$\log \mathcal{N}(\tilde{x}^{(i)}; \mu_j, \Sigma_j) = -\frac{d}{2}\log(2\pi) - \frac{1}{2}\log|\Sigma_j| - \frac{1}{2}(\tilde{x}^{(i)} - \mu_j)^T \Sigma_j^{-1}(\tilde{x}^{(i)} - \mu_j)$$

In the M-step, we are maximize semi-supervised objective, with respect to our parameters, $\theta = \{\phi_{l-1}, \mu_l, \sigma_l\}$, $l = 1, .., k$. Therefore, each of these will be updated according to a rule, derived below. Thus, the objective in light of GMM is given from notes for unsupervised part and we only add term arising from supervised part,

$$\sum_{i=1}^n \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \phi, \mu, \Sigma)}{Q_i(z^{(i)})} + \alpha \left(\sum_{i=1}^{\tilde{n}} \log p(\tilde{x}^{(i)}, \tilde{z}^{(i)}; \phi, \mu, \Sigma)\right) =$$

$$= \sum_{i=1}^n \sum_{j=1}^k Q_i(z^{(i)} = j) \log \frac{p(x^{(i)}|z^{(i)} = j; \mu, \Sigma)p(z^{(i)} = j; \phi)}{Q_i(z^{(i)} = j)} + \alpha \left(\sum_{i=1}^{\tilde{n}} \log p(\tilde{x}^{(i)}, \tilde{z}^{(i)}; \phi, \mu, \Sigma)\right) =$$

$$= \sum_{i=1}^n \sum_{j=1}^k w_j^{(i)} \left[\log \mathcal{N}(x^{(i)}; \mu_j, \Sigma_j) + \log \phi_j - \log w_j^{(i)}\right] + \alpha \left[\sum_{i=1}^{\tilde{n}} \log \sum_{j=1}^k \phi_j \mathcal{N}(\tilde{x}^{(i)}; \mu_j, \Sigma_j)\right] =$$

$$\sum_{i=1}^n \sum_{j=1}^k w_j^{(i)} \left[\log \mathcal{N}(x^{(i)}; \mu_j, \Sigma_j) + \log \phi_j - \log w_j^{(i)}\right] + \alpha \sum_{i=1}^{\tilde{n}} \sum_{j=1}^k \mathbb{1}_{\{\tilde{z}^{(i)}=j\}} \left[\log \phi_j + \log \mathcal{N}(\tilde{x}^{(i)}; \mu_j, \Sigma_j)\right].$$

We will now maximize this with respect to specific parameters.

- **Update for μ_l.** We can notice here that our cost function is composed of two terms unsupervised and supervised; therefore for simplicity we will build upon available results from notes (section 11.4, p.149) and will only compute gradient for additional term. Further, we will use some results from Matrix Cookbook – eq.(86). Thus, recall

$$\begin{aligned}
\nabla_{\mu_l}^{\text{unsup}} &\equiv \sum_{i=1}^{n} w_l^{(i)}(\Sigma_l^{-1}x^{(i)} - \Sigma_l^{-1}\mu_l); \quad \text{– given fact.} \\
\nabla_{\mu_l}^{\text{sup}} &\equiv \alpha\nabla_{\mu_l}\sum_{i=1}^{\tilde{n}}\sum_{j=1}^{k}\mathbb{1}_{\{\tilde{z}^{(i)}=j\}}\log\mathcal{N}(\tilde{x}^{(i)};\mu_j,\Sigma_j) = \\
&= \alpha\nabla_{\mu_l}\sum_{i=1}^{\tilde{n}}\sum_{j=1}^{k}\mathbb{1}_{\{\tilde{z}^{(i)}=j\}}\left[-\frac{1}{2}(\tilde{x}^{(i)}-\mu_j)^T\Sigma_j^{-1}(\tilde{x}^{(i)}-\mu_j)\right] = \\
&= -\frac{\alpha}{2}\sum_{i=1}^{\tilde{n}}\mathbb{1}_{\{\tilde{z}^{(i)}=l\}}\nabla_{\mu_l}\left[(\tilde{x}^{(i)}-\mu_l)^T\Sigma_l^{-1}(\tilde{x}^{(i)}-\mu_l)\right] \\
&\underbrace{=}_{\text{eq.(86)}} \alpha\sum_{i=1}^{\tilde{n}}\mathbb{1}_{\{\tilde{z}^{(i)}=l\}}\left[\Sigma_l^{-1}(\tilde{x}^{(i)}-\mu_l)\right].
\end{aligned}$$

  We can now take both terms and equate to 0, in order to solve for μ_l update rule,

$$\sum_{i=1}^{n} w_l^{(i)}(\Sigma_l^{-1}x^{(i)} - \Sigma_l^{-1}\mu_l) + \alpha\sum_{i=1}^{\tilde{n}}\mathbb{1}_{\{\tilde{z}^{(i)}=l\}}\left[\Sigma_l^{-1}(\tilde{x}^{(i)}-\mu_l)\right] = 0$$

  Expanding left hand side in above we get,

$$\sum_{i=1}^{n} w_l^{(i)}\Sigma_l^{-1}x^{(i)} - \sum_{i=1}^{n} w_l^{(i)}\Sigma_l^{-1}\mu_l + \alpha\sum_{i=1}^{\tilde{n}}\mathbb{1}_{\{\tilde{z}^{(i)}=l\}}\Sigma_l^{-1}\tilde{x}^{(i)} - \alpha\sum_{i=1}^{\tilde{n}}\mathbb{1}_{\{\tilde{z}^{(i)}=l\}}\Sigma_l^{-1}\mu_l = 0 \quad |\cdot\Sigma_l$$

$$\sum_{i=1}^{n} w_l^{(i)}\mu_l + \alpha\sum_{i=1}^{\tilde{n}}\mathbb{1}_{\{\tilde{z}^{(i)}=l\}}\mu_l = \sum_{i=1}^{n} w_l^{(i)}x^{(i)} + \alpha\sum_{i=1}^{\tilde{n}}\mathbb{1}_{\{\tilde{z}^{(i)}=l\}}\tilde{x}^{(i)} \quad \Rightarrow$$

$$\boxed{\mu_l = \frac{\sum_{i=1}^{n} w_l^{(i)}x^{(i)} + \alpha\sum_{i=1}^{\tilde{n}}\mathbb{1}_{\{\tilde{z}^{(i)}=l\}}\tilde{x}^{(i)}}{\sum_{i=1}^{n} w_l^{(i)} + \alpha\sum_{i=1}^{\tilde{n}}\mathbb{1}_{\{\tilde{z}^{(i)}=l\}}}}$$

- **Update for Σ_l.** As before, we will compute gradients here for each term – unsupervised and supervised respectively. Thus, for this we collect terms that depend on actual parameter,

and use facts from Matrix Cookbook – eqs (57, 61),

$$\nabla_{\Sigma_l}^{\text{unsup}} \equiv \nabla_{\Sigma_l} \sum_{i=1}^{n} \sum_{j=1}^{k} w_j^{(i)} \log \mathcal{N}(x^{(i)}; \mu_l, \Sigma_l) =$$

$$= \nabla_{\Sigma_l} \sum_{i=1}^{n} \sum_{j=1}^{k} w_j^{(i)} \log \left[\frac{1}{(2\pi)^{d/2} |\Sigma_j|^{1/2}} \exp \left\{ -\frac{1}{2} (x^{(i)} - \mu_j)^T \Sigma_j^{-1} (x^{(i)} - \mu_j) \right\} \right] =$$

$$= \sum_{i=1}^{n} w_l^{(i)} \nabla_{\Sigma_l} \left[-\frac{1}{2} \log |\Sigma_l| - \frac{1}{2} (x^{(i)} - \mu_l)^T \Sigma_l^{-1} (x^{(i)} - \mu_l) \right] =$$

$$= -\frac{1}{2} \sum_{i=1}^{n} w_l^{(i)} \left[\Sigma_l^{-1} - \Sigma_l^{-1} (x^{(i)} - \mu_l)(x^{(i)} - \mu_l)^T \Sigma_l^{-1} \right].$$

$$\nabla_{\Sigma_l}^{\text{sup}} \equiv \nabla_{\Sigma_l} \alpha \sum_{i=1}^{\tilde{n}} \sum_{j=1}^{k} \mathbb{1}_{\{\tilde{z}^{(i)}=j\}} \log \mathcal{N}(\tilde{x}^{(i)}; \mu_j, \Sigma_j) =$$

$$= -\frac{\alpha}{2} \nabla_{\Sigma_l} \sum_{i=1}^{\tilde{n}} \sum_{j=1}^{k} \mathbb{1}_{\{\tilde{z}^{(i)}=j\}} \left[\log |\Sigma_j| + (\tilde{x}^{(i)} - \mu_l)^T \Sigma_l^{-1} (\tilde{x}^{(i)} - \mu_l) \right] =$$

$$= -\frac{\alpha}{2} \sum_{i=1}^{\tilde{n}} \mathbb{1}_{\{\tilde{z}^{(i)}=l\}} \left[\Sigma_l^{-1} - \Sigma_l^{-1} (\tilde{x}^{(i)} - \mu_l)(\tilde{x}^{(i)} - \mu_l)^T \Sigma_l^{-1} \right].$$

Now, we can add both gradient terms together and equate to 0,

$$\nabla_{\Sigma_l}^{\text{unsup}} + \nabla_{\Sigma_l}^{\text{sup}} = 0$$

Solving this for Σ_l will result in an update rule,

$$\sum_{i=1}^{n} w_l^{(i)} \left[\Sigma_l^{-1} - \Sigma_l^{-1} (x^{(i)} - \mu_l)(x^{(i)} - \mu_l)^T \Sigma_l^{-1} \right] = -\alpha \sum_{i=1}^{\tilde{n}} \mathbb{1}_{\{\tilde{z}^{(i)}=l\}} \left[\Sigma_l^{-1} - \Sigma_l^{-1} (\tilde{x}^{(i)} - \mu_l)(\tilde{x}^{(i)} - \mu_l)^T \Sigma_l^{-1} \right]$$

We can multiply both sides of this equation by Σ_l twice, giving us,

$$\sum_{i=1}^{n} w_l^{(i)} \left[\Sigma_l - (x^{(i)} - \mu_l)(x^{(i)} - \mu_l)^T \right] = -\alpha \sum_{i=1}^{\tilde{n}} \mathbb{1}_{\{\tilde{z}^{(i)}=l\}} \left[\Sigma_l - (\tilde{x}^{(i)} - \mu_l)(\tilde{x}^{(i)} - \mu_l)^T \right]$$

$$\sum_{i=1}^{n} w_l^{(i)} \Sigma_l + \alpha \sum_{i=1}^{\tilde{n}} \mathbb{1}_{\{\tilde{z}^{(i)}=l\}} \Sigma_l = \sum_{i=1}^{n} w_l^{(i)} (x^{(i)} - \mu_l)(x^{(i)} - \mu_l)^T + \alpha \sum_{i=1}^{\tilde{n}} \mathbb{1}_{\{\tilde{z}^{(i)}=l\}} (\tilde{x}^{(i)} - \mu_l)(\tilde{x}^{(i)} - \mu_l)^T$$

$$\boxed{\Sigma_l = \frac{\sum_{i=1}^{n} w_l^{(i)} (x^{(i)} - \mu_l)(x^{(i)} - \mu_l)^T + \alpha \sum_{i=1}^{\tilde{n}} \mathbb{1}_{\{\tilde{z}^{(i)}=l\}} (\tilde{x}^{(i)} - \mu_l)(\tilde{x}^{(i)} - \mu_l)^T}{\sum_{i=1}^{n} w_l^{(i)} + \alpha \sum_{i=1}^{\tilde{n}} \mathbb{1}_{\{\tilde{z}^{(i)}=l\}}}}$$

- **Update for ϕ_l.** Finally, we can collect together all the terms that depend on ϕ_j, we find that we need to maximize,

$$\sum_{i=1}^{n}\sum_{j=1}^{k} w_j^{(i)} \log \phi_j + \alpha \sum_{i=1}^{\tilde{n}}\sum_{j=1}^{k} \mathbb{1}_{\{\tilde{z}^{(i)}=j\}} \log \phi_j + \beta(\sum_{j=1}^{k} \phi_j - 1),$$

here the last term was introduced to incorporate constraints imposed on ϕ_j by using Lagrange multiplier β (as per p 150 in main lecture notes). We can now compute the gradient with respect to ϕ_l, giving us,

$$\nabla_{\phi_l} \equiv \nabla_{\phi_l} \left[\sum_{i=1}^{n}\sum_{j=1}^{k} w_j^{(i)} \log \phi_j + \alpha \sum_{i=1}^{\tilde{n}}\sum_{j=1}^{k} \mathbb{1}_{\{\tilde{z}^{(i)}=j\}} \log \phi_j + \beta(\sum_{j=1}^{k} \phi_j - 1) \right] =$$

$$= \sum_{i=1}^{n} \frac{w_l^{(i)}}{\phi_l} + \alpha \sum_{i=1}^{\tilde{n}} \mathbb{1}_{\{\tilde{z}^{(i)}=l\}} \frac{1}{\phi_j} + \beta.$$

Setting this term for 0, we get,

$$\sum_{i=1}^{n} \frac{w_l^{(i)}}{\phi_l} + \alpha \sum_{i=1}^{\tilde{n}} \frac{\mathbb{1}_{\{\tilde{z}^{(i)}=l\}}}{\phi_l} + \beta = 0 \quad \Rightarrow \quad \phi_l = -\frac{\sum\limits_{i=1}^{n} w_l^{(i)} + \alpha \sum\limits_{i=1}^{\tilde{n}} \mathbb{1}_{\{\tilde{z}^{(i)}=l\}}}{\beta}$$

We can now use constraint that $\sum_j^k \phi_j = 1$ to get expression for β,

$$\sum_{j=1}^{k} \phi_j = -\frac{\sum\limits_{i=1}^{n}\sum\limits_{j=1}^{k} w_j^{(i)} + \alpha \sum\limits_{i=1}^{\tilde{n}}\sum\limits_{j=1}^{k} \mathbb{1}_{\{\tilde{z}^{(i)}=j\}}}{\beta}$$

$$1 = -\frac{\sum\limits_{i=1}^{n} 1 + \alpha \sum\limits_{i=1}^{\tilde{n}} 1}{\beta} \quad \Rightarrow \quad \beta = -(n + \alpha\tilde{n})$$

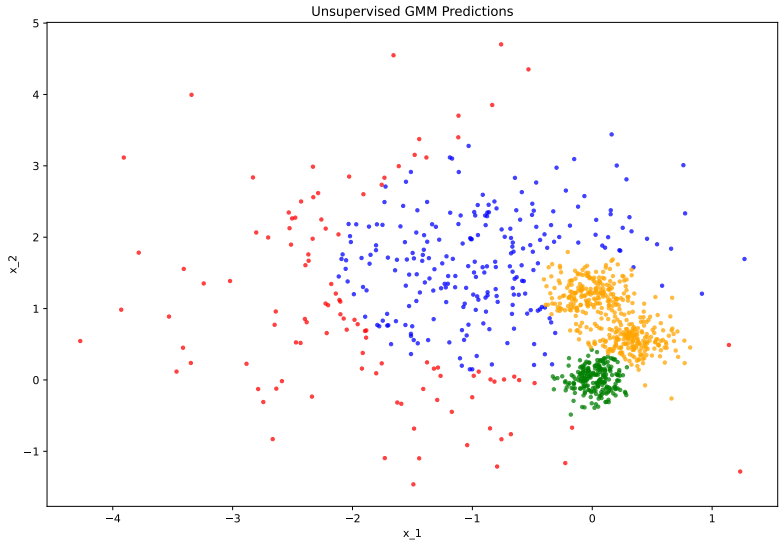
Therefore, final expression for ϕ_l update rule is,

$$\boxed{\phi_l = \frac{\sum\limits_{i=1}^{n} w_l^{(i)} + \alpha \sum\limits_{i=1}^{\tilde{n}} \mathbb{1}_{\{\tilde{z}^{(i)}=l\}}}{n + \alpha\tilde{n}}}$$

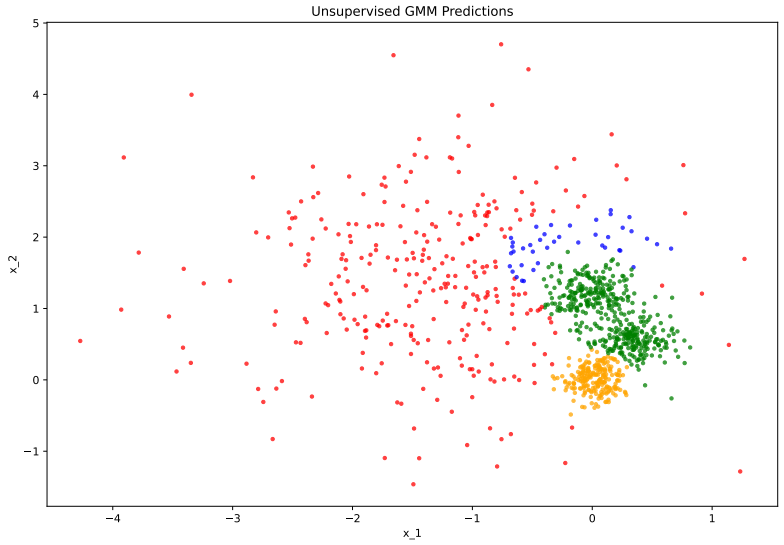
(d) **[5 points]** Classical (Unsupervised) EM Implementation. For this sub-question, we are only going to consider the n unlabelled examples. Follow the instructions in `src/semi supervised em/gmm.py` to implement the traditional EM algorithm, and run it on the unlabelled data-set until convergence.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

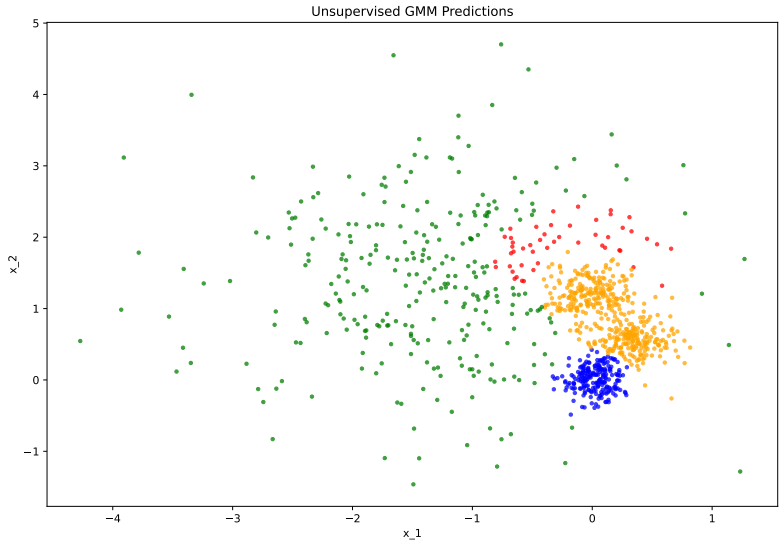
Please see plots for trial 1 .



Please see plots for trial 2.



Please see plots for trial 3.



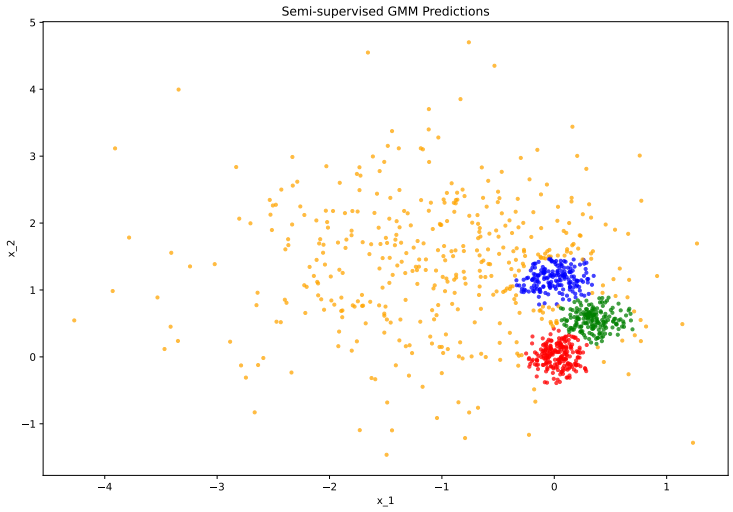
(e) **[7 points]** Semi-supervised EM Implementation. Now we will consider both the labelled and unlabelled examples (a total of $n + \tilde{n}$), with 5 labelled examples per cluster. We have provided starter code for splitting the dataset into matrices `x` and `x_tilde` of unlabelled and labelled examples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

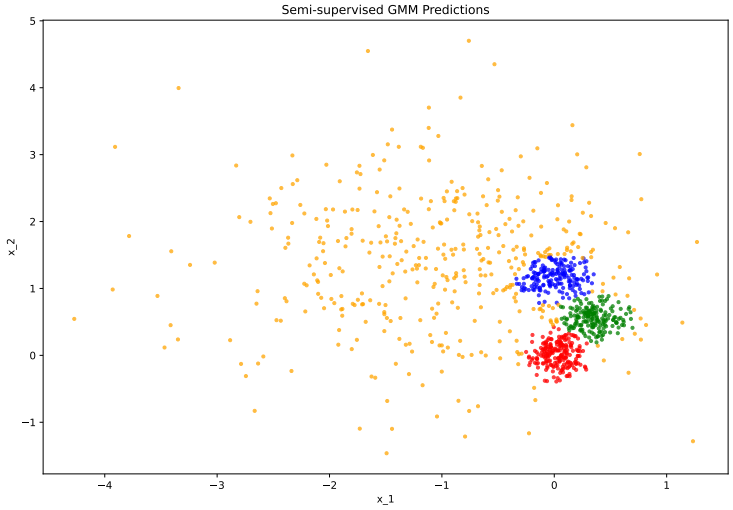
Submit the three plots obtained above in your write-up.

Answer:

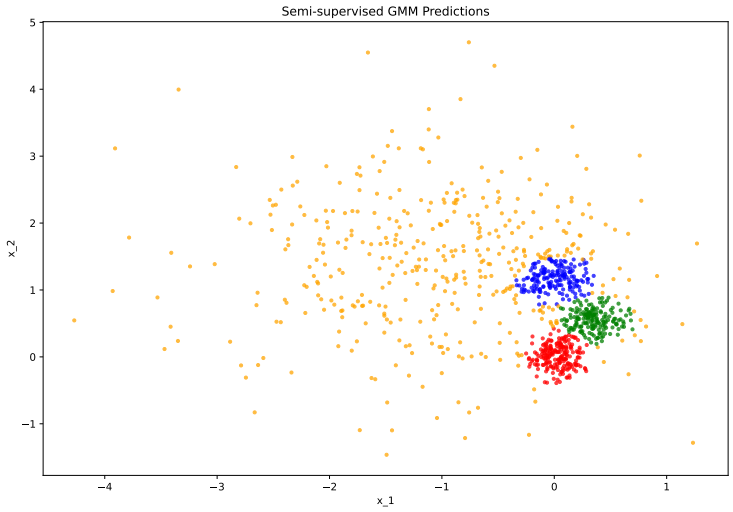
Please see trial 1 result:



Please see trial 2 result:



Please see trial 3 result:



(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.

Answer:

	Unsupervised model iterations	Semi-supervised model iterations
Trial 1	156	59
Trial 2	63	81
Trial 3	97	56
On average	**105**	**65**

We can see that on average, using semi-supervised model we have reached convergence much faster, i.e. using fewer iterations.

ii. Stability (i.e., how much did assignments change with different random initializations?)

Answer:

From the output plots, we observe that for the semi-supervised model, we consistently reached the same final cluster indicator vector across all three trials, indicating that the final assignments were identical, and thus are stable. However, the same cannot be said for the purely unsupervised model, where the final cluster assignments varied across trials.

However, it is worth noting that the final assignments in trials 2 and 3 for the unsupervised model were very similar when we disregard the color scheme; in mixture models, we do not explicitly label clusters.

From this, we can infer that performance of the semi-supervised model was significantly more stable on the given dataset compared to the unsupervised model.

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:

From the plots, it is evident that the data was simulated from a mixture model: three Gaussian distributions exhibited a "tight" covariance structure, and one displayed a "broad" structure, causing points to cluster closer or spread out, respectively. This configuration leads to overlapping clusters, which means that in the absence of prior knowledge (using only the unsupervised model), points from the broad cluster were often mislabeled as belonging to clusters with a tighter covariance structure or split clusters. However, by incorporating supplementary information (points with known cluster assignments), the semi-supervised model could more accurately discern the underlying structure in the data, resulting in more appropriate and reliable cluster assignments.

More formally, the quality of predicted assignments could be better understood if we could compare them to original indicator variables obtained while simulating this dataset; provided that such information is available to us, we could use adjusted RAND index to achieve this.

Problem 3: 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:

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

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. In other words, if $V = \{\alpha u : \alpha \in \mathbb{R}\}$, then

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

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. In other words, 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 squared distances 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:

Statements:

- Here $f_u(x^{(i)})$ denotes projection - it is a point thet represents the projection of the point $x^{(i)}$ onto the direction given by the unit vector u. Specifically, it's the point on the line (or subspace) spanned by the vector u that is closest to $x^{(i)}$.

- $V = \{\alpha u : \alpha \in \mathbb{R}\}$ defines here the space (in this case, a line) onto which we are projecting containing all possible scalar multiples of the vector u. I.e. this set represents a line in the space that passes through the origin and is oriented in the direction of u.

The $f_u(x)$ can be understood as a specific point in line V that represents a shortest distance from x to line V:

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

From notes (p.159) we know that given a unit vector u and a point x, the length of the projection of x onto u is given by $x^T u$. I.e., if $x^{(i)}$ is a point in the data set, then its projection onto u is distance ${x^{(i)}}^T u$ from the origin. We will need to consider this point in it's vector form $({x^{(i)}}^T u)u$ (scalar projection).

We will use this fact: $\|a - b\|_2^2 = a^T a - 2a^T b + b^T b$ to unpack the cost function,

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

$$= \sum_{i=1}^{n} \left\{ {x^{(i)}}^T x^{(i)} - 2{x^{(i)}}^T ({x^{(i)}}^T u)u + [({x^{(i)}}^T u)u]^T [({x^{(i)}}^T u)u] \right\} =$$

$$= \sum_{i=1}^{n} \left\{ {x^{(i)}}^T x^{(i)} - 2({x^{(i)}}^T u)^2 + \underbrace{[{x^{(i)}}^T u]^T}_{\#^T = \#} u^T \underbrace{{x^{(i)}}^T u}_{\#} u \right\} =$$

$$= \sum_{i=1}^{n} \left\{ {x^{(i)}}^T x^{(i)} - 2({x^{(i)}}^T u)^2 + ({x^{(i)}}^T u)^2 \underbrace{u^T u}_{=1} \right\} =$$

$$= \sum_{i=1}^{n} \left\{ {x^{(i)}}^T x^{(i)} - ({x^{(i)}}^T u)^2 \right\}.$$

We notice here that the first term in the sum does not depend on u, so we can disregard it. Looking closer at the second term we notice that inside the brackets the term will be always non-negative; it also looked similar to the definition of PCA in terms of "variance maximization". Therefore by maximising this term (variance), we will be minimising the cost $J(u)$.

Problem 4: Independent components analysis

While studying Independent Component Analysis (ICA) in class, we made an informal argu- ment 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 dis- tributed 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 jth row of W. Note that this implies that the jth 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

$$\ell(W) = \sum_{i=1}^{n} \left(\log |W| + \sum_{j=1}^{d} \log g'(w_j^T x^{(i)}) \right),$$

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:

Here we are given some data $s \in \mathbb{R}^d$ that is generated via d independent sources and we model the observations via

$$x = As,$$

where A is an unknown square matrix called the mixing matrix. Each row in dataset x contains repeated observations $\{x^{(i)}; i = 1, \ldots, n\}$, with goal to recover the sources $s^{(i)}$ that generated our observations ($x^{(i)} = As^{(i)}$).

For this we can consider the reverse relationship,

$$W = A^{-1}X \quad \Rightarrow \quad S = WX;$$

where W is unmixing matrix.

The *log*-likelihood is given by

$$\ell(W) = \sum_{i=1}^{n} \left(\log |W| + \sum_{j=1}^{d} \log g'(w_j^T x^{(i)}) \right),$$

We can replace g' with Gaussian term:

$$g'(z) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{z^2}{2}\right) \quad \Rightarrow \quad \log g'(z) = \log \frac{1}{\sqrt{2\pi}} - \frac{z^2}{2}$$

giving us,

$$\ell(W) = \sum_{i=1}^{n} \left(\log |W| - \frac{1}{2} \sum_{j=1}^{d} (w_j^T x^{(i)})^2 \right) =$$

$$= \sum_{i=1}^{n} \log |W| - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{d} (w_j^T x^{(i)})^2 =$$

$$= \sum_{i=1}^{n} \log |W| - \frac{1}{2} \|W^T X\|_F^2 \quad \text{see here.}$$

Thus, we can now differentiate it with respect to W; for this we will use result (49) and (132) from Matrix cookbook,

$$\nabla_W \ell(W) \equiv \nabla_W \sum_{i=1}^{n} \log |W| - \nabla_W \frac{1}{2} \|W^T X\|_F^2 =$$

$$= \sum_{i=1}^{n} \frac{1}{|W|} \cdot |W| (W^{-1})^T - \frac{1}{2} \cdot 2 W^T X X^T =$$

$$= n W^{-1^T} - W^T X X^T.$$

Equating this to 0, we get that,

$$W^{-1^T} = \frac{1}{n} X X^T W \quad | \cdot W^{-1} \quad \Rightarrow \quad W^{-1^T} W^{-1} = \frac{1}{n} X X^T.$$

Now we will work on left side to put the whole equation in different format. Here we will use the following results from the Matrix Cookbook, i.e. eqs: (3), (1)

$$W^{-1^T} W^{-1} \overset{eq.(3)}{=} W^{T^{-1}} W^{-1} \overset{eq.(1)}{=} (W W^T)^{-1}$$

Therefore, we arrive at:

$$(W W^T)^{-1} = \frac{1}{n} X X^T \quad \Rightarrow \quad (W^T W)^{-1} = \frac{1}{n} X X^T$$

We can now apply rotation matrix by picking R to be an orthogonal matrix, so that $RR^T = R^T R = I$. If we denote now $W = R\tilde{W}$ it follows that $W^T = (R\tilde{W})^T = \tilde{W}^T R^T$ therefore rewriting above equation as using this result,

$$\tilde{W}^T \underbrace{R^T R}_{=:I} \tilde{W} = \frac{1}{n} X X^T$$

$$\tilde{W}^T \tilde{W} = \frac{1}{n} X X^T \quad \text{matching result on } W \text{ (12.39)}$$

What we have shown here is that there exist a whole family of matrices and we cannot distinguish between latent variables where these differ simply by a rotation in latent space. Thus, assuming Gaussian latent variable distribution is insufficient to find independent components.

(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

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

Answer:

Here we utilise same likelihood form as before, however in this case, g' takes form of Lapace distribution. We will not be considering sum over n, as update is meant to be computed for a single s_i. Thus, we have the following likelihood,

$$\ell(W) = \log(|W|) + \sum_{j=1}^{d} \log\frac{1}{2}\exp\left\{-|w_j^T x^{(i)}|\right\} = \log(|W|) + \sum_{j=1}^{d}\left(\log\frac{1}{2} - |w_j^T x^{(i)}|\right) =$$

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

To compute the gradient of this ℓ we will use square root representation for computing derivative of absolute values.

Square root approach to compute derivatives of absolute functions:

Given the function:

$$f(x) = |ax|$$

To differentiate this, we introduce a substitution:

$$u = ax$$
$$f(x) = |u| = \sqrt{u^2}$$

Using the chain rule:

$$f'(x) = \frac{df}{du}\cdot\frac{du}{dx}$$

The derivative with respect to u:

$$f'(x) = \frac{1}{2}\cdot\frac{2u}{\sqrt{u^2}} = \frac{u}{\sqrt{u^2}}\frac{du}{dx} = \frac{au}{|au|}\cdot a$$

Given this formula derived via the chain rule we use it to obtain our derivatives:

$$\frac{w_j^T x^{(i)}}{|w_j^T x^{(i)}|}\cdot x^{(i)}.$$

Thus, using this result and results from section 4(a) we get,

$$\nabla_W \ell(W) \equiv W^{-1^T} - \sum_{j=1}^{d}\frac{w_j^T x^{(i)}}{|w_j^T x^{(i)}|}x^{(i)}.$$

So the update rule will be,

$$W := W + \alpha \left(W^{-1^T} - \sum_{j=1}^{d} \frac{w_j^T x^{(i)}}{|w_j^T x^{(i)}|} x^{(i)} \right)$$

Last term represent a matrix of same size as $W_{d \times d}$. To see this we can expand above term,

$$W := W + \alpha \left(W^{-1^T} - \begin{bmatrix} \frac{w_1^T x^{(i)}}{|w_1^T x^{(i)}|} \\ \dots \\ \frac{w_d^T x^{(i)}}{|w_d^T x^{(i)}|} \end{bmatrix} x^{(i)^T} \right).$$

The above formula is true, because the derivative of absolute value function can be computed using alternative – sign function formulation:

The sign function can be denoted as $\text{sgn}(x)$ and defined as:

$$\text{sgn}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Provided the absolute value of a function $f(x)$ is $|f(x)|$, the absolute value can be expressed in terms of the sign function :

$$|f(x)| = f(x) \cdot \text{sgn}(f(x))$$

Taking the derivative with respect to x, we get:

$$\frac{d|f(x)|}{dx} = \frac{df(x)}{dx} \cdot \text{sgn}(f(x))$$

In our case this would translate to,

$$\text{sgn}(w_j^T x^{(i)}) \cdot x^{(i)}$$

If we look closer at this, it does match my previous result, because, sign function can be expressed as,

$$\text{sgn}(w_j^T x^{(i)}) = \frac{|w_j^T x^{(i)}|}{w_j^T x^{(i)}} = \frac{w_j^T x^{(i)}}{|w_j^T x^{(i)}|}.$$

Meaning, expressions are equivalent

$$W := W + \alpha \left(W^{-1^T} - \begin{bmatrix} \text{sgn}(w_1^T x^{(i)}) \\ \dots \\ \text{sgn}(w_d^T x^{(i)}) \end{bmatrix} x^{(i)^T} \right).$$

(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 xi. 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 your implementation 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:

For convenience, W matrix is summarised below,

```
[[ 52.83273251  16.79517069  19.94026397 -10.1980702  -20.89685092]
 [ -9.93260541  -0.9781464   -4.67949962   8.04445827   1.7902254 ]
 [  8.31062501  -7.477177    19.31477305  15.17487314 -14.3263323 ]
 [-14.66719563 -26.64540631   2.4409785   21.38218656  -8.42116771]
 [ -0.26891296  18.37417645   9.31226672   9.10289145  30.59437247]]
```

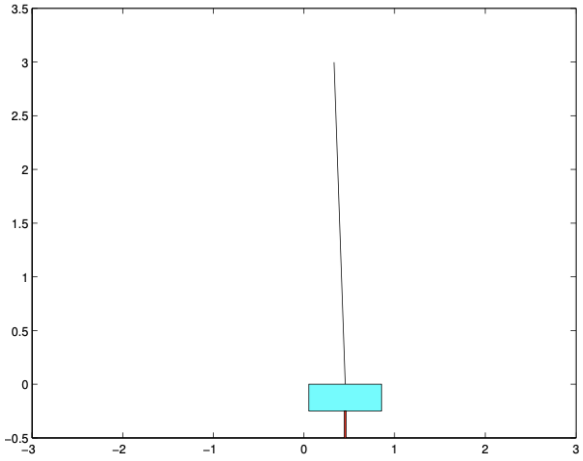
Problem 5: Reinforcement Learning: The inverted pendulum

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[1].

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.

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



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 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.

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.

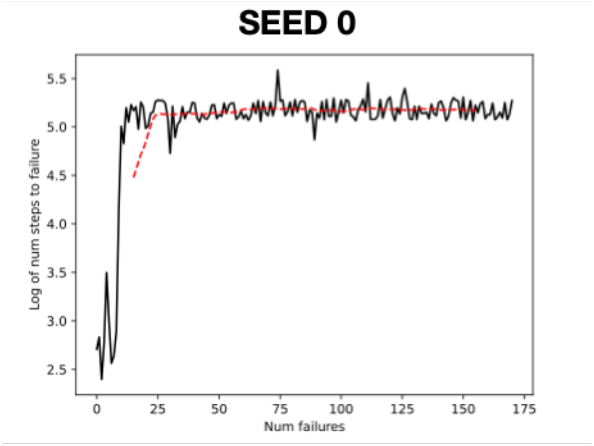
  Answer:

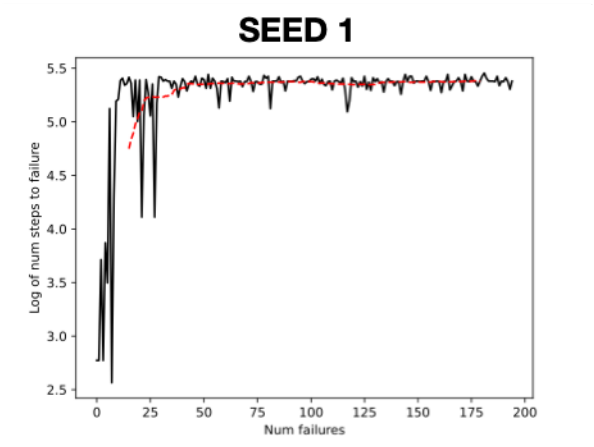
  It is approximately around after 50 – trials for seed 0, if we take a bit longer burn-in period. However, I think it could be declared a bit earlier – after approximately 40 trials.

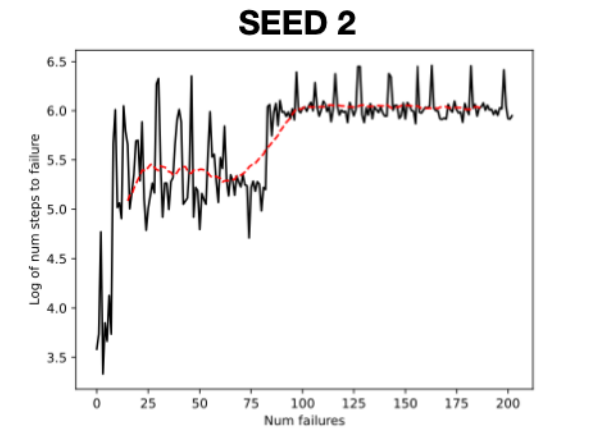
- Plot a learning curve showing the number of time-steps for which the pole was balanced on each trial. Python starter code already includes the code to plot. Include it in your submission.

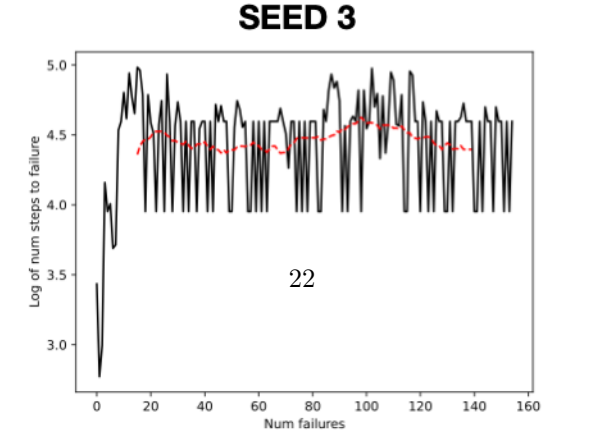
  Answer:

  Please see plot corresponding to seed 0,


SEED 0
Log of num steps to failure
Num failures
5.5
5.0
4.5
4.0
3.5
3.0
2.5
0
25
50
75
100
125
150
175


SEED 1
Log of num steps to failure
Num failures
5.5
5.0
4.5
4.0
3.5
3.0
2.5
0
25
50
75
100
125
150
175
200


SEED 2
Log of num steps to failure
Num failures
6.5
6.0
5.5
5.0
4.5
4.0
3.5
0
25
50
75
100
125
150
175
200


SEED 3
Log of num steps to failure
Num failures
5.0
4.5
4.0
3.5
3.0
0
20
40
60
80
100
120
140
160

- 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:

  Different plots here represent different seed values from 0 to 3. We observe that the algorithm exhibits sensitivity to different seed values, which translates to varying behaviors and results, such as different number of trials required to reach convergence.

  In the context of real-world applications, this sensitivity is not a desirable feature. Ideally, we want to build an algorithm that provides consistent and robust outputs (despite being stochastic in nature) in the face of minor perturbations or changes, such as seed values. This can also highlight the need to consider model refinement and the trade-off between complexity and generalization.