Question 1. Getting Started

You may typeset your homework in latex or submit neatly handwritten and scanned solutions. Please make sure to start each question on a new page, as grading (with Gradescope) is much easier that way! Deliverables:

1. Submit a PDF of your writeup to assignment on Gradescope, "HW[n] Write-Up"
2. Submit all code needed to reproduce your results, "HW[n] Code".
3. Submit your test set evaluation results, "HW[n] Test Set".

After you've submitted your homework, be sure to watch out for the self-grade form.

(a) Who else did you work with on this homework? In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

I worked with Weiran Liu and Katherine Li. I feel really sick right now but I really want to crack the homework ASAP. I'm glad to see a entire question becomes the BONUS not only a few parts. But I will do it.

(b) Please copy the following statement and sign next to it. We just want to make it *extra* clear so that no one inadvertently cheats.

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up.

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up.

Signature: 

Question 2. Classification Policy

Suppose we have a classification problem with classes labeled $1, \ldots, c$ and an additional "doubt" category labeled $c+1$. Let $f : \mathbb{R}^d \to \{1, \ldots, c+1\}$ be a decision rule. Define the loss function

$$L(f(\vec{x}), y) = \begin{cases} 0 & \text{if } f(\vec{x}) = y \quad f(\vec{x}) \in \{1, \ldots, c\}, \\ \lambda_c & \text{if } f(\vec{x}) \neq y \quad f(\vec{x}) \in \{1, \ldots, c\}, \\ \lambda_d & \text{if } f(\vec{x}) = c+1 \end{cases} \tag{1}$$

where $\lambda_c \geq 0$ is the loss incurred for making a misclassification and $\lambda_d \geq 0$ is the loss incurred for choosing doubt. In words this means the following:

- When you are correct, you should incur no loss.
- When you are incorrect, you should incur some penalty λ_c for making the wrong choice.
- When you are unsure about what to choose, you might want to select a category corresponding to "doubt" and you should incur a penalty λ_d.

We can see that in practice we'd like to have this sort of loss function if we don't want to make a decision if we are unsure about it. This sort of loss function, however, doesn't help you in instances where you have high certainty about a decision, but that decision is wrong.

To understand the expected amount of loss we will incur with decision rule $f(\vec{x})$, we look at the risk. The risk of classifying a new data point $\vec{x}$ as class $f(\vec{x}) \in \{1, 2, \ldots, c+1\}$ is

$$R(f(\vec{x})|\vec{x}) = \sum_{i=1}^{c} L(f(\vec{x}), i)\, P(Y = i|\vec{x}).$$

(a) **Show that the following policy $f_{opt}(x)$ obtains the minimum risk:**

- Find class i such that $P(Y = i|\vec{x}) \geq P(Y = j|\vec{x})$ for all j, meaning you pick the class with the highest probability given x.
- Choose class i if $P(Y = i|\vec{x}) \geq 1 - \frac{\lambda_d}{\lambda_c}$
- Choose doubt otherwise.

1. We prove that one must choose the class i that $P(Y = i|\vec{x}) \geq P(Y = j|\vec{x})$ for all j by contradiction. Let's assume one chooses class m and there exists a class k that $P(Y = m|\vec{x}) < P(Y = k|\vec{x})$. The risk can be calculated as:

$$\begin{aligned} R(f(\vec{x}) = m|\vec{x}) &= \sum_{i=1}^{c} L(m, i)\, P(Y = i|\vec{x}) \\ &= \sum_{i=1, i\neq m}^{c} L(m, i)\, P(Y = i|\vec{x}) + L(m, m)\, P(Y = m|\vec{x}) \\ &= \lambda_c \sum_{i=1, i\neq m, k}^{c} P(Y = i|\vec{x}) \\ &= \lambda_c (1 - P(Y = m|\vec{x})) \end{aligned}$$

However, if one chooses class k we will have a risk:

$$\begin{aligned} R(f(\vec{x}) = k|\vec{x}) =& \sum_{i=1}^{c} L(k,i)\, P(Y = i|\vec{x}) \\ =& \lambda_c(1 - P(Y = k|\vec{x})) \end{aligned}$$

Because we know $P(Y = m|\vec{x}) < P(Y = k|\vec{x})$, we have $1 - P(Y = m|\vec{x}) > 1 - P(Y = k|\vec{x})$. Thus we find a better choice k which has a lower risk.

$$R(f(\vec{x}) = k|\vec{x}) < R(f(\vec{x}) = m|\vec{x})$$

2. Now we prove that choose "doubt" class $c + 1$ is only better if $P(Y = i|\vec{x}) < 1 - \frac{\lambda_d}{\lambda_c}$ for all i also by contradiction. Let's assume one chooses class m instead of class "doubt" that $P(Y = m|\vec{x}) < 1 - \frac{\lambda_d}{\lambda_c}$. The risk can be calculated as:

$$\begin{aligned} R(f(\vec{x}) = m|\vec{x}) =& \sum_{i=1}^{c} L(m,i)\, P(Y = i|\vec{x}) \\ =& \lambda_c(1 - P(Y = m|\vec{x})) \\ >& \lambda_c(1 - 1 + \frac{\lambda_d}{\lambda_c}) \\ =& \lambda_d \end{aligned}$$

However, if one chooses class $c + 1$ the risk becomes:

$$\begin{aligned} R(f(\vec{x}) = c + 1|\vec{x}) =& \sum_{i=1}^{c} L(m,i)\, P(Y = i|\vec{x}) \\ =& \lambda_d \sum_{i=1}^{c} P(Y = i|\vec{x}) \\ =& \lambda_d \end{aligned}$$

Therefore, we can see choosing "doubt" class $c + 1$ gives us a smaller risk. On the contrary, if the above condition is not true, we should choose the class with the highest probability. As a summary, we have shown the following strategy yields the best choice for $f_{opt}(x)$:

- Find class i such that $P(Y = i|\vec{x}) \geq P(Y = j|\vec{x})$ for all j, meaning you pick the class with the highest probability given x.
- Choose class i if $P(Y = i|\vec{x}) \geq 1 - \frac{\lambda_d}{\lambda_c}$
- Choose doubt otherwise.

(b) **How would you modify your optimum decision rule if $\lambda_d = 0$? What happens if $\lambda_d > \lambda_c$? Explain why this is or is not consistent with what one would expect intuitively.**

1. If $\lambda_d = 0$, I would always choose "doubt" class $c + 1$ except that one class has a probability of 1. because the second rule becomes "Choose class i if $P(Y = i|\vec{x}) \geq 1$" and we know that $P(Y = i|\vec{x}) \leq 1$. This is because we have to cost to choose "doubt" and will lose all the prediction power. That is we should:

- Choose class i if $P(Y = i|\vec{x}) = 1$
- Choose doubt otherwise.

2. If $\lambda_d > \lambda_c$, I would only keep rule one and get rid of the others because the second rule becomes "Choose class i if $P(Y = i|\vec{x}) \geq -|\frac{\lambda_d}{\lambda_c} - 1|$" and we know that $P(Y = i|\vec{x}) \geq 0$. That is, we never choose doubt because it is so costly that one should rather choose another class. Therefore, our rules become:

- Find class i such that $P(Y = i|\vec{x}) \geq P(Y = j|\vec{x})$ for all j, meaning you pick the class with the highest probability given x.

This is not something one would expect intuitively because we should only choose when we are very uncertain. Always or never choosing doubt is too extreme to be true. Always choosing doubt has no prediction power; never choosing doubt usually results in overfitting.

Question 3. LDA and CCA

Consider the following random variable $\vec{X} \in \mathbb{R}^d$, generated using a *mixture of two Gaussians*. Here, the vectors $\vec{\mu}_1, \vec{\mu}_2 \in \mathbb{R}^d$ are arbitrary (mean) vectors, and $\mathbf{\Sigma} \in \mathbb{R}^{d \times d}$ represents a positive definite (covariance) matrix. For now, we will assume that we know all of these parameters.

Draw a label $L \in \{1, 2\}$ such that the label 1 is chosen with probability π_1 (and consequently, label 2 with probability $\pi_2 = 1 - \pi_1$), and generate $\vec{X} \sim \mathcal{N}(\vec{\mu}_L, \vec{\Sigma})$.

(a) Now given a particular $\vec{X} \in \mathbb{R}^d$ generated from the above model, we wish to find its label. **Write out the decision rule corresponding to the following estimates of L:**

- MLE
- MAP

Your decision rule should take the form of a threshold: if some function $f(\vec{X}) > T$, then choose the label 1, otherwise choose the label 2. **When are these two decision rules the same?** Hint: investigate the ratio between the two likelihood functions and the ratio between the two posterior probabilities respectively.

---

1. MLE:

$$\ln l(\vec{X}|L = k) = -\frac{d}{2}\ln(2\pi) - \frac{1}{2}\ln|\Sigma| - \frac{1}{2}(\vec{X} - \vec{\mu}_k)^\top \Sigma^{-1}(\vec{X} - \vec{\mu}_k)$$

The decision boundary can be calculated as:

$$\begin{aligned}
\ln l(\vec{X}|L = 1) &= \ln l(\vec{X}|L = 2) \\
-\frac{d}{2}\ln(2\pi) - \frac{1}{2}\ln|\Sigma| - \frac{1}{2}(\vec{X} - \vec{\mu}_1)^\top \Sigma^{-1}(\vec{X} - \vec{\mu}_1) &= -\frac{d}{2}\ln(2\pi) - \frac{1}{2}\ln|\Sigma| - \frac{1}{2}(\vec{X} - \vec{\mu}_2)^\top \Sigma^{-1}(\vec{X} - \vec{\mu}_2) \\
(\vec{X} - \vec{\mu}_1)^\top \Sigma^{-1}(\vec{X} - \vec{\mu}_1) &= (\vec{X} - \vec{\mu}_2)^\top \Sigma^{-1}(\vec{X} - \vec{\mu}_2) \\
\vec{X}^\top \Sigma^{-1}\vec{X} - 2\vec{\mu}_1^\top \Sigma^{-1}\vec{X} + \vec{\mu}_1^\top \Sigma^{-1}\vec{\mu}_1 &= \vec{X}^\top \Sigma^{-1}\vec{X} - 2\vec{\mu}_2^\top \Sigma^{-1}\vec{X} + \vec{\mu}_2^\top \Sigma^{-1}\vec{\mu}_2 \\
-2\vec{\mu}_1^\top \Sigma^{-1}\vec{X} + \vec{\mu}_1^\top \Sigma^{-1}\vec{\mu}_1 &= -2\vec{\mu}_2^\top \Sigma^{-1}\vec{X} + \vec{\mu}_2^\top \Sigma^{-1}\vec{\mu}_2
\end{aligned}$$

Here we define:

$$f(x) = -2(\vec{\mu}_1^\top - \vec{\mu}_2^\top)\Sigma^{-1}\vec{X} + \vec{\mu}_1^\top \Sigma^{-1}\vec{\mu}_1 - \vec{\mu}_2^\top \Sigma^{-1}\vec{\mu}_2$$

Write out my decision rule here in the required format:

- Choose class 1 if $f(x) > 0$
- Choose class 2 if $f(x) < 0$
- Choose either class 1 or 2 based on your discretion otherwise

---

2. MAP:

$$\ln\Big(P(\vec{X}|L=k)P(L=k)\Big) = -\frac{d}{2}\ln(2\pi) - \frac{1}{2}\ln|\Sigma| - \frac{1}{2}(\vec{X}-\vec{\mu}_k)^\top\Sigma^{-1}(\vec{X}-\vec{\mu}_k) + \ln(\pi_k)$$

The decision boundary can be calculated as:

$$\begin{aligned}
\ln\Big(P(\vec{X}|L=1)P(L=1)\Big) &= \ln\Big(P(\vec{X}|L=2)P(L=2)\Big) \\
-\frac{d}{2}\ln(2\pi) - \frac{1}{2}\ln|\Sigma| - \frac{1}{2}(\vec{X}-\vec{\mu}_1)^\top\Sigma^{-1}(\vec{X}-\vec{\mu}_1) + \ln(\pi_1) &= -\frac{d}{2}\ln(2\pi) - \frac{1}{2}\ln|\Sigma| - \frac{1}{2}(\vec{X}-\vec{\mu}_2)^\top\Sigma^{-1}(\vec{X}-\vec{\mu}_2) + \ln(\pi_2) \\
(\vec{X}-\vec{\mu}_1)^\top\Sigma^{-1}(\vec{X}-\vec{\mu}_1) + \ln(\pi_1) &= (\vec{X}-\vec{\mu}_2)^\top\Sigma^{-1}(\vec{X}-\vec{\mu}_2) + \ln(\pi_2) \\
-2\vec{\mu}_1^\top\Sigma^{-1}\vec{X} + \vec{\mu}_1^\top\Sigma^{-1}\vec{\mu}_1 + \ln(\pi_1) &= -2\vec{\mu}_2^\top\Sigma^{-1}\vec{X} + \vec{\mu}_2^\top\Sigma^{-1}\vec{\mu}_2 + \ln(\pi_2)
\end{aligned}$$

Here we define:

$$f(x) = -2(\vec{\mu}_1^\top - \vec{\mu}_2^\top)\Sigma^{-1}\vec{X} + \vec{\mu}_1^\top\Sigma^{-1}\vec{\mu}_1 - \vec{\mu}_2^\top\Sigma^{-1}\vec{\mu}_2 + \ln(\frac{\pi_1}{\pi_2})$$

Write out my decision rule here in the required format:

- Choose class 1 if $f(x) > 0$
- Choose class 2 if $f(x) < 0$
- Choose either class 1 or 2 based on your discretion otherwise

3. When $\pi_1 = \pi_2 = 0.5$, these two decision rules are the same because $\ln(\frac{\pi_1}{\pi_2}) = 0$.

(b) You should have noticed that the function f is *linear* in its argument $\vec{X}$, and takes the form $\vec{w}^\top(\vec{X}-\vec{v})$. We will now show that CCA defined on a suitable set of random variables leads to precisely the same decision rule.

Let $\vec{Y} \in \mathbb{R}^2$ be a one hot vector denoting the realization of the label l, i.e. $Y_l = 1$ if $L = l$, and zero otherwise. Let $\pi_1 = \pi_2 = 1/2$. **Compute the covariance matrices $\mathbf{\Sigma}_{XX}, \mathbf{\Sigma}_{XY}$ and $\mathbf{\Sigma}_{YY}$ as a function of $\vec{\mu}_1, \vec{\mu}_2, \mathbf{\Sigma}$.** Recall that the random variables are not zero-mean. Hint: when computing the covariance matrices, the tower property of the expectation is useful.

$$
\begin{aligned}
\Sigma_{XX} &= \mathbb{E}\left[(\vec{X}-\vec{\mu}_1\pi_1-\vec{\mu}_2\pi_2)(\vec{X}-\vec{\mu}_1\pi_1-\vec{\mu}_2\pi_2)^\top\right] \\
&= \mathbb{E}\left[\vec{X}\vec{X}^\top\right] - (\vec{\mu}_1\pi_1+\vec{\mu}_2\pi_2)(\vec{\mu}_1\pi_1+\vec{\mu}_2\pi_2)^\top \\
&= \mathbb{E}\left[\vec{X}\vec{X}^\top\right] - \frac{1}{4}(\vec{\mu}_1+\vec{\mu}_2)(\vec{\mu}_1+\vec{\mu}_2)^\top \\
&= \frac{1}{2}\mathbb{E}\left[\vec{X}\vec{X}^\top|L=1\right] + \frac{1}{2}\mathbb{E}\left[\vec{X}\vec{X}^\top|L=2\right] - \frac{1}{4}(\vec{\mu}_1+\vec{\mu}_2)(\vec{\mu}_1+\vec{\mu}_2)^\top \\
&= \frac{1}{2}(\Sigma+\vec{\mu}_1\vec{\mu}_1^\top) + \frac{1}{2}(\Sigma+\vec{\mu}_2\vec{\mu}_2^\top) - \frac{1}{4}(\vec{\mu}_1+\vec{\mu}_2)(\vec{\mu}_1+\vec{\mu}_2)^\top \\
&= \Sigma + \frac{1}{4}\vec{\mu}_1\vec{\mu}_1^\top - \frac{1}{4}\vec{\mu}_1\vec{\mu}_2^\top - \frac{1}{4}\vec{\mu}_2\vec{\mu}_1^\top + \frac{1}{4}\vec{\mu}_2\vec{\mu}_2^\top \\
&= \Sigma + \frac{1}{4}(\vec{\mu}_1-\vec{\mu}_2)(\vec{\mu}_1-\vec{\mu}_2)^\top
\end{aligned}
$$

$$
\begin{aligned}
\Sigma_{XY} &= \mathbb{E}\left[\mathbb{E}\left[(\vec{X}-\vec{\mu}_1\pi_1-\vec{\mu}_2\pi_2)(\vec{Y}-\frac{1}{2})^\top|L=l\right]\right] \\
&= \frac{1}{2}\mathbb{E}\left[(\vec{X}-\frac{1}{2}(\vec{\mu}_1+\vec{\mu}_2))(\vec{Y}-\frac{1}{2})^\top|L=1\right] + \frac{1}{2}\mathbb{E}\left[(\vec{X}-\frac{1}{2}(\vec{\mu}_1+\vec{\mu}_2))(\vec{Y}-\frac{1}{2})^\top|L=2\right] \\
&= \frac{1}{2}\mathbb{E}\left[(\vec{X}-\frac{1}{2}(\vec{\mu}_1+\vec{\mu}_2))\left(\begin{bmatrix}\frac{1}{2} & -\frac{1}{2}\end{bmatrix}\right)|L=1\right] + \frac{1}{2}\mathbb{E}\left[(\vec{X}-\frac{1}{2}(\vec{\mu}_1+\vec{\mu}_2))\left(\begin{bmatrix}-\frac{1}{2} & \frac{1}{2}\end{bmatrix}\right)|L=2\right] \\
&= \frac{1}{2}(\mathbb{E}[\vec{X}|L=1]-\frac{1}{2}(\vec{\mu}_1+\vec{\mu}_2))\left(\begin{bmatrix}\frac{1}{2} & -\frac{1}{2}\end{bmatrix}\right) + \frac{1}{2}(\mathbb{E}[\vec{X}|L=2]-\frac{1}{2}(\vec{\mu}_1+\vec{\mu}_2))\left(\begin{bmatrix}-\frac{1}{2} & \frac{1}{2}\end{bmatrix}\right) \\
&= \frac{1}{2}(\vec{\mu}_1-\frac{1}{2}(\vec{\mu}_1+\vec{\mu}_2))\left(\begin{bmatrix}\frac{1}{2} & -\frac{1}{2}\end{bmatrix}\right) + \frac{1}{2}(\vec{\mu}_2-\frac{1}{2}(\vec{\mu}_1+\vec{\mu}_2))\left(\begin{bmatrix}-\frac{1}{2} & \frac{1}{2}\end{bmatrix}\right) \\
&= \frac{1}{2}\times\frac{1}{2}(\vec{\mu}_1-\vec{\mu}_2)\begin{bmatrix}\frac{1}{2} & -\frac{1}{2}\end{bmatrix} + \frac{1}{2}\times\frac{1}{2}(\vec{\mu}_2-\vec{\mu}_1)\begin{bmatrix}-\frac{1}{2} & \frac{1}{2}\end{bmatrix} \\
&= \frac{1}{4}\begin{bmatrix}\frac{\vec{\mu}_1-\vec{\mu}_2}{2} & -\frac{\vec{\mu}_1-\vec{\mu}_2}{2}\end{bmatrix} + \frac{1}{4}\begin{bmatrix}-\frac{\vec{\mu}_2-\vec{\mu}_1}{2} & \frac{\vec{\mu}_2-\vec{\mu}_1}{2}\end{bmatrix} \\
&= \frac{1}{4}\begin{bmatrix}(\vec{\mu}_1-\vec{\mu}_2) & -(\vec{\mu}_1-\vec{\mu}_2)\end{bmatrix}
\end{aligned}
$$

$$
\begin{aligned}
\Sigma_{YY} &= \mathbb{E}\left[(\vec{Y}-\frac{1}{2})(\vec{Y}-\frac{1}{2})^\top\right] \\
&= \mathbb{E}\left[\vec{Y}\vec{Y}^\top\right] - \frac{1}{4} \\
&= \begin{bmatrix}1\times\frac{1}{2}+0\times\frac{1}{2} & 0 \\ 0 & 1\times\frac{1}{2}+0\times\frac{1}{2}\end{bmatrix} - \frac{1}{4}\mathbb{I}
\end{aligned}
$$

$$= \begin{bmatrix} \frac{1}{4} & -\frac{1}{4} \\ -\frac{1}{4} & \frac{1}{4} \end{bmatrix}$$

(c) Let us now perform CCA on the two random variables. Recall that in order to find the first canonical directions, we look for vectors $\vec{u} \in \mathbb{R}^d$ and $\vec{v} \in \mathbb{R}^2$ such that $\rho(\vec{u}^\top \vec{X}, \vec{v}^\top \vec{Y})$ is maximized.

Show that the maximizing $\vec{u}^*$ is proportional to $\boldsymbol{\Sigma}^{-1}(\vec{\mu}_1 - \vec{\mu}_2)$. Recall that $\vec{u}^*$ is that "direction" of $\vec{X}$ that contributes most to predicting $\vec{Y}$. **What is the relationship between $\vec{u}^*$ and the function $f(\vec{X})$ computed in part (a)?**

Hint: The Sherman-Morrison formula for matrix inversion may be useful:

Suppose $\mathbf{A} \in \mathbb{R}^{d \times d}$ is an invertible square matrix and $\vec{a}, \vec{b} \in \mathbb{R}^d$ are column vectors. Then,

$$(\mathbf{A} + \vec{a}\vec{b}^T)^{-1} = \mathbf{A}^{-1} - \frac{\mathbf{A}^{-1}\vec{a}\vec{b}^T\mathbf{A}^{-1}}{1 + \vec{b}^T\mathbf{A}^{-1}\vec{a}}.$$

According to Wikipedia page, we have $\vec{a} = \Sigma_{XX}^{1/2}\vec{u}$

$$\begin{aligned}
\rho &= \frac{\vec{a}^\top \Sigma_{XX}^{-1/2} \Sigma_{XY} \vec{v}}{\sqrt{\vec{a}^\top \vec{a}} \sqrt{\vec{v}^\top \Sigma_{YY} \vec{v}}} \\
&= \frac{\vec{a}^\top \Sigma_{XX}^{-1/2} \frac{1}{4} \begin{bmatrix} (\vec{\mu}_1 - \vec{\mu}_2) & -(\vec{\mu}_1 - \vec{\mu}_2) \end{bmatrix} \vec{v}}{\sqrt{\vec{a}^\top \vec{a}} \sqrt{\vec{v}^\top \begin{bmatrix} \frac{1}{4} & -\frac{1}{4} \\ -\frac{1}{4} & \frac{1}{4} \end{bmatrix} \vec{v}}} \\
&= \frac{\vec{a}^\top \Sigma_{XX}^{-1/2} \frac{1}{4} (\vec{\mu}_1 - \vec{\mu}_2) \begin{bmatrix} 1 & -1 \end{bmatrix} \vec{v}}{\sqrt{\vec{a}^\top \vec{a}} \sqrt{\vec{v}^\top \frac{1}{4} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \vec{v}}} \\
&= \frac{\vec{a}^\top \Sigma_{XX}^{-1/2} \frac{1}{2} (\vec{\mu}_1 - \vec{\mu}_2) \begin{bmatrix} 1 & -1 \end{bmatrix} \vec{v}}{\sqrt{\vec{a}^\top \vec{a}} \sqrt{\vec{v}^\top \begin{bmatrix} 1 \\ -1 \end{bmatrix} \begin{bmatrix} 1 & -1 \end{bmatrix} \vec{v}}} \\
&= \frac{\vec{a}^\top \Sigma_{XX}^{-1/2} \frac{1}{2} (\vec{\mu}_1 - \vec{\mu}_2) \begin{bmatrix} 1 & -1 \end{bmatrix} \vec{v}}{\sqrt{\vec{a}^\top \vec{a}} \begin{bmatrix} 1 & -1 \end{bmatrix} \vec{v}} \\
&= \frac{\vec{a}^\top \Sigma_{XX}^{-1/2} (\vec{\mu}_1 - \vec{\mu}_2)}{2\sqrt{\vec{a}^\top \vec{a}}}
\end{aligned}$$

By the CauchySchwarz inequality, we have:

$$\begin{aligned}
\rho &= \frac{\vec{a}^\top \Sigma_{XX}^{-1/2} (\vec{\mu}_1 - \vec{\mu}_2)}{2\sqrt{\vec{a}^\top \vec{a}}} \\
&\leq \frac{\vec{a}^\top \vec{a} (\vec{\mu}_1 - \vec{\mu}_2)^\top \Sigma_{XX}^{-1/2} \Sigma_{XX}^{-1/2} (\vec{\mu}_1 - \vec{\mu}_2)}{2\sqrt{\vec{a}^\top \vec{a}}}
\end{aligned}$$

The equality is true if $\vec{a}^* \propto \Sigma_{XX}^{-1/2}(\vec{\mu}_1 - \vec{\mu}_2)$. Now let's write this proportionality:

$$\begin{aligned}
\vec{a}^* &\propto \Sigma_{XX}^{-1/2}(\vec{\mu}_1 - \vec{\mu}_2) \\
\Sigma_{XX}^{1/2}\vec{u}^* &\propto \Sigma_{XX}^{-1/2}(\vec{\mu}_1 - \vec{\mu}_2) \\
\vec{u}^* &\propto \Sigma_{XX}^{-1}(\vec{\mu}_1 - \vec{\mu}_2)
\end{aligned}$$

Now we compute Σ_{XX}^{-1} using the hint (Note $\vec{\mu} = \frac{1}{2}(\vec{\mu}_1 - \vec{\mu}_2)$):

$$\begin{aligned}
&\Sigma_{XX}^{-1/2} \\
=&\left((\Sigma + \frac{1}{4}(\vec{\mu}_1 - \vec{\mu}_2)(\vec{\mu}_1 - \vec{\mu}_2)^\top)^{-1}\right)^{1/2} \\
=&(\Sigma + \vec{\mu}\vec{\mu}^\top)^{-1} \\
=&\Sigma^{-1} - \frac{\Sigma^{-1}\vec{\mu}\vec{\mu}^T\Sigma^{-1}}{1 + \vec{\mu}^T\Sigma^{-1}\vec{\mu}}
\end{aligned}$$

Substitute into the conclusion above gives us:

$$\begin{aligned}
\vec{u^*} &\propto \Sigma_{XX}^{-1}(\vec{\mu}_1 - \vec{\mu}_2) \\
\vec{u^*} &\propto \left(\Sigma^{-1} - \frac{\Sigma^{-1}\vec{\mu}\vec{\mu}^T\Sigma^{-1}}{1 + \vec{\mu}^T\Sigma^{-1}\vec{\mu}}\right) 2\vec{\mu} \\
\vec{u^*} &\propto \Sigma_{XX}^{-1}(\vec{\mu}_1 - \vec{\mu}_2) \\
\vec{u^*} &\propto \Sigma^{-1}\vec{\mu} - \frac{1}{1 + \vec{\mu}^T\Sigma^{-1}\vec{\mu}}\Sigma^{-1}\vec{\mu}\left(\vec{\mu}^T\Sigma^{-1}\vec{\mu}\right) \\
\vec{u^*} &\propto \Sigma^{-1}\vec{\mu} - \frac{\vec{\mu}^T\Sigma^{-1}\vec{\mu}}{1 + \vec{\mu}^T\Sigma^{-1}\vec{\mu}}\Sigma^{-1}\vec{\mu}
\end{aligned}$$

We notice that $\frac{\vec{\mu}^T\Sigma^{-1}\vec{\mu}}{1+\vec{\mu}^T\Sigma^{-1}\vec{\mu}}$ is a scalar so we have proved that $\vec{u^*} \propto \Sigma^{-1}(\vec{\mu}_1 - \vec{\mu}_2)$. (Caveat: the scalar here depends on)

Copy function $f(x)$ down here:

$$f(x) = -2(\vec{\mu}_1^\top - \vec{\mu}_2^\top)\Sigma^{-1}\vec{X} + \vec{\mu}_1^\top\Sigma^{-1}\vec{\mu}_1 - \vec{\mu}_2^\top\Sigma^{-1}\vec{\mu}_2$$

We can see that the slope of the decision boundary is the direction $\vec{u^*}$ we found. That is, we have found the same direction as our decision boundary.

Question 4. Sensors, Objects, and Localization (Part 2)

Let us say there are n objects and m sensors located in a $2d$ plane. The n objects are located at the points $(x_1, y_1), \ldots, (x_n, y_n)$. The m sensors are located at the points $(a_1, b_1), \ldots, (a_m, b_m)$. We have measurements for the distances between the objects and the sensors: D_{ij} is the measured distance from object i to sensor j. The distance measurement has noise in it. Specifically, we model

$$D_{ij} = ||(x_i, y_i) - (a_j, b_j)|| + Z_{ij},$$

where $Z_{ij} \sim N(0, 1)$. The noise is independent across different measurements.

Starter code has been provided for data generation to aid your explorations. All Python libraries are permitted, You will need to plot ten figures in total in this problem.

Assume we observe $D_{ij} = d_{ij}$ with $(X_i, Y_i) = (x_i, y_i)$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. Here, $m = 7$. **Our goal is to predict $(X_{i'}, Y_{i'})$ from newly observed $D_{i'1}, \ldots, D_{i'7}$.** For a data set with q points, the error is measured by the average distance between the predicted object locations and the true object locations,

$$\frac{1}{q} \sum_{i=1}^{q} \sqrt{(\hat{x}_i - x_i)^2 + (\hat{y}_i - y_i)^2},$$

where $(\hat{x}_i, \hat{y}_i)$ is the location of objects predicted by a model.

We are going to consider five models in this problem and compare their performance:

- A *generative model*: This is basically from the earlier assignment: we first estimate sensor locations from the training data and then use the estimated sensor locations to estimate the new object locations.
- A *Linear Model*. Using the training set, the linear model attempts to fit (X_i, Y_i) directly from the distance measurements $(D_{i1}, \ldots, D_{i7})$. Then it predicts $(X_{i'}, Y_{i'})$ from $(D_{i'1}, \ldots, D_{i'7})$ using the map that it found during training. (It never tries to explicitly model the underlying sensor locations.)
- A *Second-Order Polynomial Regression Model*. The set-up is similar to the linear model, but including second-order polynomial features.
- A *Third-Order Polynomial Regression Model*. The set-up is similar to the linear model, but including third-order polynomial features.
- A *Neural Network Model*. The Neural Network should have two hidden layers, each with 100 neurons, and use `Relu` as the non-linearity. (You are encouraged to explore on your own beyond this however. These parameters were chosen to teach you a hype-deflating lesson.) The neural net approach also follows the principle of finding/learning a direct connection between the distance measurements and the object location.

(a) **Implement the last four models listed above in `models.py`.** Starter code has been provided for data generation and visualization to aid your explorations. We provide you a simple gradient descent framework for you to implement the neural network, but you are also free to use the TensorFlow and PyTorch code from your previous homework.

```
import numpy as np
import scipy.spatial
from starter import *


#####################################################################
## Models used for predictions.
#####################################################################
def compute_update(single_obj_loc, sensor_loc, single_distance):
"""
Compute the gradient of the log-likelihood function for part a.

Input:
single_obj_loc: 1 * d numpy array.
Location of the single object.

sensor_loc: k * d numpy array.
Location of sensor.

single_distance: k dimensional numpy array.
Observed distance of the object.

Output:
grad: d-dimensional numpy array.

"""
loc_difference = single_obj_loc - sensor_loc # k * d.
phi = np.linalg.norm(loc_difference, axis=1) # k.
grad = loc_difference / np.expand_dims(phi, 1) # k * 2.
update = np.linalg.solve(grad.T.dot(grad), grad.T.dot(single_distance - phi))

return update


def get_object_location(sensor_loc, single_distance, num_iters=20, num_repeats=10):
"""
Compute the gradient of the log-likelihood function for part a.

Input:

sensor_loc: k * d numpy array. Location of sensor.

single_distance: k dimensional numpy array.
Observed distance of the object.

Output:
obj_loc: 1 * d numpy array. The mle for the location of the object.

"""
obj_locs = np.zeros((num_repeats, 1, 2))
distances = np.zeros(num_repeats)
for i in range(num_repeats):
obj_loc = np.random.randn(1, 2) * 100
for t in range(num_iters):
obj_loc += compute_update(obj_loc, sensor_loc, single_distance)

distances[i] = np.sum((single_distance - np.linalg.norm(obj_loc - sensor_loc, axis=1))**2)
obj_locs[i] = obj_loc

obj_loc = obj_locs[np.argmin(distances)]

return obj_loc[0]
```

```
def generative_model(X, Y, Xs_test, Ys_test):
"""
This function implements the generative model.
Input:
X: independent variables in training data.
Y: dependent variables in training data.
Xs_test: independent variables in test data.
Ys_test: dependent variables in test data.
Output:
mse: Mean square error on test data.
"""
initial_sensor_loc = np.random.randn(7, 2) * 100
estimated_sensor_loc = find_mle_by_grad_descent_part_e(
initial_sensor_loc, Y, X, lr=0.001, num_iters=1000)

mses = []
for i, X_test in enumerate(Xs_test):
Y_test = Ys_test[i]
Y_pred = np.array(
[get_object_location(estimated_sensor_loc, X_test_single) for X_test_single in X_test])
mse = np.mean(np.sqrt(np.sum((Y_pred - Y_test)**2, axis=1)))
mses.append(mse)
return mses


def oracle_model(X, Y, Xs_test, Ys_test, sensor_loc):
"""
This function implements the generative model.
Input:
X: independent variables in training data.
Y: dependent variables in training data.
Xs_test: independent variables in test data.
Ys_test: dependent variables in test data.
sensor_loc: location of the sensors.
Output:
mse: Mean square error on test data.
"""
mses = []
for i, X_test in enumerate(Xs_test):
Y_test = Ys_test[i]
Y_pred = np.array([
get_object_location(sensor_loc, X_test_single)
for X_test_single in X_test
])
mse = np.mean(np.sqrt(np.sum((Y_pred - Y_test)**2, axis=1)))
mses.append(mse)
return mses


def construct_second_order_data(X):
"""
This function computes second order variables
for polynomial regression.
Input:
X: Independent variables.
Output:
A data matrix composed of both first and second order terms.
"""
X_second_order = []
m = X.shape[1]
for i in range(m):
for j in range(m):
```

```
if j <= i:
X_second_order.append(X[:, i] * X[:, j])
X_second_order = np.array(X_second_order).T
return np.concatenate((X, X_second_order), axis=1)


def linear_regression(X, Y, Xs_test, Ys_test):
"""
This function performs linear regression.
Input:
X: independent variables in training data.
Y: dependent variables in training data.
Xs_test: independent variables in test data.
Ys_test: dependent variables in test data.
Output:
mse: Mean square error on test data.
"""

## YOUR CODE HERE
##################
X = np.hstack((np.ones((X.shape[0], 1)), X))
#w = np.linalg.solve(X.T @ X, X.T @ Y)
w = np.linalg.lstsq(X, Y)[0]
mses = []
for i, X_test in enumerate(Xs_test):
Y_test = Ys_test[i]
X_test = np.hstack((np.ones((X_test.shape[0], 1)), X_test))
Y_pred = np.array(X_test @ w)
mse = np.mean(np.sqrt(np.sum((Y_pred - Y_test) ** 2, axis=1)))
mses.append(mse)
return mses


def linear_regression_no_bias(X, Y, Xs_test, Ys_test):
"""
This function performs linear regression.
Input:
X: independent variables in training data.
Y: dependent variables in training data.
Xs_test: independent variables in test data.
Ys_test: dependent variables in test data.
Output:
mse: Mean square error on test data.
"""

## YOUR CODE HERE
##################
w = np.linalg.lstsq(X, Y)[0]
mses = []
for i, X_test in enumerate(Xs_test):
Y_test = Ys_test[i]
Y_pred = np.array(X_test @ w)
mse = np.mean(np.sqrt(np.sum((Y_pred - Y_test) ** 2, axis=1)))
mses.append(mse)
return mses


def polynomial(x, D):
n_feature = x.shape[1]
Q = [(np.ones(x.shape[0]), 0, 0)]
i = 0
while Q[i][1] < D:
cx, degree, last_index = Q[i]
```

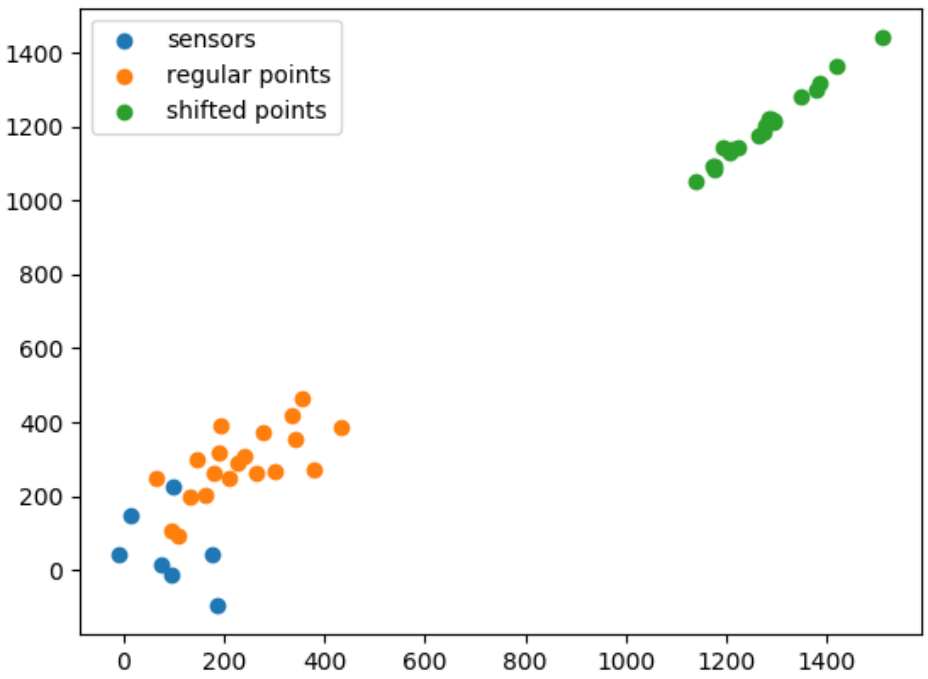
```
for j in range(last_index, n_feature):
Q.append((cx * x[:, j], degree + 1, j))
i += 1
return np.column_stack([q[0] for q in Q])


def poly_regression_second(X, Y, Xs_test, Ys_test):
"""
This function performs second order polynomial regression.
Input:
X: independent variables in training data.
Y: dependent variables in training data.
Xs_test: independent variables in test data.
Ys_test: dependent variables in test data.
Output:
mse: Mean square error on test data.
"""
## YOUR CODE HERE
##################
Xs_test_poly = []
for i, X_test in enumerate(Xs_test):
Xs_test_poly.append(polynomial(X_test, 2))

return linear_regression_no_bias(polynomial(X, 2), Y, Xs_test_poly, Ys_test)


def poly_regression_cubic(X, Y, Xs_test, Ys_test):
"""
This function performs third order polynomial regression.
Input:
X: independent variables in training data.
Y: dependent variables in training data.
Xs_test: independent variables in test data.
Ys_test: dependent variables in test data.
Output:
mse: Mean square error on test data.
"""
## YOUR CODE HERE
##################
X = np.array(X)
Y = np.array(Y)
Xs_test_poly = []
for i, X_test in enumerate(Xs_test):
Xs_test_poly.append(polynomial(X_test, 3))
return linear_regression_no_bias(polynomial(X, 3), Y, Xs_test_poly, Ys_test)


def neural_network(X, Y, Xs_test, Ys_test):
"""
This function performs neural network prediction.
Input:
X: independent variables in training data.
Y: dependent variables in training data.
Xs_test: independent variables in test data.
Ys_test: dependent variables in test data.
Output:
mse: Mean square error on test data.
"""
## YOUR CODE HERE
##################
# Copied from backprop_sol
meanX = np.mean(X, axis=0)
stdX = np.std(X, axis=0)
```

```
meanY = np.mean(Y, axis=0)
stdY = np.std(Y, axis=0)
X = (X - meanX) / stdX
Y = (Y - meanY) / stdY
activations = dict(ReLU=ReLUActivation,
tanh=TanhActivation,
linear=LinearActivation)
lr = dict(ReLU=0.1, tanh=0.02, linear=0.005)
iterations = 2000
names = ['ReLU', 'linear', 'tanh']
key = names[0]
#### PART G ####
activation = activations[key]
model = Model(X.shape[1])
model.addLayer(DenseLayer(100, activation()))
model.addLayer(DenseLayer(100, activation()))
model.addLayer(DenseLayer(Y.shape[1], LinearActivation()))
model.initialize(QuadraticCost())

# Train the model and display the results
model.train(X, Y, iterations, GDOptimizer(eta=lr[key]))
mses = []
for i, X_test in enumerate(Xs_test):
Y_test = Ys_test[i]
Y_pred = model.predict((X_test - meanX) / stdX) * stdY + meanY
mse = np.mean(np.sqrt(np.sum((Y_pred - Y_test) ** 2, axis=1)))
mses.append(mse)
return mses
```

(b) **Fix a set of 7 sensors and generate the following data sets:**

- 15 training sets where n_{train} varies from 10 to 290 in increments of 20.
- A "regular" testing data set where $n_{\text{test}} = 1000$.
- A "shifted" testing data set where $n_{\text{test}} = 1000$. You can do this by setting original_dist to `False` in the function `generate_data` in `starter.py`.

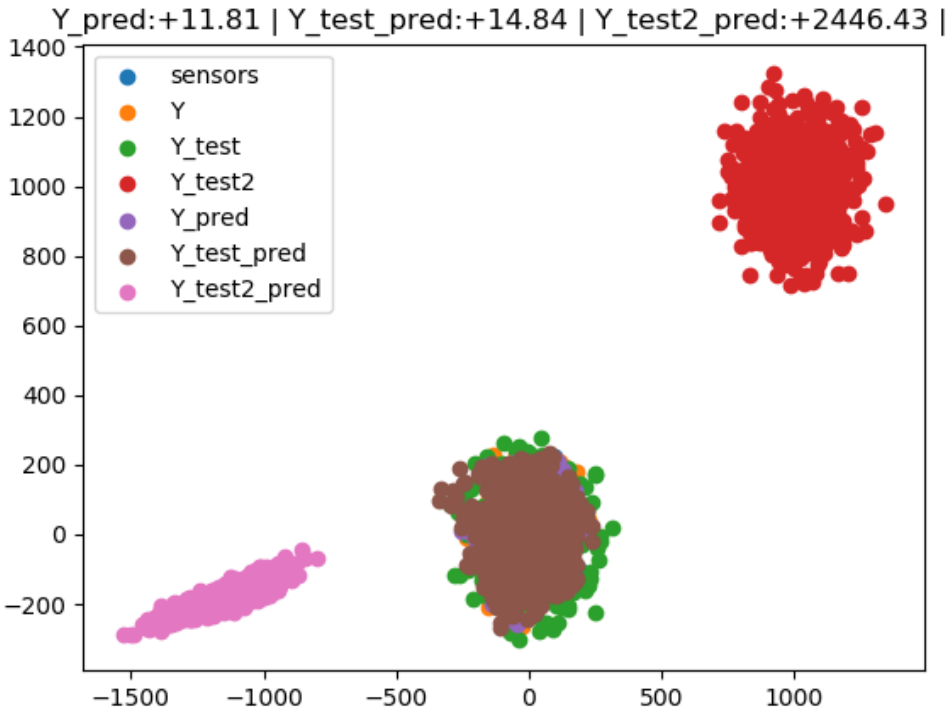
The difference between the "regular" testing data and the "shifted" testing data is that the "regular" testing data is drawn from the same distribution as the training data, whereas the "shifted" testing data is farther away.
Run `plot0.py` to visualize the sensor location, sampled regular data, and sampled shifted data. **Attach the plot**.




```
import matplotlib.pyplot as plt
import numpy as np

from models import *
from starter import *


def main():
np.random.seed(0)
sensor_loc = generate_sensors()
regular_loc, _ = generate_dataset(
sensor_loc,
num_sensors=sensor_loc.shape[0],
```

```
spatial_dim=2,
num_data=20,
original_dist=True,
noise=1)
shifted_loc, _ = generate_dataset(
sensor_loc,
num_sensors=sensor_loc.shape[0],
spatial_dim=2,
num_data=20,
original_dist=False,
noise=1)

plt.scatter(sensor_loc[:, 0], sensor_loc[:, 1], label="sensors")
plt.scatter(regular_loc[:, 0], regular_loc[:, 1], label="regular points")
plt.scatter(shifted_loc[:, 0], shifted_loc[:, 1], label="shifted points")
plt.legend()
plt.savefig("Figure_4a-dataset.png")
# plt.show()


if __name__ == "__main__":
main()
```

EXTRA: I visualize the predictions of my neural network and get the graph below. We can see that neural net in this case is terrible at predicting the shifted data points:



My code for debugging

```
import matplotlib.pyplot as plt
import numpy as np

from models import *
from starter import *


def neural_network_debug(X, Y, Xs_test, Ys_test,
lr=0.0001, iterations=5000, num_neurons=100, num_layers=2, activation='ReLU'):
"""
This function performs neural network prediction.
Input:
X: independent variables in training data.
Y: dependent variables in training data.
Xs_test: independent variables in test data.
Ys_test: dependent variables in test data.
Output:
mse: Mean square error on test data.
"""
## YOUR CODE HERE
##################
# Copied from backprop_sol
meanX = np.mean(X, axis=0)
stdX = np.std(X, axis=0)
meanY = np.mean(Y, axis=0)
stdY = np.std(Y, axis=0)
X = (X - meanX) / stdX
Y = (Y - meanY) / stdY
activations = dict(ReLU=ReLUActivation,
tanh=TanhActivation,
```

```
linear=LinearActivation)

#### PART G ####
activation_func = activations[activation]
model = Model(X.shape[1])
for iLayer in range(num_layers):
model.addLayer(DenseLayer(num_neurons, activation_func()))
model.addLayer(DenseLayer(Y.shape[1], LinearActivation()))
model.initialize(QuadraticCost())

# Train the model and display the results
# model.trainBatch(X, Y, int(iterations / 10), iterations, GDOptimizer(eta=lr))
model.train(X, Y, iterations, GDOptimizer(eta=lr))
mses = []
Y_predicts = []
for i, X_test in enumerate(Xs_test):
Y_test = Ys_test[i]
Y_pred = model.predict((X_test - meanX) / stdX) * stdY + meanY
mse = np.mean(np.sqrt(np.sum((Y_pred - Y_test) ** 2, axis=1)))
mses.append(mse)
Y_predicts.append(Y_pred)
return mses, Y_predicts


def main():
np.random.seed(0)
def generate_data(sensor_loc, k=7, d=2, n=1, original_dist=True, noise=1):
return generate_dataset(
sensor_loc,
num_sensors=k,
spatial_dim=d,
num_data=n,
original_dist=original_dist,
noise=noise)

sensor_loc = generate_sensors()
X_test, Y_test = generate_data(sensor_loc, n=1000)
X_test2, Y_test2 = generate_data(
sensor_loc, n=1000, original_dist=False)
X, Y = generate_data(sensor_loc, n=200)
Xs_test, Ys_test = [X, X_test, X_test2], [Y, Y_test, Y_test2]
# int(np.sqrt(10000 / 40))
mses, Y_predictions = \
neural_network_debug(X, Y, Xs_test, Ys_test, lr=0.1, iterations=2000,
num_layers=2, num_neurons=100, activation='ReLU')

plt.scatter(sensor_loc[:, 0], sensor_loc[:, 1], label="sensors")
plt.scatter(Y[:, 0], Y[:, 1], label="Y")
plt.scatter(Y_test[:, 0], Y_test[:, 1], label="Y_test")
plt.scatter(Y_test2[:, 0], Y_test2[:, 1], label="Y_test2")

#Y_predictions = [Y_predictions[0]]
pred_labels = ['Y_pred', 'Y_test_pred', 'Y_test2_pred']

plt.title(''.join(label + ":{:+.2f} | ".format(mses[i]) for i, label in enumerate(pred_labels)))
#print(Y_predictions[0])
for i, y_predict in enumerate(Y_predictions):
plt.scatter(y_predict[:, 0], y_predict[:, 1], label=pred_labels[i])

plt.legend()
plt.savefig("Figure_4b-debug_test.png")
plt.show()
```

```
if __name__ == "__main__":
main()
```

I also modified the starter code, which is attached below:

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import scipy.spatial


# Gradient descent optimization
# The learning rate is specified by eta
class GDOptimizer(object):
def __init__(self, eta):
self.eta = eta

def initialize(self, layers):
pass

# This function performs one gradient descent step
# layers is a list of dense layers in the network
# g is a list of gradients going into each layer before the nonlinear activation
# a is a list of of the activations of each node in the previous layer going
def update(self, layers, g, a):
m = a[0].shape[1]
for layer, curGrad, curA in zip(layers, g, a):
layer.updateWeights(-self.eta / m * np.dot(curGrad, curA.T))
layer.updateBias(-self.eta / m * np.sum(curGrad, 1).reshape(layer.b.shape))


# Cost function used to compute prediction errors
class QuadraticCost(object):

# Compute the squared error between the prediction yp and the observation y
# This method should compute the cost per element such that the output is the
# same shape as y and yp
@staticmethod
def fx(y, yp):
return 1 / y.shape[1] * np.square(yp - y)

# Derivative of the cost function with respect to yp
@staticmethod
def dx(y, yp):
return 2 / y.shape[1] * (yp - y)


# Sigmoid function fully implemented as an example
class SigmoidActivation(object):
@staticmethod
def fx(z):
return 1 / (1 + np.exp(-z))

@staticmethod
def dx(z):
return SigmoidActivation.fx(z) * (1 - SigmoidActivation.fx(z))


# Hyperbolic tangent function
class TanhActivation(object):
```

```
# Compute tanh for each element in the input z
@staticmethod
def fx(z):
return np.tanh(z)

# Compute the derivative of the tanh function with respect to z
@staticmethod
def dx(z):
return 1 - np.square(np.tanh(z))


# Rectified linear unit
class ReLUActivation(object):
@staticmethod
def fx(z):
return np.maximum(0, z)

@staticmethod
def dx(z):
return (z > 0).astype('float')


# Linear activation
class LinearActivation(object):
@staticmethod
def fx(z):
return z

@staticmethod
def dx(z):
return np.ones(z.shape)


# This class represents a single hidden or output layer in the neural network
class DenseLayer(object):

# numNodes: number of hidden units in the layer
# activation: the activation function to use in this layer
def __init__(self, numNodes, activation):
self.numNodes = numNodes
self.activation = activation

def getNumNodes(self):
return self.numNodes

# Initialize the weight matrix of this layer based on the size of the matrix W
def initialize(self, fanIn, scale=1.0):
s = scale * np.sqrt(6.0 / (self.numNodes + fanIn))
self.W = np.random.normal(0, s, (self.numNodes, fanIn))
self.b = np.random.uniform(-1, 1, (self.numNodes, 1))

# Apply the activation function of the layer on the input z
def a(self, z):
return self.activation.fx(z)

# Compute the linear part of the layer
# The input a is an n x k matrix where n is the number of samples
# and k is the dimension of the previous layer (or the input to the network)
def z(self, a):
return self.W.dot(a) + self.b # Note, this is implemented where we assume a is k x n

# Compute the derivative of the layer's activation function with respect to z
# where z is the output of the above function.
```

```
# This derivative does not contain the derivative of the matrix multiplication
# in the layer. That part is computed below in the model class.
def dx(self, z):
return self.activation.dx(z)

# Update the weights of the layer by adding dW to the weights
def updateWeights(self, dW):
self.W = self.W + dW

# Update the bias of the layer by adding db to the bias
def updateBias(self, db):
self.b = self.b + db


# This class handles stacking layers together to form the completed neural network
class Model(object):

# inputSize: the dimension of the inputs that go into the network
def __init__(self, inputSize):
self.layers = []
self.inputSize = inputSize

# Add a layer to the end of the network
def addLayer(self, layer):
self.layers.append(layer)

# Get the output size of the layer at the given index
def getLayerSize(self, index):
if index >= len(self.layers):
return self.layers[-1].getNumNodes()
elif index < 0:
return self.inputSize
else:
return self.layers[index].getNumNodes()

# Initialize the weights of all of the layers in the network and set the cost
# function to use for optimization
def initialize(self, cost, initializeLayers=True):
self.cost = cost
if initializeLayers:
for i in range(0, len(self.layers)):
if i == len(self.layers) - 1:
self.layers[i].initialize(self.getLayerSize(i - 1))
else:
self.layers[i].initialize(self.getLayerSize(i - 1))

# Compute the output of the network given some input a
# The matrix a has shape n x k where n is the number of samples and
# k is the dimension
# This function returns
# yp - the output of the network
# a - a list of inputs for each layer of the newtork where
#     a[i] is the input to layer i
# z - a list of values for each layer after evaluating layer.z(a) but
#     before evaluating the nonlinear function for the layer
def evaluate(self, x):
curA = x.T
a = [curA]
z = []
for layer in self.layers:
z.append(layer.z(curA))
curA = layer.a(z[-1])
a.append(curA)
```

```
yp = a.pop()
return yp, a, z

# Compute the output of the network given some input a
# The matrix a has shape n x k where n is the number of samples and
# k is the dimension
def predict(self, a):
a, _, _ = self.evaluate(a)
return a.T

# Train the network given the inputs x and the corresponding observations y
# The network should be trained for numEpochs iterations using the supplied
# optimizer
def train(self, x, y, numEpochs, optimizer):

# Initialize some stuff
hist = []
optimizer.initialize(self.layers)

# Run for the specified number of epochs
for epoch in range(0, numEpochs):

# Feed forward
# Save the output of each layer in the list a
# After the network has been evaluated, a should contain the
# input x and the output of each layer except for the last layer
yp, a, z = self.evaluate(x)

# Compute the error
C = self.cost.fx(y.T, yp)
d = self.cost.dx(y.T, yp)
grad = []

# Backpropogate the error
idx = len(self.layers)
for layer, curZ in zip(reversed(self.layers), reversed(z)):
idx = idx - 1
# Here, we compute dMSE/dz_i because in the update
# function for the optimizer, we do not give it
# the z values we compute from evaluating the network
grad.insert(0, np.multiply(d, layer.dx(curZ)))
d = np.dot(layer.W.T, grad[0])

# Update the errors
optimizer.update(self.layers, grad, a)

# Compute the error at the end of the epoch
yh = self.predict(x)
C = self.cost.fx(y.T, yh.T)
C = np.mean(C)
hist.append(C)
return hist

def trainBatch(self, x, y, batchSize, numEpochs, optimizer):

# Copy the data so that we don't affect the original one when shuffling
x = x.copy()
y = y.copy()
hist = []
n = x.shape[0]

for epoch in np.arange(0, numEpochs):
```

```
# Shuffle the data
r = np.random.choice(n, n, replace=False)
x = x[r, :]
y = y[r, :]
e = []

# Split the data in chunks and run SGD
for i in range(0, n, batchSize):
end = min(i + batchSize, n)
batchX = x[i:end, :]
batchY = y[i:end, :]
e += self.train(batchX, batchY, 1, optimizer)
hist.append(np.mean(e))

return hist


############################################################################
######### Part b ###########################################################
############################################################################


############################################################################
######### Gradient Computing and MLE ######################################
############################################################################
def compute_gradient_of_likelihood(single_obj_loc, sensor_loc, single_distance, noise=1):
"""
Compute the gradient of the loglikelihood function for part a.

Input:
single_obj_loc: 1 * d numpy array.
Location of the single object.

sensor_loc: k * d numpy array.
Location of sensor.

single_distance: k dimensional numpy array.
Observed distance of the object.

Output:
grad: d-dimensional numpy array.

"""
loc_difference = single_obj_loc - sensor_loc # k * d.
phi = np.linalg.norm(loc_difference, axis=1) # k.
weight = (phi - single_distance) / phi # k.

grad = -np.sum(np.expand_dims(weight, 1) * loc_difference, axis=0) / noise**2 # d
return grad


############################################################################
######### Part c ###########################################################
############################################################################
def log_likelihood(obj_loc, sensor_loc, distance, noise=1):
"""
This function computes the log likelihood (as expressed in Part a).
Input:
obj_loc: shape [1,2]
sensor_loc: shape [7,2]
distance: shape [7]
Output:
The log likelihood function value.
```

```
"""
diff_distance = np.sqrt(np.sum((sensor_loc - obj_loc)**2, axis=1)) - distance
func_value = -sum((diff_distance)**2) / (2 * noise**2)
return func_value


#############################################################################
######### Part e, f, g ######################################################
#############################################################################


#############################################################################
######### Gradient Computing and MLE ########################################
#############################################################################
def compute_grad_likelihood_part_e(sensor_loc, obj_loc, distance, noise=1):
"""
Compute the gradient of the loglikelihood function for part d.

Input:
sensor_loc: k * d numpy array.
Location of sensors.

obj_loc: n * d numpy array.
Location of the objects.

distance: n * k dimensional numpy array.
Observed distance of the object.

Output:
grad: k * d numpy array.
"""
grad = np.zeros(sensor_loc.shape)
for i, single_sensor_loc in enumerate(sensor_loc):
single_distance = distance[:, i]
grad[i] = compute_gradient_of_likelihood(single_sensor_loc, obj_loc, single_distance,
noise)

return grad


def find_mle_by_grad_descent_part_e(initial_sensor_loc,
obj_loc,
distance,
noise=1,
lr=0.001,
num_iters=1000):
"""
Compute the gradient of the loglikelihood function for part a.

Input:
initial_sensor_loc: k * d numpy array.
Initialized Location of the sensors.

obj_loc: n * d numpy array. Location of the n objects.

distance: n * k dimensional numpy array.
Observed distance of the n object.

Output:
sensor_loc: k * d numpy array. The mle for the location of the object.

"""
sensor_loc = initial_sensor_loc
```

```
for t in range(num_iters):
sensor_loc += lr * compute_grad_likelihood_part_e(sensor_loc, obj_loc, distance, noise)

return sensor_loc


##########################################################################

######### Estimate distance given estimated sensor locations. #########
##########################################################################


def compute_distance_with_sensor_and_obj_loc(sensor_loc, obj_loc):
"""
stimate distance given estimated sensor locations.

Input:
sensor_loc: k * d numpy array.
Location of the sensors.

obj_loc: n * d numpy array. Location of the n objects.

Output:
distance: n * k dimensional numpy array.
"""
estimated_distance = scipy.spatial.distance.cdist(obj_loc, sensor_loc, metric='euclidean')
return estimated_distance


##########################################################################
######### Data Generating Functions ######################################
##########################################################################


def generate_sensors(num_sensors=7, spatial_dim=2):
"""
Generate sensor locations.
Input:
num_sensors: The number of sensors.
spatial_dim: The spatial dimension.
Output:
sensor_loc: num_sensors * spatial_dim numpy array.
"""
sensor_loc = 100 * np.random.randn(num_sensors, spatial_dim)
return sensor_loc


def generate_dataset(sensor_loc,
num_sensors=7,
spatial_dim=2,
num_data=1,
original_dist=True,
noise=1):
"""
Generate the locations of n points.

Input:
sensor_loc: num_sensors * spatial_dim numpy array. Location of sensor.
num_sensors: The number of sensors.
spatial_dim: The spatial dimension.
num_data: The number of points.
original_dist: Whether the data are generated from the original
distribution.
```

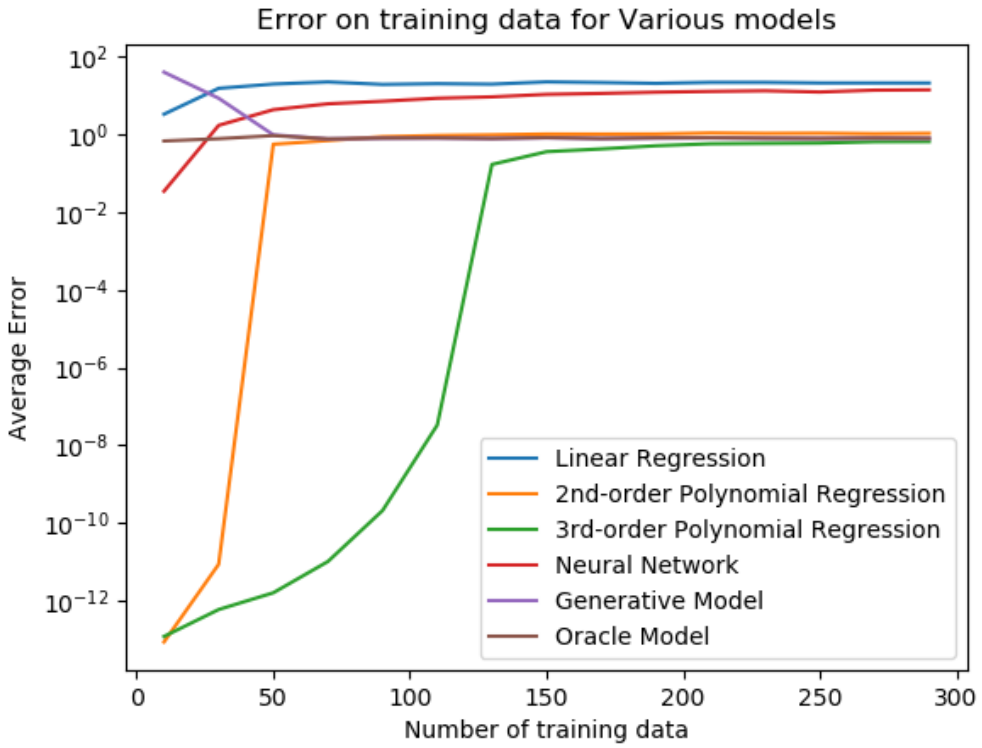
```
Output:
obj_loc: num_data * spatial_dim numpy array. The location of the num_data objects.
distance: num_data * num_sensors numpy array. The distance between object and
the num_sensors sensors.
"""
assert num_sensors, spatial_dim == sensor_loc.shape

obj_loc = 100 * np.random.randn(num_data, spatial_dim)
if not original_dist:
obj_loc += 1000

distance = scipy.spatial.distance.cdist(obj_loc, sensor_loc, metric='euclidean')
distance += np.random.randn(num_data, num_sensors) * noise
return distance, obj_loc
```

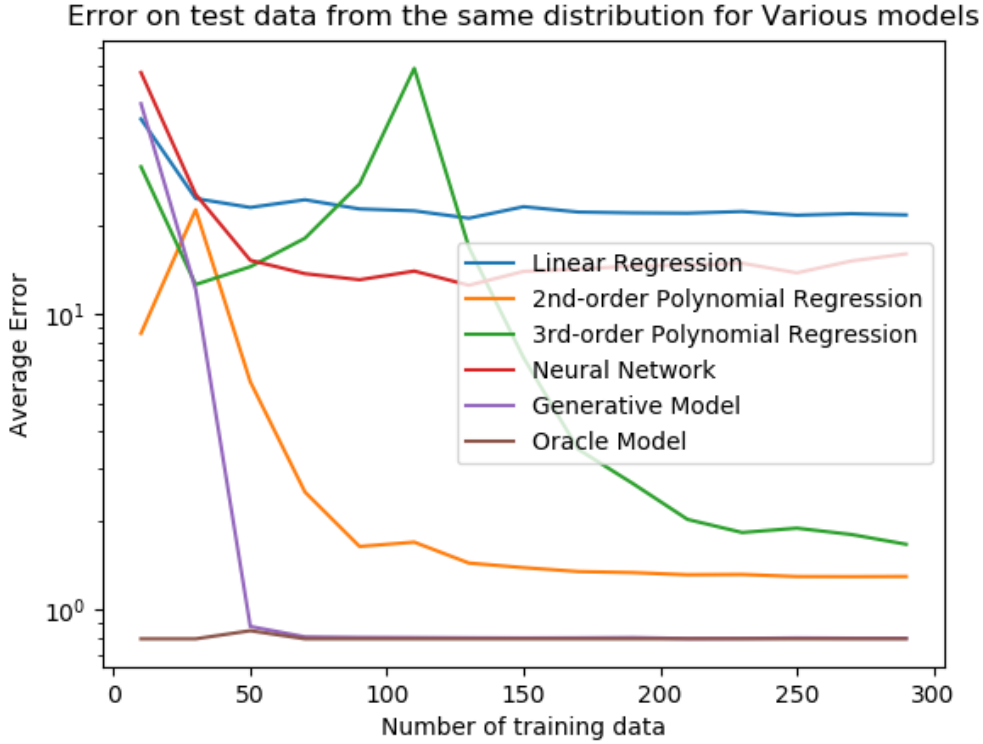
(c) **Use `plot1.py` to train each of the five models on each of the fifteen training sets. Use your results to generate three figures. Each figure should include *all* of the models on the same plot so that you can compare them**:

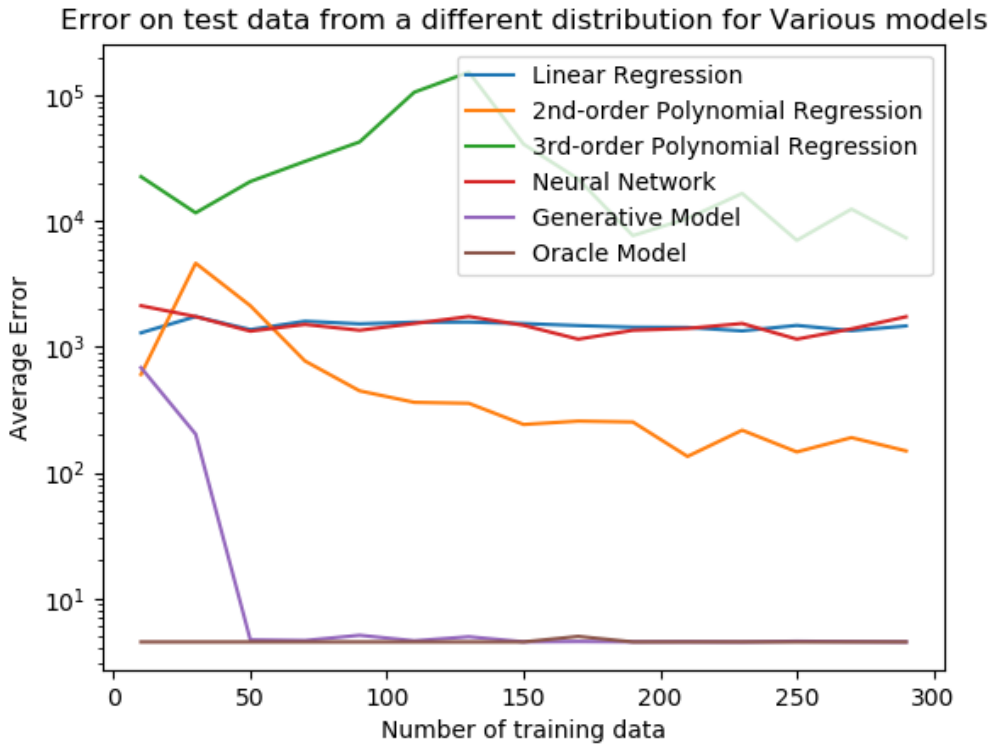
- A plot of *training error* versus n_{train} (the amount of data used to train the model) for all of the models.
- A plot of *testing error* on the "regular" test set versus n_{train} (the amount of data used to train the model) for all of the models.
- A plot of *testing error* on the "shifted" test set versus n_{train} (the amount of data used to train the model) for all of the models.

Briefly describe your observations. What are the strengths and weaknesses of each model?

As we can see below, all models get worse when the number of training data increases. This is because we have more complicated landscapes making it hard to find the global minimum.
The 3-rd order polynomial regression does better at training data probably just because they overfit.
Generative and oracle models perform the best in validation data because we know the true underlying model in these two cases so we can find a more accurate representation of our observations.
Neural network is better than linear in the validation data from the same distribution but not from the different distribution because neural network is hard to generalize to the data it has not been trained on before.
The 2-nd order polynomial regression seems to be the third best option following the generative and oracle models. This is because we find the sweet spot of the bias-variance tradeoff and minimize the overall error.
Linear model is similar to neural network, which performs better than 3rd order polynomial regression because it doesn't overfit.


Error on training data for Various models
Average Error
Number of training data
Linear Regression
2nd-order Polynomial Regression
3rd-order Polynomial Regression
Neural Network
Generative Model
Oracle Model


Error on test data from the same distribution for Various models
Average Error
Number of training data
Linear Regression
2nd-order Polynomial Regression
3rd-order Polynomial Regression
Neural Network
Generative Model
Oracle Model

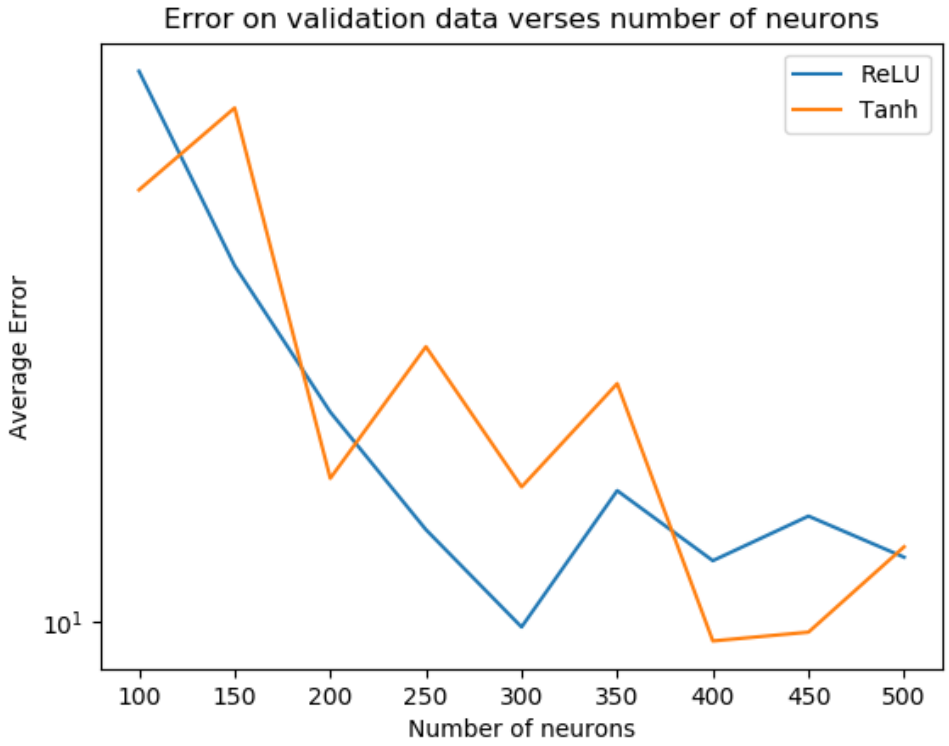



```
import matplotlib.pyplot as plt
import numpy as np

from models import *
from starter import *


def main():
################################################################################
#######################PLOT PART 1##############################################
################################################################################
np.random.seed(0)

ns = np.arange(10, 310, 20)
replicates = 5
#ns = np.arange(10, 100, 20)
#replicates = 2
num_methods = 6
num_sets = 3
mses = np.zeros((len(ns), replicates, num_methods, num_sets))

def generate_data(sensor_loc, k=7, d=2, n=1, original_dist=True, noise=1):
return generate_dataset(
sensor_loc,
num_sensors=k,
spatial_dim=d,
num_data=n,
original_dist=original_dist,
noise=noise)

for s in range(replicates):
sensor_loc = generate_sensors()
X_test, Y_test = generate_data(sensor_loc, n=1000)
X_test2, Y_test2 = generate_data(
```

```
sensor_loc, n=1000, original_dist=False)
for t, n in enumerate(ns):
X, Y = generate_data(sensor_loc, n=n) # X [n * 7] Y [n * 2]
Xs_test, Ys_test = [X, X_test, X_test2], [Y, Y_test, Y_test2]
### Linear regression:
mse = linear_regression(X, Y, Xs_test, Ys_test)
mses[t, s, 0] = mse

### Second-order Polynomial regression:
mse = poly_regression_second(X, Y, Xs_test, Ys_test)
mses[t, s, 1] = mse

### 3rd-order Polynomial regression:
mse = poly_regression_cubic(X, Y, Xs_test, Ys_test)
mses[t, s, 2] = mse

### Neural Network:
mse = neural_network(X, Y, Xs_test, Ys_test)
mses[t, s, 3] = mse

### Generative model:
mse = generative_model(X, Y, Xs_test, Ys_test)
mses[t, s, 4] = mse

### Oracle model:
mse = oracle_model(X, Y, Xs_test, Ys_test, sensor_loc)
mses[t, s, 5] = mse

print('{}th Experiment with {} samples done...'.format(s, n))

### Plot MSE for each model.
plt.figure()
regressors = [
'Linear Regression', '2nd-order Polynomial Regression',
'3rd-order Polynomial Regression', 'Neural Network',
'Generative Model', 'Oracle Model'
]
for a in range(6):
plt.plot(ns, np.mean(mses[:, :, a, 0], axis=1), label=regressors[a])

plt.title('Error on training data for Various models')
plt.xlabel('Number of training data')
plt.ylabel('Average Error')
plt.legend(loc='best')
plt.yscale('log')
plt.savefig('Figure_4c-train_mse.png')
#plt.show()

plt.figure()
for a in range(6):
plt.plot(ns, np.mean(mses[:, :, a, 1], axis=1), label=regressors[a])

plt.title(
'Error on test data from the same distribution for Various models')
plt.xlabel('Number of training data')
plt.ylabel('Average Error')
plt.legend(loc='best')
plt.yscale('log')
plt.savefig('Figure_4c-val_same_mse.png')
# plt.show()

plt.figure()
for a in range(6):
```

```
plt.plot(ns, np.mean(mses[:, :, a, 2], axis=1), label=regressors[a])

plt.title(
'Error on test data from a different distribution for Various models')
plt.xlabel('Number of training data')
plt.ylabel('Average Error')
plt.legend(loc='best')
plt.yscale('log')
plt.savefig('Figure_4c-val_different_mse.png')
# plt.show()


if __name__ == '__main__':
main()
```

(d) We are now going to do some hyper-parameter tuning on our neural network. Fix the number of hidden layers to be two and let l be the number of neurons in each of these two layers. Try values for l between 100 and 500 in increments of 50. Use data sets with $n_{\text{train}} = 200$ and $n_{\text{test}} = 1,000$. **What is the best choice for l (the number of neurons in the hidden layers)? Justify your answer with plots.** The starter code is in `plot2.py`.



The best number of neurons for ReLU is 300 and for Tanh is 400. This is because we have low validation errors for these two points on the plot. Therefore, I would say using 300-400 neurons is the best for our model.

```
import matplotlib.pyplot as plt
import numpy as np

from starter import *


def neural_network(X, Y, X_test, Y_test, num_neurons, activation):
"""
This function performs neural network prediction.
Input:
X: independent variables in training data.
Y: dependent variables in training data.
X_test: independent variables in test data.
Y_test: dependent variables in test data.
num_neurons: number of neurons in each layer
activation: type of activation, ReLU or tanh
Output:
mse: Mean square error on test data.
"""
```

```
## YOUR CODE HERE
##################
meanX = np.mean(X, axis=0)
stdX = np.std(X, axis=0)
meanY = np.mean(Y, axis=0)
stdY = np.std(Y, axis=0)
X = (X - meanX) / stdX
Y = (Y - meanY) / stdY
activations = dict(ReLU=ReLUActivation,
tanh=TanhActivation,
linear=LinearActivation)
lr = dict(ReLU=0.1, tanh=0.1, linear=0.05)
iterations = dict(ReLU=1000, tanh=1000, linear=1000)
names = ['ReLU', 'linear', 'tanh']
#### PART G ####
activation_func = activations[activation]
model = Model(X.shape[1])
model.addLayer(DenseLayer(num_neurons, activation_func()))
model.addLayer(DenseLayer(num_neurons, activation_func()))
model.addLayer(DenseLayer(Y.shape[1], LinearActivation()))
model.initialize(QuadraticCost())

# Train the model and display the results
model.train(X, Y, iterations[activation], GDOptimizer(eta=lr[activation]))
Y_pred = model.predict((X_test - meanX) / stdX) * stdY + meanY
mse = np.mean(np.sqrt(np.sum((Y_pred - Y_test) ** 2, axis=1)))
return mse


################################################################################
#######################PLOT PART 2##############################################
################################################################################
def generate_data(sensor_loc, k=7, d=2, n=1, original_dist=True, noise=1):
return generate_dataset(
sensor_loc,
num_sensors=k,
spatial_dim=d,
num_data=n,
original_dist=original_dist,
noise=noise)


np.random.seed(0)
n = 200
num_neuronss = np.arange(100, 550, 50)
mses = np.zeros((len(num_neuronss), 2))

# for s in range(replicates):

sensor_loc = generate_sensors()
X, Y = generate_data(sensor_loc, n=n) # X [n * 2] Y [n * 7]
X_test, Y_test = generate_data(sensor_loc, n=1000)
for t, num_neurons in enumerate(num_neuronss):
### Neural Network:
mse = neural_network(X, Y, X_test, Y_test, num_neurons, "ReLU")
mses[t, 0] = mse

mse = neural_network(X, Y, X_test, Y_test, num_neurons, "tanh")
mses[t, 1] = mse

print('Experiment with {} neurons done...'.format(num_neurons))

### Plot MSE for each model.
```

```
plt.figure()
activation_names = ['ReLU', 'Tanh']
for a in range(2):
plt.plot(num_neuronss, mses[:, a], label=activation_names[a])

plt.title('Error on validation data verses number of neurons')
plt.xlabel('Number of neurons')
plt.ylabel('Average Error')
plt.legend(loc='best')
plt.yscale('log')
plt.savefig('Figure_4d-num_neurons.png')
plt.close()
```

(e) We are going to do some more hyper-parameter tuning on our neural network. Let k be the number of hidden layers and let l be the number of neurons in each hidden layer. **Write a formula for the total number of weights in our network in terms of l and k. If we want to keep the total number of weights in the network approximately equal to** 10000**, find a formula for l in terms of k.** Try values of k between 1 and 4 with the appropriate implied choice for l. Use data sets with $n_{\text{train}} = n_{\text{test}} = 200$. **What is the best choice for k (the number of layers)? Justify your answer with plots.** The starter code is in `plot3.spy`.

Let's assume we have p inputs and q outputs. According to the solution to the Guerrilla section problem, we have the number of weights with bias terms:

$$\text{Number of weights with bias terms: } (p+1)l + \max(0, k-1)(l+1)l + (l+1)q$$

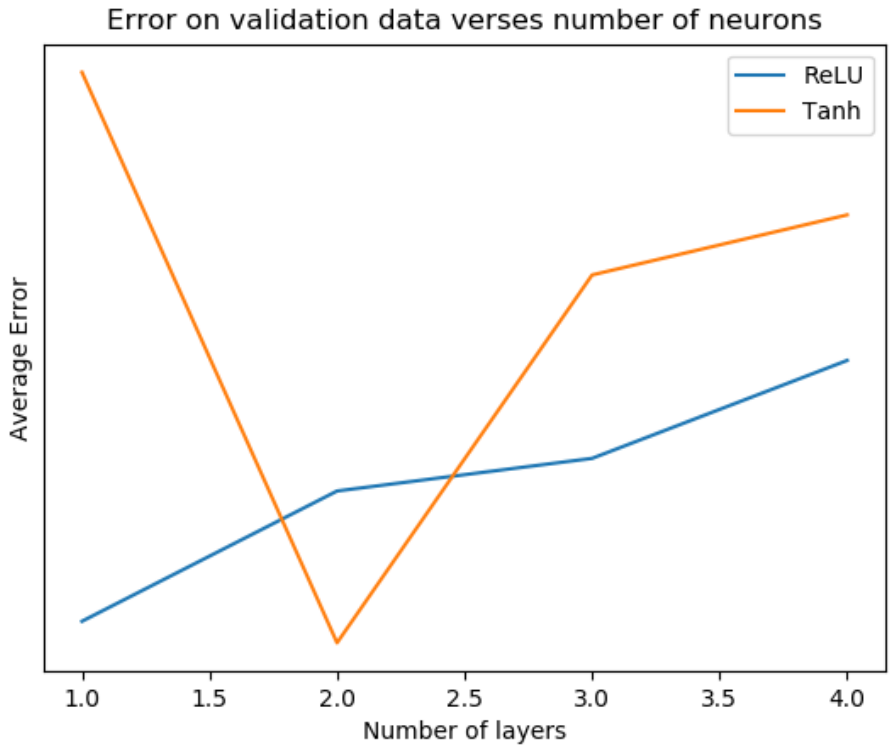
To get the number of neurons we solve the following quadratic equation:

$$(k-1)l^2 + (p+q+k)l + q = 10000$$

Therefore, we have obtained a formula for l in terms of k:

$$l = \frac{-(p+q+k) + \sqrt{(p+q+k)^2 - 4(q-10000)(k-1)}}{2(k-1)}$$

where we have $p = 7 \quad q = 2$ in this question



As we can see on the plot, for ReLU the minimum error is found at 1 layer and for tanh the minimum error is found at 3 layers. The overall minimum error is found at 1 layer using ReLU.

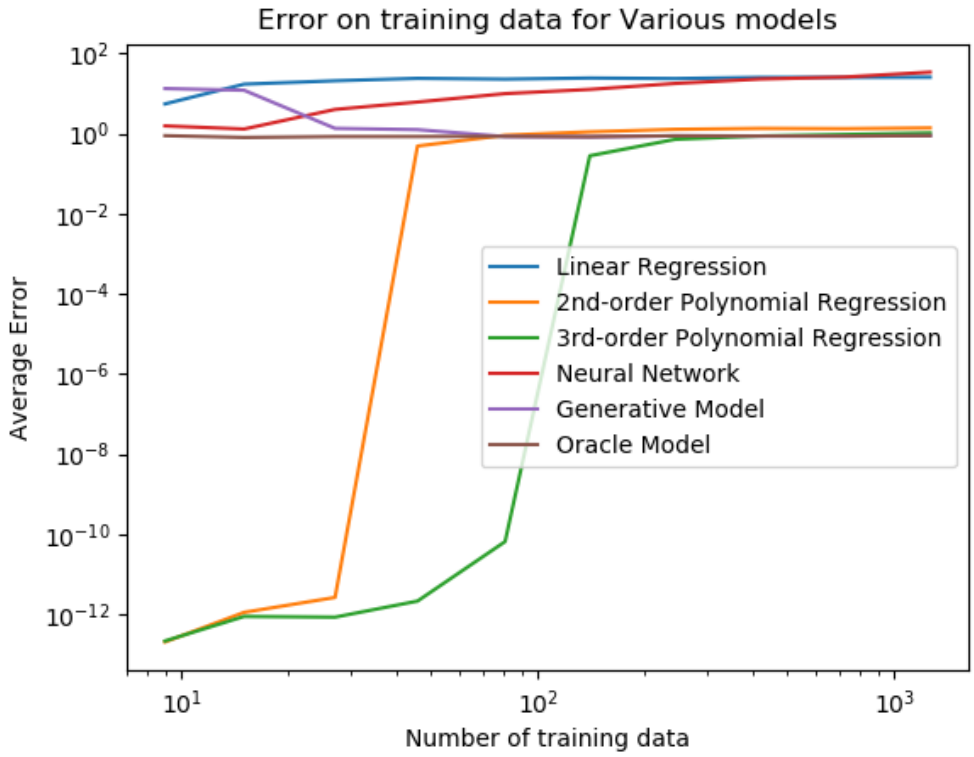
```
import matplotlib.pyplot as plt
```

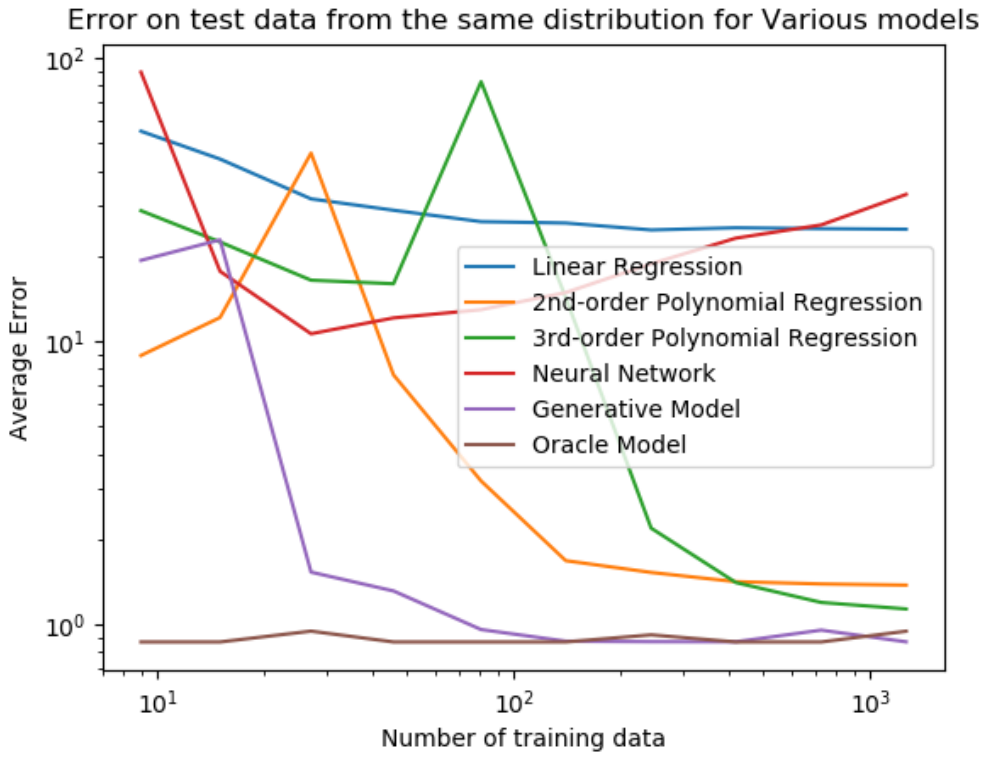
```
import numpy as np

from starter import *


def neural_network(X, Y, X_test, Y_test, num_layers, activation):
"""
This function performs neural network prediction.
Input:
X: independent variables in training data.
Y: dependent variables in training data.
X_test: independent variables in test data.
Y_test: dependent variables in test data.
num_layers: number of layers in neural network
activation: type of activation, ReLU or tanh
Output:
mse: Mean square error on test data.
"""
## YOUR CODE HERE
##################
meanX = np.mean(X, axis=0)
stdX = np.std(X, axis=0)
meanY = np.mean(Y, axis=0)
stdY = np.std(Y, axis=0)
X = (X - meanX) / stdX
Y = (Y - meanY) / stdY
activations = dict(ReLU=ReLUActivation,
tanh=TanhActivation,
linear=LinearActivation)
lr = dict(ReLU=0.1, tanh=0.1, linear=0.005)
iterations = dict(ReLU=2000, tanh=2000, linear=2000)
roots = np.sort(np.roots([(num_layers - 1), (num_layers + Y.shape[1] + X.shape[1] + 1), Y.shape[1]-10000]))
num_neurons = int(roots[-1])
names = ['ReLU', 'linear', 'tanh']
#### PART G ####
activation_func = activations[activation]
model = Model(X.shape[1])
for iLayer in range(num_layers):
model.addLayer(DenseLayer(num_neurons, activation_func()))
model.addLayer(DenseLayer(Y.shape[1], LinearActivation()))
model.initialize(QuadraticCost())

# Train the model and display the results
model.train(X, Y, iterations[activation], GDOptimizer(eta=lr[activation]))
Y_pred = model.predict((X_test - meanX) / stdX) * stdY + meanY
mse = np.mean(np.sqrt(np.sum((Y_pred - Y_test) ** 2, axis=1)))
return mse


##############################################################################
######################PLOT PART 2#############################################
##############################################################################
def generate_data(sensor_loc, k=7, d=2, n=1, original_dist=True, noise=1):
return generate_dataset(
sensor_loc,
num_sensors=k,
spatial_dim=d,
num_data=n,
original_dist=original_dist,
noise=noise)


np.random.seed(0)
```

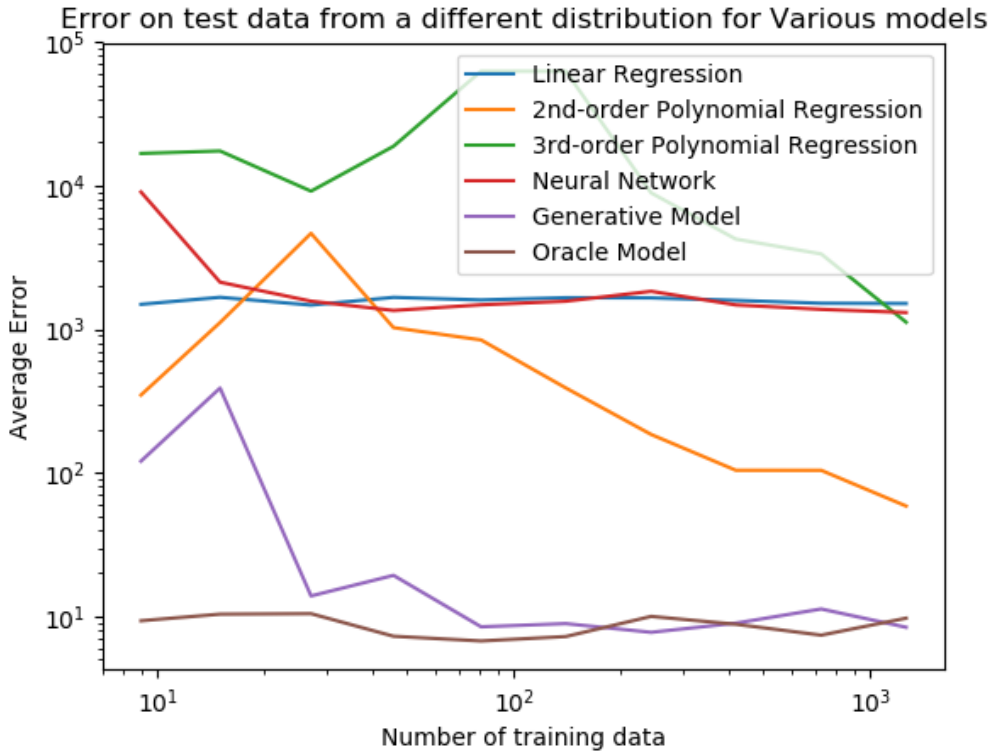
```
n = 200
num_layerss = [1, 2, 3, 4]
mses = np.zeros((len(num_layerss), 2))

# for s in range(replicates):
sensor_loc = generate_sensors()
X, Y = generate_data(sensor_loc, n=n) # X [n * 2] Y [n * 7]
X_test, Y_test = generate_data(sensor_loc, n=1000)
for t, num_layers in enumerate(num_layerss):
### Neural Network:
mse = neural_network(X, Y, X_test, Y_test, num_layers, "ReLU")
mses[t, 0] = mse

mse = neural_network(X, Y, X_test, Y_test, num_layers, "tanh")
mses[t, 1] = mse

print('Experiment with {} layers done...'.format(num_layers))

### Plot MSE for each model.
plt.figure()
activation_names = ['ReLU', 'Tanh']
for a in range(2):
plt.plot(num_layerss, mses[:, a], label=activation_names[a])

plt.title('Error on validation data verses number of neurons')
plt.xlabel('Number of layers')
plt.ylabel('Average Error')
plt.legend(loc='best')
plt.yscale('log')
plt.savefig('Figure_4e-num_layers.png')
plt.close()
```

(f) You might have seen the neural network performance is disappointing compared to the generative model in the "shifted" data. Try increasing the number of training data and tune the hyper-parameters. Can you get it to generalize to the "shifted" test data? **Attach the "number of training data vs accuracy" plot to justify your conclusion.** What is the intuition how neural network works on predicting the D? The starter kit is provided in `plot4.py`.

I changed the number of training data to be logarithmically scaled within $[3^1, 3^7]$. I also used the best hyperparameter from the previous part which is 1 hidden layer and 908 neurons with ReLU.







As we can see from the plot, after tuning the hyperparameter the performance of neural network doesn't really get any better. This is because "shifted" data come from a different distribution and our neural network doesn't have any clue about it. This is the same problem for linear regression with augmented features because they all tend to overfit within the range of training data. Generative models, however, is trying to use the training data to extract a underlying probability distribution,

where we know the number of sensors and we know what the actual models is in our case. If we don't have these information, generative models might not perform as well.

```
import numpy as np
import matplotlib.pyplot as plt

from starter import *
from plot1 import *


def neural_network(X, Y, Xs_test, Ys_test,
lr=0.1, iterations=2000, num_neurons=1000, num_layers=1, activation='ReLU'):
"""
This function performs neural network prediction.
Input:
X: independent variables in training data.
Y: dependent variables in training data.
Xs_test: independent variables in test data.
Ys_test: dependent variables in test data.
Output:
mse: Mean square error on test data.
"""
## YOUR CODE HERE
##################
# Copied from backprop_sol
meanX = np.mean(X, axis=0)
stdX = np.std(X, axis=0)
meanY = np.mean(Y, axis=0)
stdY = np.std(Y, axis=0)
X = (X - meanX) / stdX
Y = (Y - meanY) / stdY
activations = dict(ReLU=ReLUActivation,
tanh=TanhActivation,
linear=LinearActivation)

#### PART G ####
activation_func = activations[activation]
model = Model(X.shape[1])
for iLayer in range(num_layers):
model.addLayer(DenseLayer(num_neurons, activation_func()))
model.addLayer(DenseLayer(Y.shape[1], LinearActivation()))
model.initialize(QuadraticCost())

# Train the model and display the results
# model.trainBatch(X, Y, int(iterations / 10), iterations, GDOptimizer(eta=lr))
model.train(X, Y, iterations, GDOptimizer(eta=lr))
mses = []
for i, X_test in enumerate(Xs_test):
Y_test = Ys_test[i]
Y_pred = model.predict((X_test - meanX) / stdX) * stdY + meanY
mse = np.mean(np.sqrt(np.sum((Y_pred - Y_test) ** 2, axis=1)))
mses.append(mse)
return mses


def main():
##############################################################################
######################PLOT PART 1#############################################
##############################################################################
np.random.seed(0)

ns = np.arange(2, 7, 0.5)
ns = np.array(np.power(3, ns), dtype=int)
```

```
replicates = 5
num_methods = 6
num_sets = 3
mses = np.zeros((len(ns), replicates, num_methods, num_sets))

def generate_data(sensor_loc, k=7, d=2, n=1, original_dist=True, noise=1):
return generate_dataset(
sensor_loc,
num_sensors=k,
spatial_dim=d,
num_data=n,
original_dist=original_dist,
noise=noise)

for s in range(replicates):
sensor_loc = generate_sensors()
X_test, Y_test = generate_data(sensor_loc, n=1000)
X_test2, Y_test2 = generate_data(
sensor_loc, n=1000, original_dist=False)
for t, n in enumerate(ns):
X, Y = generate_data(sensor_loc, n=n) # X [n * 7] Y [n * 2]
Xs_test, Ys_test = [X, X_test, X_test2], [Y, Y_test, Y_test2]
### Linear regression:
mse = linear_regression(X, Y, Xs_test, Ys_test)
mses[t, s, 0] = mse

### Second-order Polynomial regression:
mse = poly_regression_second(X, Y, Xs_test, Ys_test)
mses[t, s, 1] = mse

### 3rd-order Polynomial regression:
mse = poly_regression_cubic(X, Y, Xs_test, Ys_test)
mses[t, s, 2] = mse

### Neural Network:
mse = neural_network(X, Y, Xs_test, Ys_test)
mses[t, s, 3] = mse

### Generative model:
mse = generative_model(X, Y, Xs_test, Ys_test)
mses[t, s, 4] = mse

### Oracle model:
mse = oracle_model(X, Y, Xs_test, Ys_test, sensor_loc)
mses[t, s, 5] = mse

print('{}th Experiment with {} samples done...'.format(s, n))

### Plot MSE for each model.
plt.figure()
regressors = [
'Linear Regression', '2nd-order Polynomial Regression',
'3rd-order Polynomial Regression', 'Neural Network',
'Generative Model', 'Oracle Model'
]
for a in range(6):
plt.plot(ns, np.mean(mses[:, :, a, 0], axis=1), label=regressors[a])

plt.title('Error on training data for Various models')
plt.xlabel('Number of training data')
plt.ylabel('Average Error')
plt.legend(loc='best')
plt.xscale('log')
```

```
plt.yscale('log')
plt.savefig('Figure_4f-train_mse.png')
#plt.show()
plt.close()

plt.figure()
for a in range(6):
plt.plot(ns, np.mean(mses[:, :, a, 1], axis=1), label=regressors[a])

plt.title(
'Error on test data from the same distribution for Various models')
plt.xlabel('Number of training data')
plt.ylabel('Average Error')
plt.legend(loc='best')
plt.xscale('log')
plt.yscale('log')
plt.savefig('Figure_4f-val_same_mse.png')
#plt.show()
plt.close()

plt.figure()
for a in range(6):
plt.plot(ns, np.mean(mses[:, :, a, 2], axis=1), label=regressors[a])

plt.title(
'Error on test data from a different distribution for Various models')
plt.xlabel('Number of training data')
plt.ylabel('Average Error')
plt.legend(loc='best')
plt.xscale('log')
plt.yscale('log')
plt.savefig('Figure_4f-val_different_mse.png')
#plt.show()
plt.close()


if __name__ == '__main__':
main()
```

Question 5. Entropy, KL Divergences, and Cross-Entropy

Stepping back for a bit, notice that so far we have mostly considered so called *Euclidean* spaces, the most prominent example is the vector space $\mathbb{R}^d$ with inner product $\langle \vec{x}, \vec{y} \rangle = \vec{x}^\top \vec{y}$ and norm $\|\vec{x}\|_2^2 = \vec{x}^\top \vec{x}$. For example in linear regression, we had the loss function

$$f(\vec{\theta}) = \|\mathbf{X}^\top \vec{\theta} - \vec{y}\|_2^2 = \sum_{i=1}^{n} (\vec{x}_i^\top \vec{\theta} - y_i)^2$$

which uses precisely the Euclidean norm squared to measure the error.

In this problem we will implicitly consider a different geometric structure, which defines a metric on probability distributions. In more advanced courses, this can be developed further to show how probability distributions are naturally imbued with a curved non-Euclidean intrinsic geometry. Here, our goals are more modest — we just want you to better understand the relationship between probability distributions, entropy, KL Divergence, and cross-entropy.

Let $\vec{p}$ and $\vec{q}$ be two probability distributions, i.e. $p_i \geq 0$, $q_i \geq 0$, $\sum_i p_i = 1$ and $\sum_i q_i = 1$, then we define the Kullback-Leibler divergence

$$\mathrm{KL}(\vec{p}, \vec{q}) = \sum_i p_i \log \frac{p_i}{q_i}$$

which is the "distance" to $\vec{p}$ from $\vec{q}$. We have $\mathrm{KL}(\vec{p}, \vec{p}) = 0$ and $\mathrm{KL}(\vec{p}, \vec{q}) \geq 0$ (the latter by Jensen's inequality) as would be expected from a distance metric. However, $\mathrm{KL}(\vec{p}, \vec{q}) \neq \mathrm{KL}(\vec{q}, \vec{p})$ since the KL divergence is not symmetric.

(a) *Entropy motivation:* Let $X_1, X_2, \ldots, X_n$ be independent identically distributed random variables taking values in a finite set $\{0, 1, \ldots, m\}$, i.e. $p_j = \mathbf{P}(X_i = j)$ for $j \in \{0, 1, \ldots, m\}$. The *empirical number of occurrences* is then a random vector that we can denote $\vec{F}^{(n)}$ where $F_j^{(n)}$ is the number of variables X_i that happen to take a value equal to j.

Intuitively, we can consider coin tosses with $j = 0$ corresponding to heads and $j = 1$ corresponding to tails. Say we do an experiment with $n = 100$ coin tosses, then $F_0^{(100)}$ is the number of heads that came up and $F_1^{(100)}$ is the number of tails.

Recall that the number of configurations of $X_1, X_2, \ldots, X_n$ that have $f^{(n)}$ as their empirical type is $\binom{n}{f_0^{(n)}, f_1^{(n)}, \ldots, f_m^{(n)}}$. Further notice that dividing the empirical type by n yeilds an empirical probability distribution.

Show using the crudest form of Stirling's approximation ($l! \approx (\frac{l}{e})^l$) that this is approximately equal to $\exp(nH(f^{(n)}/n))$ where the entropy H of a probability distribution is defined as $H(\vec{p}) = \sum_{j=0}^{m} p_j \ln \frac{1}{p_j}$.

$$
\begin{aligned}
&\binom{n}{f_0^{(n)}, f_1^{(n)}, \ldots, f_m^{(n)}} \\
=&\frac{n!}{f_0^{(n)}! \ldots f_m^{(n)}!} \\
=&\frac{\frac{n^n}{e^n}}{\prod_{j=0}^{m} \left(\frac{f_j^{(n)}}{e}\right)^{f_j^{(n)}}} \\
=&\frac{n^n e^{\sum_{j=0}^{m}(f_j^{(n)}/n)}}{\prod_{j=0}^{m} \left(f_j^{(n)}\right)^{f_j^{(n)}}} \\
\approx&\frac{n^n}{\prod_{j=0}^{m} \left(f_j^{(n)}\right)^{f_j^{(n)}}} \\
=&\frac{n^{\sum_{j=0}^{m} f_j^{(n)}}}{\prod_{j=0}^{m} \left(f_j^{(n)}\right)^{f_j^{(n)}}}
\end{aligned}
$$

$$
\begin{aligned}
&\exp(nH(f^{(n)}/n)) \\
=&\exp(n \sum_{j=0}^{m} \frac{f_j^{(n)}}{n} \ln \frac{n}{f_j^{(n)}}) \\
=&\exp(\sum_{j=0}^{m} f_j^{(n)} \ln \frac{n}{f_j^{(n)}}) \\
=&\prod_{j=0}^{m} \left(\frac{n}{f_j^{(n)}}\right)^{f_j^{(n)}} \\
=&\frac{n^{\sum_{j=0}^{m} f_j^{(n)}}}{\prod_{j=0}^{m} (f_j^{(n)})^{f_j^{(n)}}}
\end{aligned}
$$

Note that we have the following equality because the probability sum up to 1:

$$\sum_{j=0}^{m} (f_j^{(n)}/n) = 1$$

The last two lines of the formula above are the same.

(b) *KL divergence motivation:* Recall that the probability of seeing a particular empirical type is given by:

$$\mathbf{P}(\vec{F}^{(n)} = \vec{f}^{(n)}) = \binom{n}{f_0^{(n)}, f_1^{(n)}, \ldots, f_m^{(n)}} \prod_{j=1}^{m} p_j^{f_j^{(n)}}.$$

Consider the limit of large n and a sequence of empirical types so that $\frac{1}{n}\vec{f}^{(n)} \to \vec{f}$ for $n \to \infty$, where $\vec{f}$ is some distribution of interest.

Use Stirling's approximation to show that

$$\lim_{n\to\infty} \frac{1}{n} \log \mathbf{P}(\vec{F}^{(n)} = \vec{f}^{(n)}) = -\operatorname{KL}(\vec{f}, \vec{p})$$

Intuitively this means that the larger $\operatorname{KL}(\vec{f}, \vec{p})$ is, the easier it is to conclude $\vec{f} \neq \vec{p}$ from empirical data since the chance that we would get confused in that way is decaying exponentially. Note also that the empirical distribution is the first argument of the KL divergence and the true model is the second argument of the KL divergence — we are going from the true model to the empirical one.

We can use the conclusion in part (a):

$$\begin{aligned}
&\lim_{n\to\infty} \frac{1}{n} \log \mathbf{P}(\vec{F}^{(n)} = \vec{f}^{(n)}) \\
&= \lim_{n\to\infty} \frac{1}{n} \log \left(\exp(nH(\frac{f^{(n)}}{n})) \prod_{j=0}^{m} p_j^{f_j^{(n)}} \right) \\
&= \lim_{n\to\infty} \frac{1}{n} \log \left(\exp(n \sum_{j=0}^{m} \frac{f_j^{(n)}}{n} \ln \frac{n}{f_j^{(n)}}) \prod_{j=0}^{m} p_j^{f_j^{(n)}} \right) \\
&= \lim_{n\to\infty} \frac{1}{n} \left(n \sum_{j=0}^{m} \frac{f_j^{(n)}}{n} \ln \frac{n}{f_j^{(n)}} \right) + \log \left(\prod_{j=0}^{m} p_j^{f_j^{(n)}} \right) \\
&= \lim_{n\to\infty} \frac{1}{n} \left(n \sum_{j=0}^{m} \frac{f_j^{(n)}}{n} \ln \frac{n}{f_j^{(n)}} \right) + \frac{1}{n} \left(\sum_{j=0}^{m} f_j^{(n)} \log p_j \right) \\
&= \lim_{n\to\infty} \left(\sum_{j=0}^{m} \frac{f_j^{(n)}}{n} \ln \frac{n}{f_j^{(n)}} \right) + \left(\sum_{j=0}^{m} \frac{f_j^{(n)}}{n} \log p_j \right) \\
&= \left(\sum_{j=0}^{m} f_j \ln \frac{1}{f_j} \right) + \left(\sum_{j=0}^{m} f_j \log p_j \right) \\
&= \sum_{j=0}^{m} f_j \ln \frac{p_j}{f_j} \\
&= -\sum_{j=0}^{m} f_j \ln \frac{f_j}{p_j} \\
&= -\operatorname{KL}(\vec{f}, \vec{p})
\end{aligned}$$

(c) **Show that for probability distributions $p(\vec{x}, y)$ and $q_\theta(\vec{x}, y) = q_\theta(y \mid \vec{x})q(\vec{x})$ with $\vec{x}$ from some discrete set $\mathcal{X}$ and y from some discrete set $\mathcal{Y}$ we have**

$$\mathrm{KL}(p, q_\theta) = c - \sum_{\vec{x}\in\mathcal{X}} \sum_{y\in\mathcal{Y}} p(\vec{x}, y) \log q_\theta(y \mid \vec{x}) \tag{2}$$

for some constant c independent of θ.

$$\begin{aligned}
&\mathrm{KL}(p, q_\theta) \\
&= \sum_{\vec{x}\in\mathcal{X}} \sum_{y\in\mathcal{Y}} \left(p(\vec{x}, y) \log \frac{p(\vec{x}, y)}{q_\theta(\vec{x}, y)} \right) \\
&= \sum_{\vec{x}\in\mathcal{X}} \sum_{y\in\mathcal{Y}} \left(p(\vec{x}, y) \log \frac{p(\vec{x}, y)}{q_\theta(y \mid \vec{x})q(\vec{x})} \right) \\
&= \sum_{\vec{x}\in\mathcal{X}} \sum_{y\in\mathcal{Y}} \left(p(\vec{x}, y) \log \frac{p(\vec{x}, y)}{q(\vec{x})} \right) - \sum_{\vec{x}\in\mathcal{X}} \sum_{y\in\mathcal{Y}} (p(\vec{x}, y) \log q_\theta(y \mid \vec{x})) \\
&= c - \sum_{\vec{x}\in\mathcal{X}} \sum_{y\in\mathcal{Y}} (p(\vec{x}, y) \log q_\theta(y \mid \vec{x}))
\end{aligned}$$

The last step is true because we can see that the first term doesn't have θ in it:

$$c = \sum_{\vec{x}\in\mathcal{X}} \sum_{y\in\mathcal{Y}} \left(p(\vec{x}, y) \log \frac{p(\vec{x}, y)}{q(\vec{x})} \right)$$

(d) In logistic regression we predict labels $y_i = +1$ or $y_i = -1$ from features $\vec{x}_i$ using the transition probability model

$$q_\theta(y_i \mid \vec{x}_i) = \frac{1}{1 + e^{-y_i \vec{\theta}^\top \vec{x}_i}}. \tag{3}$$

We now show that the cross-entropy loss you have seen in lectures can be formulated as minimizing the KL distance to the empirical probabilities from the probabilities induced by the model q_θ.

For convenience, we assume that all the feature $\vec{x}_i$ are distinct — no two training points are identical.

Use (c) to show that with the empirical distribution

$$p(\vec{x}, y) = \begin{cases} \frac{1}{n} & \text{if } \vec{x} = \vec{x}_i \text{ and } y = y_i \text{ for some } i = 1, 2, \ldots, n \\ 0 & \text{otherwise} \end{cases}$$

we get

$$\min_{\vec{\theta}} \mathrm{KL}(p, q_{\vec{\theta}}) = \min_{\vec{\theta}} -\frac{1}{n} \sum_i \log q_{\vec{\theta}}(y_i \mid \vec{x}_i),$$

which is the cross entropy loss derived in lectures.

$$\begin{aligned}
&\min_{\vec{\theta}} \mathrm{KL}(p, q_{\vec{\theta}}) \\
&= \min_{\vec{\theta}} \left(\sum_{\vec{x} \in \mathcal{X}} \sum_{y \in \mathcal{Y}} \left(p(\vec{x}, y) \log \frac{p(\vec{x}, y)}{q_\theta(\vec{x}, y)} \right) \right) \\
&= \min_{\vec{\theta}} \left(c - \sum_{\vec{x} \in \mathcal{X}} \sum_{y \in \mathcal{Y}} (p(\vec{x}, y) \log q_\theta(y \mid \vec{x})) \right) \\
&= \min_{\vec{\theta}} - \left(\sum_{\vec{x} \in \mathcal{X}} \sum_{y \in \mathcal{Y}} (p(\vec{x}, y) \log q_\theta(y \mid \vec{x})) \right) \\
&= \min_{\vec{\theta}} - \left(\frac{1}{n} \sum_i (\log q_\theta(y_i \mid \vec{x}_i)) + 0 \sum_{i \neq j} (\log q_\theta(y_i \mid \vec{x}_j)) \right) \\
&= \min_{\vec{\theta}} -\frac{1}{n} \sum_i \log q_{\vec{\theta}}(y_i \mid \vec{x}_i)
\end{aligned}$$

Question 6. Your Own Question

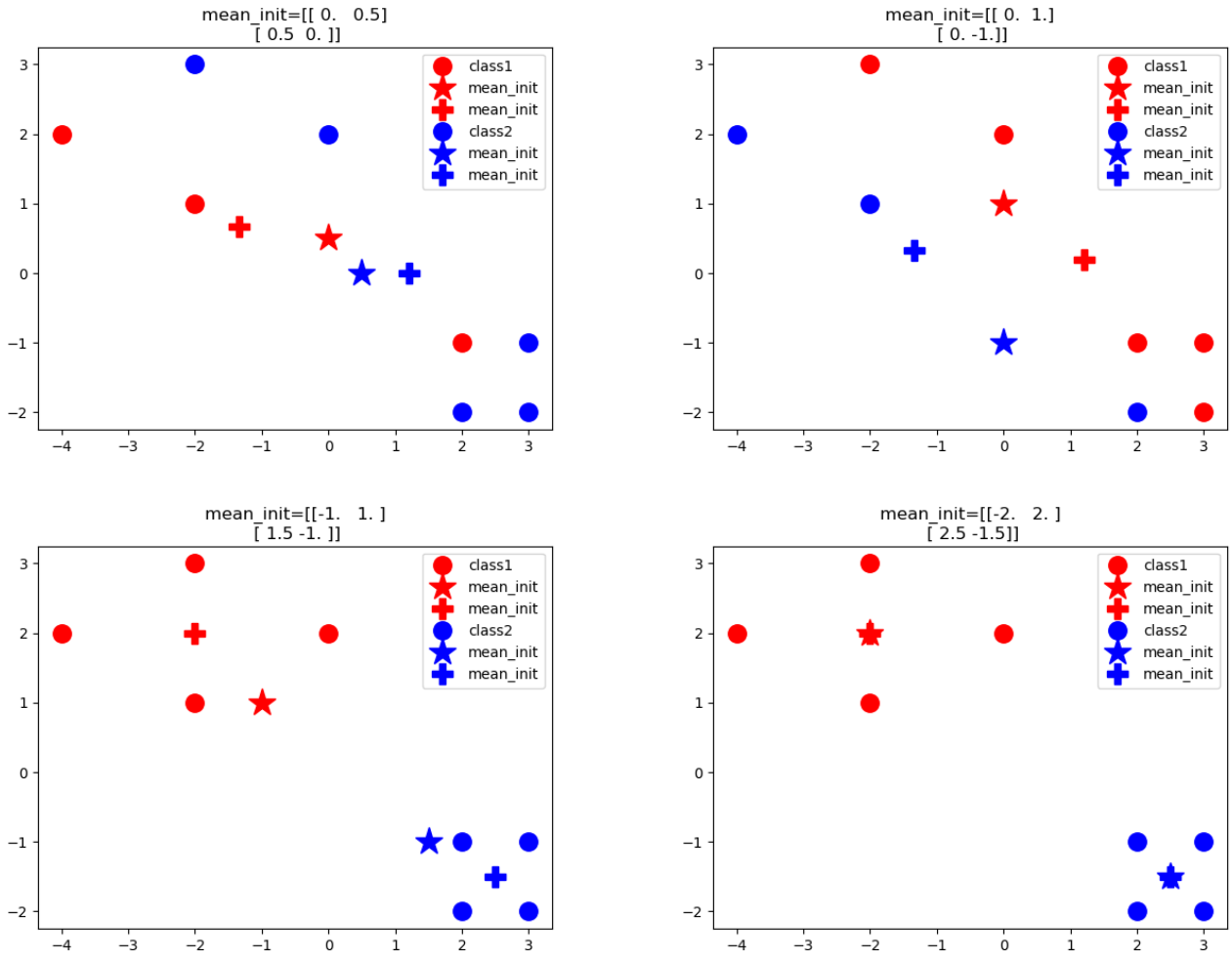
Write your own question, and provide a thorough solution.

Writing your own problems is a very important way to really learn the material. The famous "Bloom's Taxonomy" that lists the levels of learning is: Remember, Understand, Apply, Analyze, Evaluate, and Create. Using what you know to create is the top-level. We rarely ask you any HW questions about the lowest level of straight-up remembering, expecting you to be able to do that yourself. (e.g. make yourself flashcards) But we don't want the same to be true about the highest level.

As a practical matter, having some practice at trying to create problems helps you study for exams much better than simply counting on solving existing practice problems. This is because thinking about how to create an interesting problem forces you to really look at the material from the perspective of those who are going to create the exams.

Besides, this is fun. If you want to make a boring problem, go ahead. That is your prerogative. But it is more fun to really engage with the material, discover something interesting, and then come up with a problem that walks others down a journey that lets them share your discovery. You don't have to achieve this every week. But unless you try every week, it probably won't happen ever.

What happened if we have very few data points and run Expectation-Maximization? I implemented EM using sklearn and select the following eight data points:



As we can see that EM is pretty sensitive to the initial condition. If our initial guesses are close to what our eyes tell us, we can still get a the optimal result. However, because we have very few points, our prior could be extremely biased that we never find the best approximation. Weird things

could also happen if our initial means of two classes are close to each other.
All in all, just like all other ML algorithm, without enough data, EM will not perform very well either.

```
from sklearn.mixture.gaussian_mixture import GaussianMixture
import numpy as np
import matplotlib.pyplot as plt

X_train = np.array([[-4, 2], [-2, 1], [-2, 3], [0, 2], [2, -1], [3, -1], [2, -2], [3, -2]])

estimator = GaussianMixture(n_components=2, max_iter=1000, random_state=0, init_params='random')


plt.figure()
plt.plot(X_train[:, 0], X_train[:, 1], 'k.', markersize=25)
plt.savefig('Figure_6-datapoints.png')
plt.close()

colors = ['r', 'b']


# initial means
estimator.means_init = np.array([[0, 0.5], [0.5, 0]])
# Train the other parameters using the EM algorithm.
estimator.fit(X_train)
classes = estimator.predict(X_train)

plt.figure()
for i in range(2):
plt.plot(X_train[classes == i, 0], X_train[classes == i, 1], colors[i]+'.', markersize=25,
    label='class'+str(i+1))
plt.plot(estimator.means_init[i, 0], estimator.means_init[i, 1], colors[i]+'*', markersize=20,
    label='mean_init')
plt.plot(np.mean(X_train[classes == i, 0]), np.mean(X_train[classes == i, 1]), colors[i] + 'P',
    markersize=15, label='mean_init')
plt.title('mean_init='+str(estimator.means_init))
plt.legend()
plt.savefig('Figure_6-EM1.png')
plt.close()

# initial means
estimator.means_init = np.array([[0, 1], [0, -1]])
# Train the other parameters using the EM algorithm.
estimator.fit(X_train)
classes = estimator.predict(X_train)

plt.figure()
for i in range(2):
plt.plot(X_train[classes == i, 0], X_train[classes == i, 1], colors[i]+'.', markersize=25,
    label='class'+str(i+1))
plt.plot(estimator.means_init[i, 0], estimator.means_init[i, 1], colors[i]+'*', markersize=20,
    label='mean_init')
plt.plot(np.mean(X_train[classes == i, 0]), np.mean(X_train[classes == i, 1]), colors[i] + 'P',
    markersize=15, label='mean_init')
plt.title('mean_init=' + str(estimator.means_init))
plt.legend()
plt.savefig('Figure_6-EM2.png')
plt.close()

# initial means
estimator.means_init = np.array([[-1, 1], [1.5, -1]])
# Train the other parameters using the EM algorithm.
estimator.fit(X_train)
```

```
classes = estimator.predict(X_train)

plt.figure()
for i in range(2):
plt.plot(X_train[classes == i, 0], X_train[classes == i, 1], colors[i]+'.', markersize=25,
    label='class'+str(i+1))
plt.plot(estimator.means_init[i, 0], estimator.means_init[i, 1], colors[i]+'*', markersize=20,
    label='mean_init')
plt.plot(np.mean(X_train[classes == i, 0]), np.mean(X_train[classes == i, 1]), colors[i] + 'P',
    markersize=15, label='mean_init')
plt.title('mean_init=' + str(estimator.means_init))
plt.legend()
plt.savefig('Figure_6-EM3.png')
plt.close()

# initial means
estimator.means_init = np.array([[-2, 2], [2.5, -1.5]])
# Train the other parameters using the EM algorithm.
estimator.fit(X_train)
classes = estimator.predict(X_train)

plt.figure()
for i in range(2):
plt.plot(X_train[classes == i, 0], X_train[classes == i, 1], colors[i]+'.', markersize=25,
    label='class'+str(i+1))
plt.plot(estimator.means_init[i, 0], estimator.means_init[i, 1], colors[i]+'*', markersize=20,
    label='mean_init')
plt.plot(np.mean(X_train[classes == i, 0]), np.mean(X_train[classes == i, 1]), colors[i] + 'P',
    markersize=15, label='mean_init')
plt.title('mean_init='+str(estimator.means_init))
plt.legend()
plt.savefig('Figure_6-EM4.png')
plt.close()
```