CS 189 Introduction to Machine Learning
Spring 2018 HW6

Your self-grade URL is http://eecs189.org/self_grade?question_ids=1_1,1_2,2_1,2_2,3_1,3_2,3_3,4_1,4_2,4_3,4_4,4_5,4_6,5_1,5_2,6.

This homework is due **Monday, July 30 at 11:59pm.**

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

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

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 $f(\mathbf{x})$, we look at the risk. The risk of classifying a new data point $\mathbf{x}$ as class $f(\mathbf{x}) \in \{1, 2, \ldots, c+1\}$ is

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

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

- (**R1**) Find class i such that $P(Y = i|\mathbf{x}) \geq P(Y = j|\mathbf{x})$ for all j, meaning you pick the class with the highest probability given x.
- (**R2**) Choose class i if $P(Y = i|\mathbf{x}) \geq 1 - \frac{\lambda_d}{\lambda_c}$
- (**R3**) Choose doubt otherwise.

Solution:

- Let's first simplify the risk given our specific loss function. If $f(\mathbf{x}) = i$ where i is not doubt, then the risk is

$$R(f(\mathbf{x}) = i|\mathbf{x}) = \sum_{j=1}^{c} L(f(\mathbf{x}) = i, y = j)P(Y = j|\mathbf{x}) \tag{2}$$

$$= 0 \cdot P(Y = i|\mathbf{x}) + \lambda_c \sum_{j=1, j\neq i} P(Y = j|\mathbf{x}) \tag{3}$$

$$= \lambda_c \left(1 - P(Y = i|\mathbf{x})\right) \tag{4}$$

When $f(\mathbf{x}) = c + 1$, meaning you've chosen doubt, the risk is:

$$R(f(\mathbf{x}) = c + 1|\mathbf{x}) = \sum_{j=1}^{c} L(f(\mathbf{x}) = c + 1, y = j)P(Y = j|\mathbf{x}) \tag{5}$$

$$= \lambda_d \sum_{j=1} P(Y = j|\mathbf{x}) \tag{6}$$

$$= \lambda_d \tag{7}$$

because $\sum_{j=1} P(Y = j|\mathbf{x})$ should sum to 1 since its a proper probability distribution. Now let $f_{opt} : \mathbb{R}^d \to \{1, \ldots, c + 1\}$ be the decision rule which implements (**R1**)–(**R3**). We want to show that in expectation the rule f_{opt} is at least as good as an arbitrary rule f. Let $\mathbf{x} \in \mathbb{R}^d$ be a data point, which we want to classify. Let's examine all the possible scenarios where $f_{opt}(\mathbf{x})$ and another arbitrary rule $f(\mathbf{x})$ might differ:
Case 1: Let $f_{opt}(\mathbf{x}) = i$ where $i \neq c + 1$.

- Case 1a: $f(\mathbf{x}) = k$ where $k \neq i$. Then we get with (**R1**) that

$$R(f_{opt}(\mathbf{x}) = i|\mathbf{x}) = \lambda_c \left(1 - P(Y = i|\mathbf{x})\right)$$
$$\leq \lambda_c \left(1 - P(Y = k|\mathbf{x})\right) = R(f(\mathbf{x}) = k|\mathbf{x}).$$

- Case 1b: $f(\mathbf{x}) = c + 1$. Then we get with (**R1**) that

$$R(f_{opt}(\mathbf{x}) = i|\mathbf{x}) = \lambda_c \left(1 - P(Y = i|\mathbf{x})\right)$$
$$\leq \lambda_c(1 - (1 - \frac{\lambda_d}{\lambda_c})) = \lambda_d = R(f(\mathbf{x}) = c + 1|\mathbf{x}).$$

Case 2: Let $f_{opt}(\mathbf{x}) = c + 1$ and $f(\mathbf{x}) = k$ where $k \neq c + 1$. Then:

$$R(f(\mathbf{x}) = k|\mathbf{x}) = \lambda_c(1 - P(Y = k|\mathbf{x})$$

$$R(f_{opt}(\mathbf{x}) = c + 1|\mathbf{x}) = \lambda_d$$

We are in case (**R3**) which means that:

$$\max_{j\in\{1,\dots,c\}} P(Y=j|\mathbf{x}) < 1-\lambda_d/\lambda_c$$

hence $P(Y=k|\mathbf{x}) < 1-\lambda_d/\lambda_c$, which means

$$R(f(\mathbf{x})=k|\mathbf{x}) > \lambda_d = R(f_{opt}(\mathbf{x}))$$

Therefore in every case we proved that the rule f_{opt} is at least as good as the arbitrary rule f, which proves that f_{opt} is an optimal rule.

(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.**

Solution: If $\lambda_d = 0$, then property (1) will hold iff there exists an $i \in \{1,\dots,c\}$ such that $P(f_{opt}(\mathbf{x})=i|\mathbf{x}) = 1$. So we will either classify x in class i if we are 100% sure about this, or else we will choose doubt. Of course this is completely consistent with our intuition, because choosing doubt does not have any penalty at all, since $\lambda_d = 0$.

If $\lambda_d > \lambda_c$, then we will always classify x in the class $i \in \{1,\dots,c\}$ which gives the highest probability of correct classification. Once again this makes sense, since the cost of choosing doubt is higher than classifying $\mathbf{x}$ in any of the classes, hence our best option is to classify x in the class which gives the highest probability for a correct classification.

3 Gradient Descent Framework

In HW1, you modeled the classification of digit numbers of the MNIST dataset as a linear regression problem and solved it using its closed-form solution. In this homework, you will model it better by using classification models such as logistic regression and neural networks, and solve it using stochastic gradient descent. The goal of this problem is to show the power of modern machine learning frameworks such as TensorFlow and PyTorch, and how they can solve a wide range of problems. TensorFlow is the recommended framework in this homework and we also provide the starter kit in PyTorch.

(a) The starter code contains an implementation of linear regression for the MNIST dataset to classify 10 digits. Let $\mathbf{x}_i$ be the feature vector and $\mathbf{y}_i$ be a one-hot vector encoding of its class, i.e., $y_{i,j} = 1$ if and only if ith images is in class j and $y_{i,j} = 0$ if not. In order to use linear regression to solve a multi-class classification, we will use the *one-vs-all* strategy here. In particular, for each j we will fit a linear regression model $(\mathbf{w}_j, b_j)$ to minimize $\sum_i (\mathbf{w}_j{}^\top \mathbf{x}_i + b_j - y_{i,j})_2^2$. Then for any image $\mathbf{x}$, the prediction of its class will be $\arg\max_j (\mathbf{w}_j{}^\top \mathbf{x} + b_j)$.

Read the implementation and run it with batch size equal to 50, 100, and 200. **Attach the "epoch vs validation accuracy" plot. Report the running time for all the batch sizes. Explain the potential benefits and drawbacks of using small batch size, i.e., SGD vs GD.**

Solution:

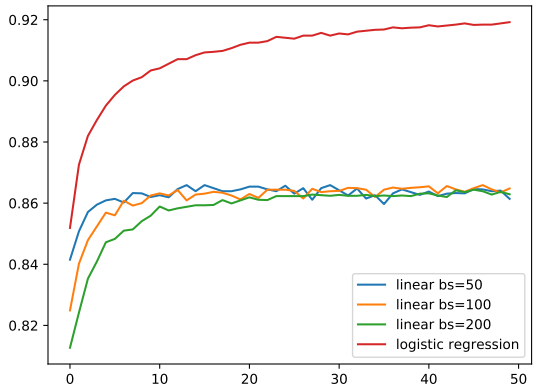


Figure 1: Epoch vs Accuracy

Figure 1 shows the plot. The running time for 50 epoches on my laptop is 51s, 28s, and 21s, respectively. The larger the batch size, the less time it takes to finish a constant number of epoches. The is because for larger batch size, there is less overhead due to the vectorization. On the other hand, when we use large batch size, we can see that it converges slower in practice. So, there is a tradeoff here.

(b) **Implement the `train_logistic` function to do the multi-class logistic regression using softmax function. Attach the plot of the "epoch vs validation accuracy" curve.**. The loss function of the multi-class logistic regression is

$$\ell = -\sum_{i=1}^{n} \log \left[\mathbf{softmax} \left(\mathbf{W}\mathbf{x}_i + \mathbf{b} \right)^\top \mathbf{y}_i \right] \tag{8}$$

where the softmax function $\mathbf{softmax} : \mathbb{R}^d \to \mathbb{R}^d$ is defined as

$$\text{softmax}(\mathbf{z})_j = \frac{\exp\left(z_j\right)}{\sum_k \exp(z_k)} = \frac{\exp\left(z_j - z'\right)}{\sum_k \exp(z_k - z')}. \tag{9}$$

Here $z' = \max_j z_j$. The expression on the right is a numerical stable formula to compute the softmax. You may NOT use any functions in `tf.nn.*`, `tf.losses.*`, and `torch.nn.*` for all the parts of this problem.

Solution:

See Figure 1.

Note that the curve in the plot depends on the learning rate. If you are using a different learning rate, you will get different plot. There is some confusion on whether we should average or sum the error for training. The answer is that they are the same because the loss function are differed by a constant scale, so that you can just multiply or divide the learning rate by your batch size to get the exact same plot.

(c) Copy your code from `train_logistic` to `train_nn` and add an additional $\tanh$ nonlinear layer to it. You loss function should be something like

$$\ell = -\sum_i \log \left[\mathbf{softmax} \left(\mathbf{W}^{(2)} \mathbf{tanh} \left(\mathbf{W}^{(1)} \mathbf{x}_i + \mathbf{b}^{(1)} \right) + \mathbf{b}^{(2)} \right)^\top \mathbf{y}_i \right]. \tag{10}$$

Attach the plot of the "epoch vs validation accuracy" curve. You have the freedom but **are NOT required to** to choose the hyper-parameters to get the best performance.

Solution:

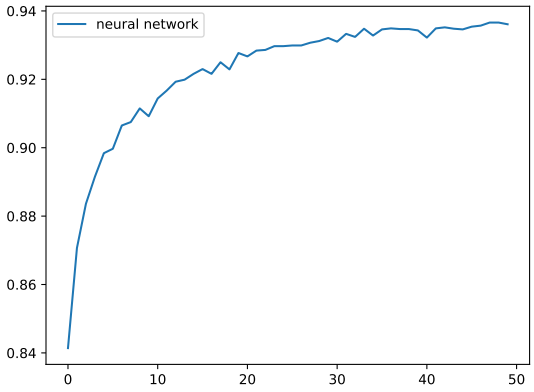


Figure 2: Epoch vs Accuracy (NN)

See Figure 2.

4 Sensors, Objects, and Localization

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.

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 six models in this problem and compare their performance:

- A *Generative Model*: We first estimate sensor locations from the training data by solving the nonlinear learst squares problem using the Gauss-Newton algorithm[1], and then use the estimated sensor locations to estimate the new object locations.
- An *Oracle Model*: This is the same as generative model except that we will use the ground truth sensor location rather than the estimated sensor location.
- 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 and/or tanh for 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 and other 3rd libraries.

Solution: See the code that we released.

(b) In the following parts, we will deal with two types of data, the "regular" data set, and the "shifted" data set. The "regular" data set has the same distribution as the training data set, while the "shifted" data set has a different distribution. **Run `plot0.py` to visualize** the sensor location, the distribution of the "regular" data set, and the distribution of the "shifted" data set. **Attach the plot**.

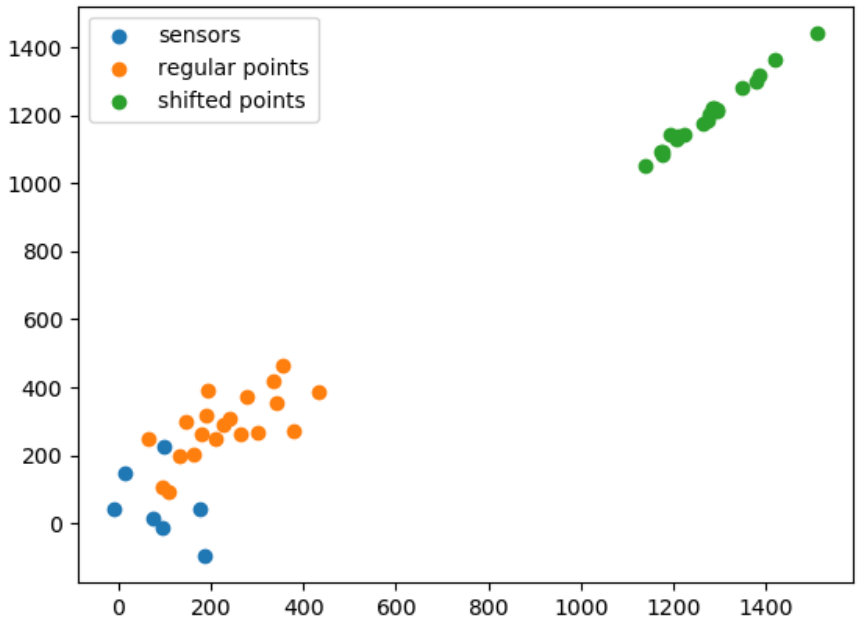
Solution:

See the attached figure.

(c) The starter code generated a set of 7 sensors and the following data sets:

- 15 training sets where n_{train} varies from 10 to 290 in increments of 20.

[1]This is not covered in the class but you can find the related information in `https://www.wikiwand.com/en/Gauss-Newton_algorithm`



- 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`.

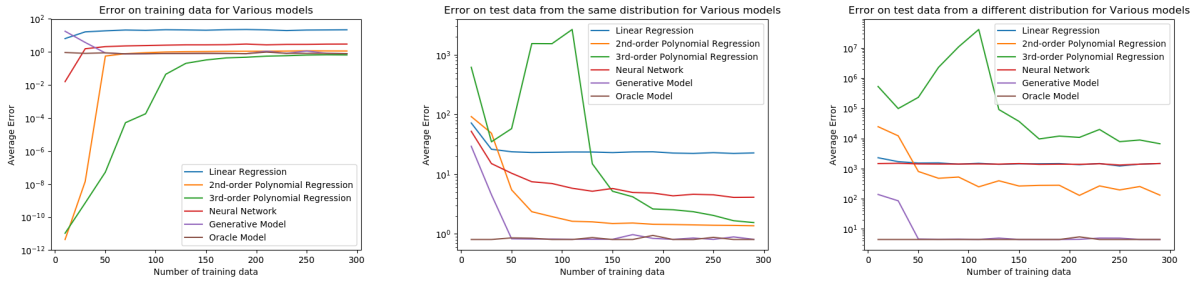
Use `plot1.py` to train each of the five models on each of the fifteen training sets. Use your results to generate three figures. Each figure should include *all* of the models on the same plot so that you can compare them:

- 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?

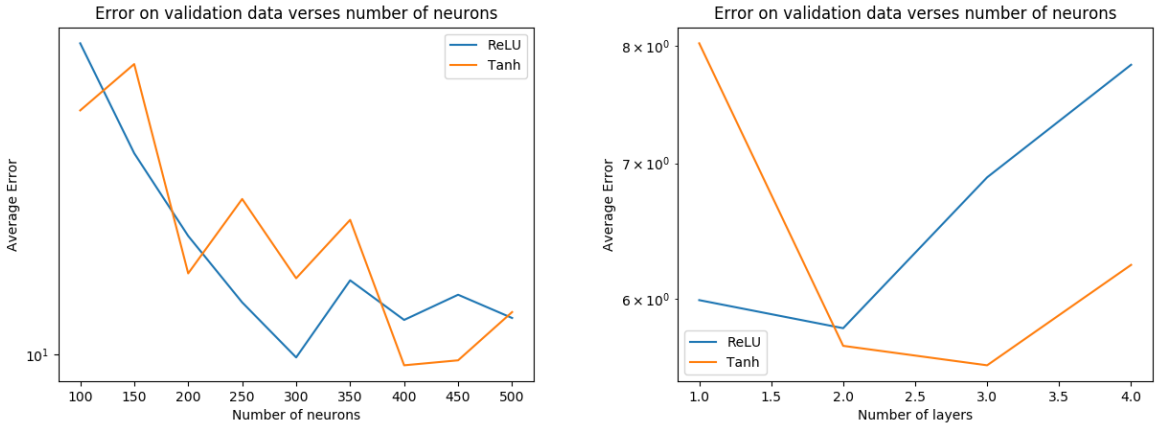
Solution: The plot by varying the numbers of data points n_{train} from 10 to 290 by 20 is shown. We observe that the generative model achieves the best performance as expected. Why? When data is drawn from a different distribution, we see that although all the models are performing worse, the generative model is still generalizing at some reasonable level. The rest of the models are behaving quite poorly indeed. This is because extrapolation is fundamentally challenging for any universal-function-approximation type model. Second-order polynomial has the second better performance. Third order polynomial performs well when the data is

generated from the same distribution, followed by neural networks. The linear model fails to perform well even with ample training data.



(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`.

Solution: The best performance is achieved at the case when number of neurons equals to $300 - 500$. (Any reasonable answer that is supported with a plot is acceptable.)



(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 $n_{\text{train}} = 200$ and $n_{\text{test}} = 1000$. **What is the best choice for k (the number of layers)? Justify your answer with plots.** The starter code is in `plot3.py`.

Solution: For $k > 1$: The number of neurons n_{neurons} for given k and ℓ, it is enough to use the approximate formula $n_{\text{neurons}} \approx (k-1)\ell^2$ (each interior layer has ℓ^2 neurons). We therefore

get $\ell \approx 100/\sqrt{k-1}$. We get the following numbers $k = 1$: $\ell \approx 1100$, $k = 2$: $\ell \approx 100$, $k = 3$: $\ell \approx 70$ and $k = 4$: $\ell \approx 50$.

The best performance is achieved at the case when number of layers equals to 3 with tanh and 2 with ReLU. (Any reasonable answer that is supported with a plot is acceptable.)

(f) You might have seen that 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 the neural network works on predicting D? The starter kit is provided in `plot4.py`.

Solution:

Figure 3: Generalization of Neural Network

No, increasing the training data does not work well as we saw in Figure 3. The reason is that the data are generated from a different distribution and neural network cannot transfer its performance to that, as mentioned above. Performance of the tuned neural network on data generated from a different distribution can be seen in the figures above. The neural network works like using a nonlinear metric to interpolate the test data using the training samples. When the distribution of testing data is not similar to the training data, it is hard for the neural network to figure it out without the prior knowledge of the underlying model.

5 Expectation Maximization (EM) Algorithm: In Action!

Suppose we have the following general mixture of Gaussians. We describe the model by a pair of random variables $(\mathbf{X}, Z)$ where $\mathbf{X}$ takes values in $\mathbb{R}^d$ and Z takes value in the set $[K] = \{1, \ldots, K\}$. The joint-distribution of the pair $(\mathbf{X}, Z)$ is given to us as follows:

$$Z \sim \text{Multinomial}(\boldsymbol{\pi}),$$
$$(\mathbf{X}|Z = k) \sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k), \quad k \in [K],$$

where $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_K)^\top$ and $\sum_{k=1}^K \pi_k = 1$. Note that we can also write

$$\mathbf{X} \sim \sum_{k=1}^K \pi_k \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k).$$

Suppose we are given a dataset $\{\mathbf{x}_i\}_{i=1}^n$ without their labels. Our goal is to identify the K-clusters of the data. To do this, we are going to estimate the parameters $\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k$ using this dataset. We are going to use the following three algorithms for this clustering task.

K-Means: For each data-point for iteration t we find its cluster by computing:

$$y_i^{(t)} = \arg\min_{j \in [K]} \|\mathbf{x}_i - \boldsymbol{\mu}_j^{(t-1)}\|^2$$
$$C_j^{(t)} = \{\mathbf{x}_i : y_i^{(t)} == j\}_{i=1}^n$$

where $\boldsymbol{\mu}_j^{(t-1)}$ denotes the mean of $C_j^{(t-1)}$, the j-th cluster in iteration $t-1$. The cluster means are then recomputed as:

$$\boldsymbol{\mu}_j^{(t)} = \frac{1}{|C_j^{(t)}|} \sum_{\mathbf{x}_i \in C_j^{(t)}} \mathbf{x}_i.$$

We can run K-means till convergence (that is we stop when the cluster memberships do not change anymore). Let us denote the final iteration as T, then the estimate of the covariances Σ_k from the final clusters can be computed as:

$$\boldsymbol{\Sigma}_j = \frac{1}{|C_j^{(T)}|} \sum_{\mathbf{x}_i \in C_j^{(T)}} (\mathbf{x}_i - \boldsymbol{\mu}_j^{(T)})(\mathbf{x}_i - \boldsymbol{\mu}_j^{(T)})^\top.$$

Notice that this method can be viewed as a "hard" version of EM.

K-QDA: Given that we also estimate the covariance, we may consider a QDA version of K-means where the covariances keep getting updated at every iteration and also play a role in determining cluster membership. The objective at the assignment-step would be given by

$$y_i^{(t)} = \arg\min_{j \in [K]} (\mathbf{x}_i - \boldsymbol{\mu}_j^{(t-1)})^\top (\boldsymbol{\Sigma}_j^{(t-1)})^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_j^{(t-1)}).$$
$$C_j^{(t)} = \{x_i : y_i^{(t)} == j\}_{i=1}^n$$

We could then use $C_j^{(t)}$ to recompute the parameters as follows:

$$\boldsymbol{\mu}_j^{(t)} = \frac{1}{|C_j^{(t)}|} \sum_{\mathbf{x}_i \in C_j^{(t)}} \mathbf{x}_i, \quad \text{and}$$

$$\boldsymbol{\Sigma}_j^{(t)} = \frac{1}{|C_j^{(t)}|} \sum_{\mathbf{x}_i \in C_j^{(t)}} (\mathbf{x}_i - \boldsymbol{\mu}_j^{(t)})(\mathbf{x}_i - \boldsymbol{\mu}_j^{(t)})^\top.$$

We again run K-QDA until convergence (that is we stop when the cluster memberships do not change anymore). Notice that, again, this method can be viewed as another variant for the "hard" EM method.

EM: The EM updates are given by

- E-step: For $k = 1, \ldots, K$ and $i = 1 \ldots, n$, we have

$$q_i^{(t)}(Z_i = k) = p(Z = k | \mathbf{X} = \mathbf{x}_i; \boldsymbol{\theta}^{(t-1)}).$$

- M-step: For $k = 1, \ldots, K$, we have

$$\pi_k^{(t)} = \frac{1}{n} \sum_{i=1}^{n} q_i^{(t)}(Z_i = k) = \frac{1}{n} \sum_{i=1}^{n} p(Z = k | \mathbf{X} = \mathbf{x}_i; \boldsymbol{\theta}^{(t-1)}),$$

$$\boldsymbol{\mu}_k^{(t)} = \frac{\sum_{i=1}^{n} q_i^{(t)}(Z_i = k)\mathbf{x}_i}{\sum_{i=1}^{n} q_i^{(t)}(Z_i = k)}, \quad \text{and}$$

$$\boldsymbol{\Sigma}_k{}^{(t)} = \frac{\sum_{i=1}^{n} q_i^{(t)}(Z_i = k)(\mathbf{x}_i - \boldsymbol{\mu}_k^{(t)})(\mathbf{x}_i - \boldsymbol{\mu}_k^{(t)})^\top}{\sum_{i=1}^{n} q_i^{(t)}(Z_i = k)}.$$

Notice that unlike previous two methods, in the EM updates, each data point contributes in determining the mean and covariance for each cluster.

We now see the three methods in action. You are provided with a code for all the above 3 algorithms (`gmm_em_kmean.py`). You can run it by calling the following function from main:

```
experiments(seed, factor, num_samples, num_clusters)
```

We assume that $\mathbf{x} \in \mathbb{R}^2$, and the default settings are number of samples is 500 ($n = 500$), and the number of clusters is 3 ($K = 3$). Notice that `seed` will determine the randomness and `factor` will determine how far apart are the clusters.

(a) Run the following setting:

```
experiments(seed=11, factor=1, num_samples=500, num_clusters=3)
```

Observe the initial guesses for the means and the plots for the 3 algorithms on convergence. **Comment on your observations. Attach the two plots for this case.**

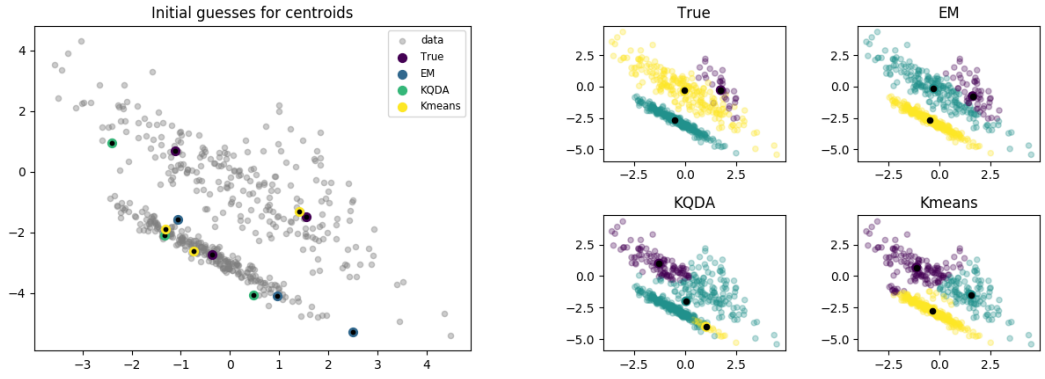
Note that the colors are used to indicate that the points that belong to different clusters, to help you visualize the data and understand the results.

Solution:

The K-means method is not able to properly cluster samples from the two gaussians that are very close to one another, as one would expect. Notice that K-QDA does not converge well, so the fact that we try to estimate the variance from the clustered data does not help when clusters are close to one another. EM performs fairly well, as expected, even in this challenging setting and despite the fact that the initial guesses for the centroids were extremely inaccurate.

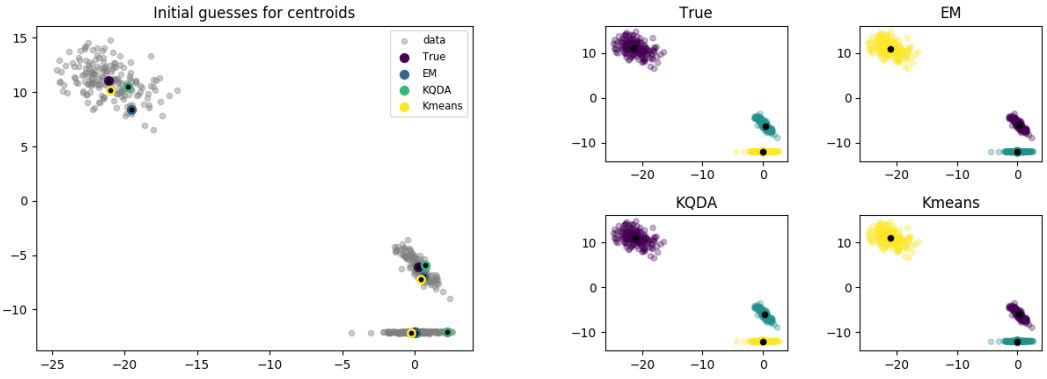
(b) **Comment on the results obtained for the following setting:**

```
experiments(seed=63, factor=10, num_samples=500, num_clusters=3)
```



and attach the two plots for this case as well.

Solution:



All three methods are able to converge to the correct clustering, given that the clusters are fairly separated and the initial guesses for the centroids are roughly in the area of the correct values, the results are what one would expect.

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.