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'm not sure why the quality of the stater code drops so dramatically this time. It might be because spring break is coming and I'm still working on this homework hoping to get it done so that I could work on other stuff.

(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. Regularized and Kernel k-Means

Recall that in k-means clustering we are attempting to minimize an objective defined as follows:

$$\min_{C_1,C_2,\ldots,C_k} \sum_{i=1}^{k} \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2, \text{ where}$$

$$\mu_i = \text{argmin}_{\mu_i \in \mathbb{R}^d} \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2 = \frac{1}{|C_i|} \sum_{x_j \in C_i} x_j, \quad i = 1, 2, \ldots, k.$$

The samples are $\{x_1, \ldots, x_n\}$, where $x_j \in \mathbb{R}^d$, and C_i is the set of samples assigned to cluster i. Each sample is assigned to exactly one cluster, and clusters are non-empty.

(a) **What is the minimum value of the objective when $k = n$ (the number of clusters equals the number of samples)?**

The minimum value of the objective is zero. This is because we could just choose sample points as the centroid of our clusters.

$$\min_{C_1,C_2,\ldots,C_k} \sum_{i=1}^{k} \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2 = 0$$

$$\text{where } \forall i \quad \mu_i = x_i$$

(b) (Regularized k-means) Suppose we add a regularization term to the above objective. That is, the objective now becomes

$$\sum_{i=1}^{k} \left(\lambda \|\mu_i\|_2^2 + \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2 \right).$$

Show that the optimum of

$$\min_{\mu_i \in \mathbb{R}^d} \lambda \|\mu_i\|_2^2 + \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2$$

is obtained at $\mu_i = \frac{1}{|C_i|+\lambda} \sum_{x_j \in C_i} x_j$.

First, the "absolute value" of a set is called "Cardinality" which is the number of elements in a set.

$$\begin{aligned}
&\nabla_{\mu_i} \left(\lambda \|\mu_i\|_2^2 + \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2 \right) \\
=&\lambda 2\mu_i + 2 \sum_{x_j \in C_i} (\mu_i - x_i) \\
=&2(\lambda + |C_i|)\mu_i - 2 \sum_{x_j \in C_i} x_i
\end{aligned}$$

Set the derivative to zero and get:

$$\begin{aligned}
2(\lambda + |C_i|)\mu_i &= 2 \sum_{x_j \in C_i} x_i \\
\mu_i &= \frac{1}{|C_i| + \lambda} \sum_{x_j \in C_i} x_j
\end{aligned}$$

(c) Here is an example where we would want to regularize clusters. Suppose there are n students who live in a $\mathbb{R}^2$ Euclidean world and who wish to share rides efficiently to Berkeley for their final exam in CS189. The university permits k vehicles which may be used for shuttling students to the exam location. The students need to figure out k good locations to meet up. The students will then each *drive* to the closest meet up point and then the shuttles will deliver them to the exam location. Define x_j to be the location of student j, and let the exam location be at $(0, 0)$. Assume that we can drive as the crow flies, i.e. by taking the shortest paths between two points. **Write down an appropriate objective function to solve this ridesharing problem and minimize the total distance that all vehicles need to travel.** Your result should be similar to the regularized k-means.

$$\min_{\mu_1,\mu_2,\ldots\mu_k\in\mathbb{R}^2} \sum_{i=1}^{k} \left(\|\mu_i\|_2 + \sum_{x_j \in C_i} \|x_j - \mu_i\|_2 \right)$$

The meanings of variables are as follows:

μ_i is the i-th meet up location

$\vec{x}_j$ is the the location of the j-th student

C_i is the set of all students who meet up at the i-th location which has the coordinates μ_i

(d) (Kernel k-means) Suppose we have a dataset $\{\vec{x}_i\}_{i=1}^n, \vec{x}_i \in \mathbb{R}^l$ that we want to split into k clusters, i.e., finding the best k-means clustering *without the regularization*. Furthermore, suppose we know *a priori* that this data is best clustered in an impractically high-dimensional feature space $\mathbb{R}^m$ with an appropriate metric. Fortunately, instead of having to deal with the (implicit) feature map $\phi : \mathbb{R}^l \to \mathbb{R}^m$ and (implicit) distance metric[1], we have a kernel function $\kappa(\vec{x}_1, \vec{x}_2) = \langle \phi(\vec{x}_1), \phi(\vec{x}_2) \rangle$ that we can compute easily on the raw samples. How should we perform the kernelized counterpart of k-means clustering?

Derive the underlined portion of this algorithm.

Algorithm 1 Kernel K-means

Require: Data matrix $X \in \mathbb{R}^{n \times l}$; Number of clusters k; kernel function $\kappa(\vec{x}_1, \vec{x}_2)$
Ensure: Cluster id $i(j)$ for each sample x_j.
  function KERNEL-K-MEANS(X, k)
    Randomly initialize $i(j)$ to be an integer in $1, 2, \ldots, k$ for each x_j.
    while not converged **do**
      for $i \leftarrow 1$ **to** k **do**
        Set $C_i = \{j \in \{1, 2, \ldots, n\} : i(j) = i\}$.
      for $j \leftarrow 1$ **to** n **do**
        Set $i(j) = \text{argmin}_i$ _____
    Return C_i for $i = 1, 2, \ldots, k$.

(Hint: there will be no explicit representation of the "means" $\vec{\mu}_i$, instead each cluster's membership itself will implicitly define the relevant quantity, in keeping with the general spirit of kernelization that we've seen elsewhere as well.)

Although there is no explicit representation of the "means" $\vec{\mu}_i$, it's still helpful to write out the "means" without regularization as:

$$\vec{\mu}_i = \frac{1}{|C_i|} \sum_{\vec{x}_k \in C_i} \Phi(\vec{x}_k)$$

Now we just need to compute the distance between each point and the "means". Here is a reference on how to do this.

$$\begin{aligned}
\text{dist}(\vec{x}_j, \vec{\mu}_i) &= \| \frac{1}{|C_i|} \sum_{\vec{x}_k \in C_i} \Phi(\vec{x}_k) - \Phi(\vec{x}_j) \|_2^2 \\
&= \frac{1}{|C_i|^2} \sum_{\vec{x}_k \in C_i} \sum_{\vec{x}_l \in C_i} < \Phi(\vec{x}_k), \Phi(\vec{x}_l) > - \frac{2}{|C_i|} \sum_{\vec{x}_k \in C_i} < \Phi(\vec{x}_k), \Phi(\vec{x}_j) > + < \Phi(\vec{x}_j), \Phi(\vec{x}_j) > \\
&= \frac{1}{|C_i|^2} \sum_{\vec{x}_k \in C_i} \sum_{\vec{x}_l \in C_i} \kappa(\vec{x}_k, \vec{x}_l) - \frac{2}{|C_i|} \sum_{\vec{x}_k \in C_i} \kappa(\vec{x}_k, \vec{x}_j) + \kappa(\vec{x}_j, \vec{x}_j)
\end{aligned}$$

The last term doesn't depend on i so we could remove it from the optimization.ref

[1] Just as how the interpretation of kernels in kernelized ridge regression involves an implicit prior/regularizer as well as an implicit feature space, we can think of kernels as generally inducing an implicit distance metric as well. Think of how you would represent the squared distance between two points in terms of pairwise inner products and operations on them.

Algorithm 2 Kernel K-means

Require: Data matrix $X \in \mathbb{R}^{n \times l}$; Number of clusters k; kernel function $\kappa(\vec{x}_1, \vec{x}_2)$
Ensure: Cluster id $i(j)$ for each sample x_j.
function KERNEL-K-MEANS(X, k)
Randomly initialize $i(j)$ to be an integer in $1, 2, \ldots, k$ for each x_j.
while not converged **do**
for $i \leftarrow 1$ **to** k **do**
Set $C_i = \{j \in \{1, 2, \ldots, n\} : i(j) = i\}$.
for $j \leftarrow 1$ **to** n **do**
Set $i(j) = \text{argmin}_i \; \frac{1}{|C_i|^2} \sum_{\vec{x}_k \in C_i} \sum_{\vec{x}_l \in C_i} \kappa(\vec{x}_k, \vec{x}_l) - \frac{2}{|C_i|} \sum_{\vec{x}_k \in C_i} \kappa(\vec{x}_k, \vec{x}_j)$
Return C_i for $i = 1, 2, \ldots, k$.

We might set $\lambda = 0$ if we don't want regularization.

Question 3. Linear Methods on Fruits and Veggies

In this problem, we will use the dataset of fruits and vegetables that was collected in HW5. The goal is to accurately classify the produce in the image. Instead of operating on the raw pixel values, we operate on extracted HSV colorspace histogram features from the image. HSV histogram features extract the color spectrum of an image, so we expect these features to serve well for distinguishing produce like bananas from apples. Denote the input state $x \in \mathbb{R}^{729}$, which is an HSV histogram generated from an RGB image with a fruit centered in it. Each data point will have a corresponding class label, which corresponds to their matching produce. Given 25 classes, we can denote the label as $y \in \{0, ..., 24\}$.

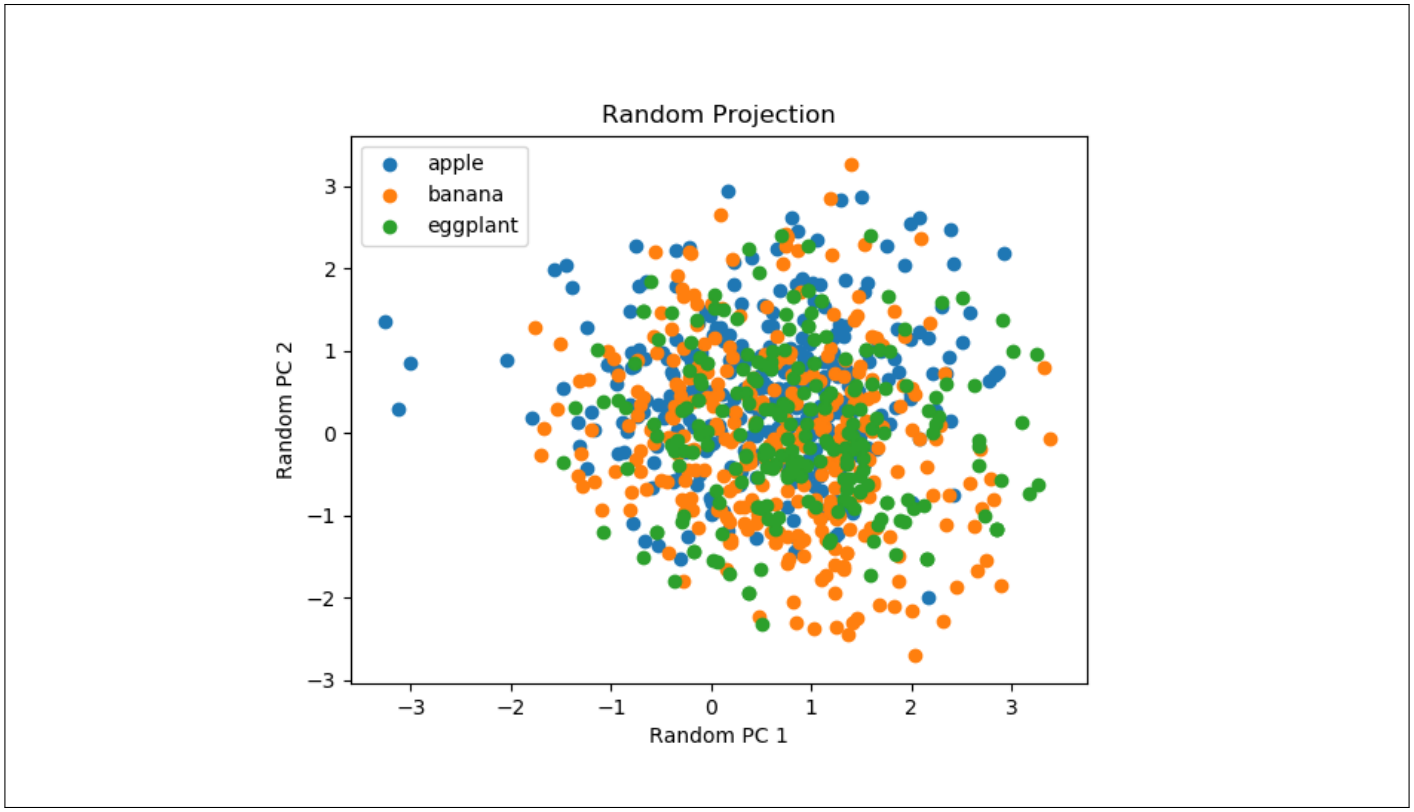
Better features would of course give better results, but we chose color spectra for an initial problem for ease of interpretation.

Classification here is still a hard problem because the state space is much larger than the amount of data we obtained in the class – we are trying to perform classification in a 729 dimensional space with only a few hundred data points from each of the 25 classes. In order to obtain higher accuracy, we will examine how to perform hyper-parameter optimization and dimensionality reduction. We will first build out each component and test on a smaller dataset of just 3 categories: apple, banana, eggplant. Then we will combine the components to perform a search over the entire dataset.

Note all python packages needed for the project, will be imported already. **DO NOT import new Python libraries.**

(a) Before we classify our data, we will study how to reduce the dimensionality of our data. We will project some of the dataset into 2D to visualize how effective different dimensionality reduction procedures are. The first method we consider is a random projection, where a matrix is randomly created and the data is linearly projected along it.

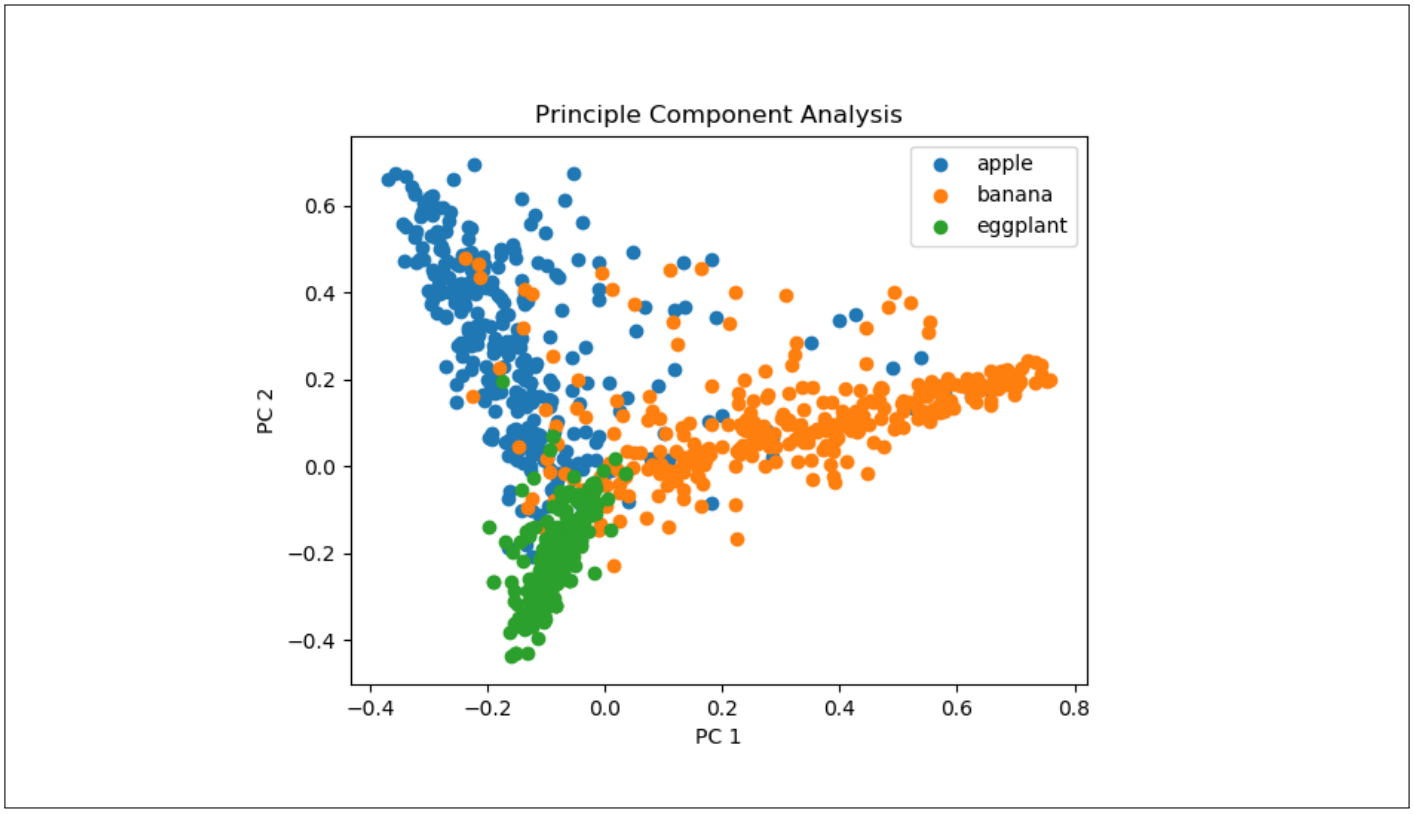
For random projections, it produces a matrix, $A \in \mathbb{R}^{2\times 729}$ where each element A_{ij} is sampled independently from a normal distribution (i.e. $A_{ij} \sim N(0, 1)$). **Run the code projection.py and report the resulting plot of the projection. You do not need to write any code.**



(b) We will next examine how well PCA performs as a dimensionality-reduction tool. PCA projects the data into the subspace with the most variance, which is determined via the covariance matrix Σ_{XX}. We can compute the principal components via the singular value decomposition $U\Lambda V^T = \Sigma_{XX}$. Collecting the first two column vectors of U in the matrix U_2, we can project our data as follows:

$$\bar{x} = U_2^T x$$

Report the projected 2D points figure. You do not need to write any code.

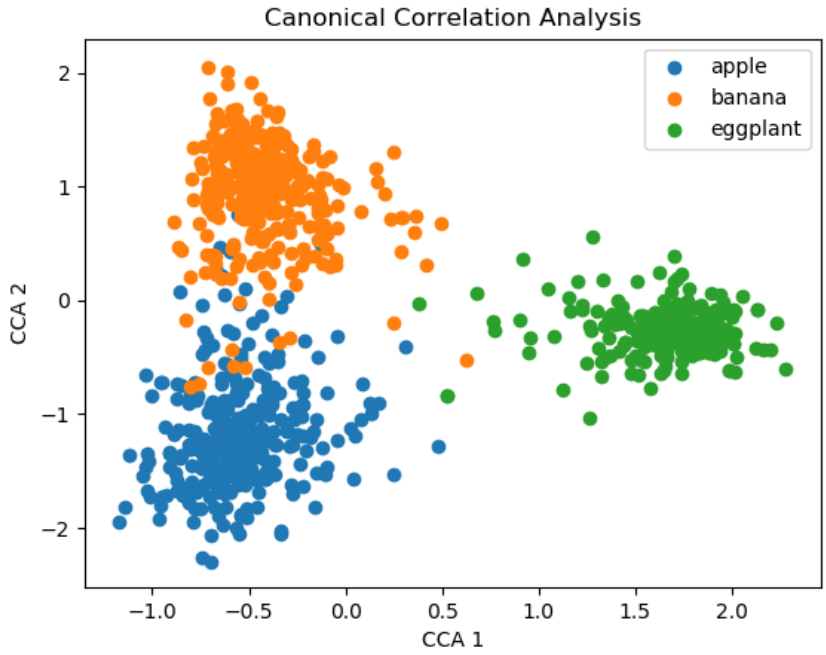


(c) Finally, we will project our data into the Canonical Variates. In order to perform CCA, we must first turn our labels y into a one-hot encoding vector $\bar{y} \in \{0,1\}^J$, where each element corresponds to the label. Note J is the number of class labels, which is $J = 3$ for this part. Next we need to compute the canonical correlation matrix $\Sigma_{XX}^{-\frac{1}{2}}\Sigma_{XY}\Sigma_{YY}^{-\frac{1}{2}}$ and compute the singular value decomposition $U\Lambda V^T = \Sigma_{XX}^{-\frac{1}{2}}\Sigma_{XY}\Sigma_{YY}^{-\frac{1}{2}}$.

We can then project to the canonical variates by using the first k columns in U, or U_k. The projection can be written as follows:

$$\bar{x} = U_k^T \Sigma_{XX}^{-\frac{1}{2}} x$$

Report the resulting plot for CCA. You do not need to write any code. Among the dimension reduction methods we have tried, i.e. random projection, PCA and CCA, which is the best for separation among classes? Which is the worst? Why do you think this happens?



Among the three dimension reduction methods we have tried, CCA is the best for separation among classes. Random projection is the worst.

Random projection is bad because we only keep the first two directions and random projection is accurate only when we keep a large number of projections.

CCA is better than PCA because it whitens the data so the variance of X is the same for all directions in X. As we can see on the graph above, three clouds are more spherical than those on the graph in part(b).

Besides, CCA takes into account Y. It maximize the prediction power of X. Therefore, the subspace we choose in CCA gives better separation between three classes.

(d) We will now examine ways to perform classification using the smaller projected space from CCA as our features. One technique is to regress to the class labels and then greedily choose the model's best guess. In this problem, we will use ridge regression to learn a mapping from the HSV histogram features to the one-hot encoding $\bar{y}$ described in the previous problem. Solve the following Ridge Regression problem:

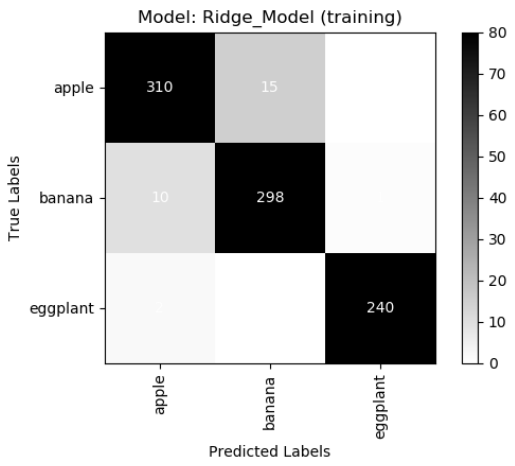
$$\min_{w} \sum_{n=1}^{N} ||\bar{y} - w^T x_n||_2^2 + \lambda ||w||_F^2$$

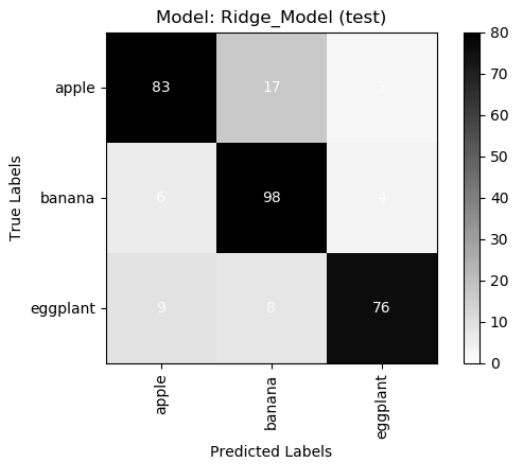
Then we will make predictions with the following function:

$$y = \underset{j \in 0,..,J-1}{\text{argmax}} \, (w^T x)_j$$

where $(w^T x)_j$ considers the j-th coordinate of the predicted vector. **You do not need to write any code.**

Run linear_classification.py. It will output a confusion matrix, a matrix that compares the actual label to the predicted label of the model. The higher the numerical value on the diagonal, the higher the percentage of correct predictions made by the model, thus the better model. **Report the Ridge Regression confusion matrix for the training data and validation data**.





Ridge_Model
training data: Computing Confusion Matrix

$$\begin{bmatrix} 310 & 15 & 0 \\ 10 & 298 & 1 \\ 2 & 0 & 240 \end{bmatrix}$$

test data: Computing Confusion Matrix

$$\begin{bmatrix} 83 & 17 & 3 \\ 6 & 98 & 4 \\ 9 & 8 & 76 \end{bmatrix}$$

```
from numpy.random import uniform
import random
import time

import numpy as np
import numpy.linalg as LA

import sys

from sklearn.linear_model import Ridge

from utils import create_one_hot_label


class Ridge_Model():

def __init__(self, class_labels):
self.lmbda = 1.0
self.NUM_CLASSES = len(class_labels)

def train_model(self, X, Y):
Y_one_hot = create_one_hot_label(Y, self.NUM_CLASSES)

self.ridge = Ridge(alpha=self.lmbda)

self.ridge.fit(X, Y_one_hot)

def eval(self, x):
x = np.array(x).reshape(-1, 1).T
prediction = self.ridge.predict(x)

return np.argmax(prediction)
```

Here is the modified linear_classification.py:

```
from numpy.random import uniform
import random
import time

import matplotlib.pyplot as plt

import numpy as np
import numpy.linalg as LA

import sys

from projection import Project2D, Projections
from confusion_mat import getConfusionMatrixPlot

from ridge_model import Ridge_Model
from qda_model import QDA_Model
from lda_model import LDA_Model
from svm_model import SVM_Model
from logistic_model import Logistic_Model

CLASS_LABELS = ['apple', 'banana', 'eggplant']

import sys
import warnings

if not sys.warnoptions:
warnings.simplefilter("ignore")
```

```
class Model():
""" Generic wrapper for specific model instance. """

def __init__(self, model):
""" Store specific pre-initialized model instance. """

self.model = model

def train_model(self, X, Y):
""" Train using specific model's training function. """

self.model.train_model(X, Y)

def test_model(self, X, Y):
""" Test using specific model's eval function. """
if hasattr(self.model, "evals"):
labels = np.array(Y)
p_labels = self.model.evals(X)

else:
labels = [] # List of actual labels
p_labels = [] # List of model's predictions
success = 0 # Number of correct predictions
total_count = 0 # Number of images

for i in range(len(X)):

x = X[i] # Test input
y = Y[i] # Actual label

y_ = self.model.eval(x) # Model's prediction
labels.append(y)
p_labels.append(y_)

if y == y_:
success += 1
total_count += 1

print("Computing Confusion Matrix \\\\")
# Compute Confusion Matrix
print('\\[')
plt = getConfusionMatrixPlot(labels, p_labels, CLASS_LABELS)
print('\\]')
plt.xlabel('True Labels')
plt.ylabel('Predicted Labels')
plt.title('Model: '+self.model.__class__.__name__)
return plt

if __name__ == "__main__":
# Load Training Data and Labels
X = list(np.load('little_x_train.npy'))
Y = list(np.load('little_y_train.npy'))

# Load Validation Data and Labels
X_val = list(np.load('little_x_val.npy'))
Y_val = list(np.load('little_y_val.npy'))

CLASS_LABELS = ['apple', 'banana', 'eggplant']

# Project Data to 200 Dimensions using CCA
feat_dim = max(X[0].shape)
projections = Projections(feat_dim, CLASS_LABELS)
cca_proj, white_cov = projections.cca_projection(X, Y, k=2)
```

```
X = projections.project(cca_proj, white_cov, X)
X_val = projections.project(cca_proj, white_cov, X_val)

####RUN RIDGE REGRESSION#####
ridge_m = Ridge_Model(CLASS_LABELS)
model = Model(ridge_m)

print('\n'+model.model.__class__.__name__.replace('_', '\\_')+'\\\\')
model.train_model(X, Y)
print('training data: ')
plt = model.test_model(X, Y)
plt.title('Model: '+model.model.__class__.__name__+' (training)')
plt.savefig('Figure_3d-training.png')
plt.close()
print('test data: ')
plt = model.test_model(X_val, Y_val)
plt.title('Model: '+model.model.__class__.__name__+' (test)')
plt.savefig('Figure_3d-test.png')
plt.close()

####RUN LDA REGRESSION#####

lda_m = LDA_Model(CLASS_LABELS)
model = Model(lda_m)

print('\n'+model.model.__class__.__name__.replace('_', '\\_')+'\\\\')
model.train_model(X, Y)
print('training data: ')
plt = model.test_model(X, Y)
plt.title('Model: '+model.model.__class__.__name__+' (training)')
plt.savefig('Figure_3e-training.png')
plt.close()
print('test data: ')
plt = model.test_model(X_val, Y_val)
plt.title('Model: '+model.model.__class__.__name__+' (test)')
plt.savefig('Figure_3e-test.png')
plt.close()

####RUN QDA REGRESSION#####

qda_m = QDA_Model(CLASS_LABELS)
model = Model(qda_m)

print('\n'+model.model.__class__.__name__.replace('_', '\\_')+'\\\\')
model.train_model(X, Y)
print('training data: ')
plt = model.test_model(X, Y)
plt.title('Model: '+model.model.__class__.__name__+' (training)')
plt.savefig('Figure_3f-training.png')
plt.close()
print('test data: ')
plt = model.test_model(X_val, Y_val)
plt.title('Model: '+model.model.__class__.__name__+' (test)')
plt.savefig('Figure_3f-test.png')
plt.close()

####RUN SVM REGRESSION#####

svm_m = SVM_Model(CLASS_LABELS)
model = Model(svm_m)

print('\n'+model.model.__class__.__name__.replace('_', '\\_')+'\\\\')
```

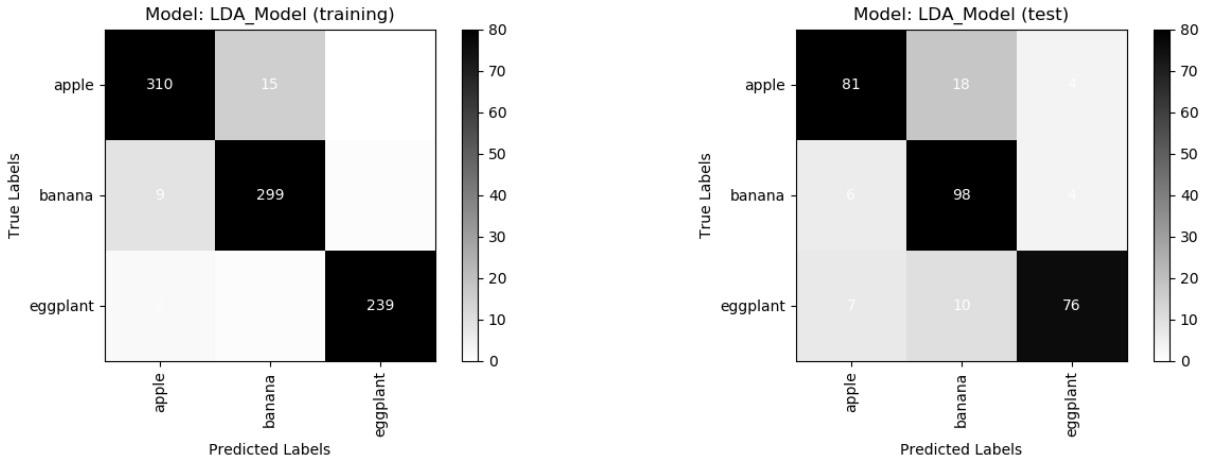
```
model.train_model(X, Y)
print('training data: ')
plt = model.test_model(X, Y)
plt.title('Model: '+model.model.__class__.__name__+' (training)')
plt.savefig('Figure_3g-training.png')
plt.close()
print('test data: ')
plt = model.test_model(X_val, Y_val)
plt.title('Model: '+model.model.__class__.__name__+' (test)')
plt.savefig('Figure_3g-test.png')
plt.close()

####RUN Logistic REGRESSION#####
lr_m = Logistic_Model(CLASS_LABELS)
model = Model(lr_m)

print('\n'+model.model.__class__.__name__.replace('_', '\\_')+'\\\\')
model.train_model(X, Y)
print('training data: ')
plt = model.test_model(X, Y)
plt.title('Model: '+model.model.__class__.__name__+' (training)')
plt.savefig('Figure_3h-training.png')
plt.close()
print('test data: ')
plt = model.test_model(X_val, Y_val)
plt.title('Model: '+model.model.__class__.__name__+' (test)')
plt.savefig('Figure_3h-test.png')
plt.close()
```

(e) Instead of performing regression, we can potentially obtain better performance by using algorithms that are more tailored for classification problems. LDA (Linear Discriminant Analysis) approaches the problem by assuming each $p(x|y=j)$ is a normal distribution with mean μ_j and covariance Σ. Notice that the covariance matrix is assumed to be the same for all the class labels.

LDA works by fitting μ_j and Σ on the dimensionality-reduced dataset. During prediction, the class with the highest likelihood is chosen.

$$y = \underset{j \in 0,\ldots,J-1}{\operatorname{argmin}} \; (x-\mu_j)^T \Sigma^{-1} (x-\mu_j)$$

Fill in the class LDA_Model. Then **run linear_classification.py and report the LDA confusion matrix for the training and validation data**.



LDA_Model
training data: Computing Confusion Matrix

$$\begin{bmatrix} 310 & 15 & 0 \\ 9 & 299 & 1 \\ 2 & 1 & 239 \end{bmatrix}$$

test data: Computing Confusion Matrix

$$\begin{bmatrix} 81 & 18 & 4 \\ 6 & 98 & 4 \\ 7 & 10 & 76 \end{bmatrix}$$

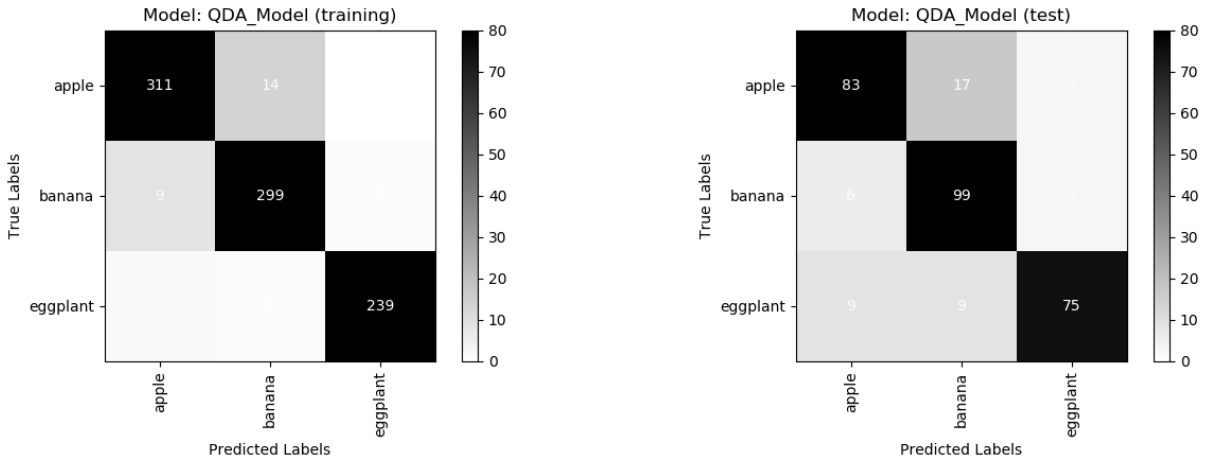
```
import random
import time

import glob
import os
import pickle
import matplotlib.pyplot as plt

import numpy as np
import numpy.linalg as LA

import sys
```

```
from numpy.linalg import inv
from numpy.linalg import det
from sklearn.svm import LinearSVC
from projection import Project2D, Projections


class LDA_Model():

def __init__(self, class_labels):
###SCALE AN IDENTITY MATRIX BY THIS TERM AND ADD TO COMPUTED COVARIANCE MATRIX TO PREVENT IT BEING
    SINGULAR ###
self.reg_cov = 0.001
self.NUM_CLASSES = len(class_labels)
self.mus = []
self.sigma = []
self.yclasses = []
self.d = -1
self.inv_sigma = []

def train_model(self, X, Y):
''''
FILL IN CODE TO TRAIN MODEL
MAKE SURE TO ADD HYPERPARAMTER TO MODEL

'''
X = np.array(X, dtype=float)
self.d = X.shape[1]
self.yclasses = np.unique(Y)
self.mus = np.zeros((self.yclasses.size, X.shape[1]))
self.sigma = np.zeros((X.shape[1], X.shape[1]), dtype=float)
for iclass, classid in enumerate(self.yclasses):
classX = X[Y == classid, :]
self.mus[iclass] = np.mean(classX, axis=0).real
classX -= self.mus[iclass]
self.sigma += classX.T.dot(classX)
self.sigma /= X.shape[0]
# Add the hyperparameter
self.sigma += self.reg_cov * np.eye(self.d)
self.inv_sigma = np.linalg.inv(self.sigma)

def eval(self, x):
''''
Fill in code to evaluate model and return a prediction
Prediction should be an integer specifying a class
'''
# x is a row vector here
predclass = -1
lossValue = np.inf
x = np.array(x).reshape(-1, 1).T.real

for iclass, classid in enumerate(self.yclasses):
tempValue = (x - self.mus[iclass]).dot(self.inv_sigma).\
dot((x - self.mus[iclass]).T)
tempValue = np.real(tempValue)

if tempValue < lossValue:
predclass = classid
lossValue = tempValue

return predclass
```

(f) LDA makes an assumption that all classes have the same covariance matrix. We can relax this assumption with QDA. In QDA, we will now parametrize each conditional distribution (still normal) by μ_j and Σ_j. The prediction function is then computed[2] as

$$y = \underset{j \in 0,..,J-1}{\operatorname{argmin}} \ (x - \mu_j)^T \Sigma_j^{-1} (x - \mu_j) + \ln(\det|\Sigma_j|)$$

Fill in the class QDA_Model. Then **run linear_classification.py and report the QDA confusion matrix for the training data and validation data**.



QDA_Model
training data: Computing Confusion Matrix

$$\begin{bmatrix} 311 & 14 & 0 \\ 9 & 299 & 1 \\ 2 & 1 & 239 \end{bmatrix}$$

test data: Computing Confusion Matrix

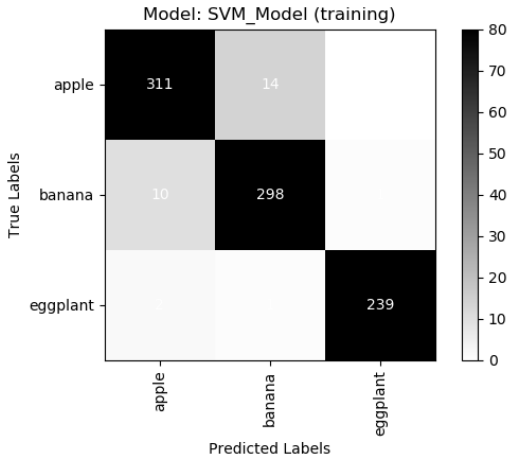
$$\begin{bmatrix} 83 & 17 & 3 \\ 6 & 99 & 3 \\ 9 & 9 & 75 \end{bmatrix}$$

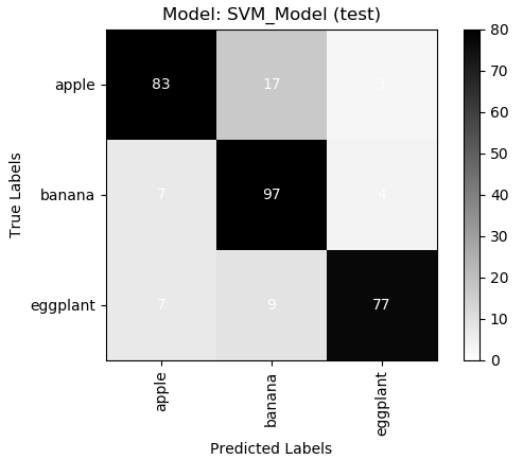
```
import random
import time

import numpy as np
import numpy.linalg as LA

from numpy.linalg import inv
from numpy.linalg import det

from projection import Project2D, Projections


class QDA_Model():
```

[2]You should verify for yourself why this is indeed the maximum likelihood way to pick a class.

```
def __init__(self, class_labels):
###SCALE AN IDENTITY MATRIX BY THIS TERM AND ADD TO COMPUTED COVARIANCE MATRIX TO PREVENT IT BEING
    SINGULAR ###
self.reg_cov = 0.01
self.NUM_CLASSES = len(class_labels)
self.mus = []
self.sigma = []
self.yclasses = []
self.d = -1
self.inv_sigma = []
self.log_det_sigma = []

def train_model(self, X, Y):

''''
FILL IN CODE TO TRAIN MODEL
MAKE SURE TO ADD HYPERPARAMTER TO MODEL

'''
X = np.array(X, dtype=float)
self.d = X.shape[1]
self.yclasses = np.unique(Y)
self.mus = np.zeros((self.yclasses.size, X.shape[1]))
self.sigma = np.zeros((self.yclasses.size, X.shape[1], X.shape[1]), dtype=float)
self.inv_sigma = np.zeros((self.yclasses.size, X.shape[1], X.shape[1]), dtype=float)
self.log_det_sigma = np.zeros(self.yclasses.size, dtype=float)
for iclass, classid in enumerate(self.yclasses):
classX = X[Y == classid, :]
self.mus[iclass] = np.mean(classX, axis=0)
classX -= self.mus[iclass]
self.sigma[iclass] = classX.T.dot(classX) / classX.shape[0]
self.sigma[iclass] += self.reg_cov * np.eye(self.d)
self.inv_sigma[iclass] = np.linalg.inv(self.sigma[iclass])
# slogdet is suggested by numpy.linalg.det for large matrix
_, self.log_det_sigma[iclass] = np.linalg.slogdet(self.sigma[iclass])


def eval(self, x):
''''
Fill in code to evaluate model and return a prediction
Prediction should be an integer specifying a class
'''
# print('x.shape'+str(x.shape))
# print('mus[0].shape'+str(self.mus[0].shape))
# x is a row vector here
predclass = -1
lossValue = np.inf
x = np.array(x).reshape(-1, 1).T
for iclass, classid in enumerate(self.yclasses):
tempValue = (x - self.mus[iclass]).dot(self.inv_sigma[iclass]). \
dot((x - self.mus[iclass]).T) + self.log_det_sigma[iclass]
if tempValue < lossValue:
predclass = classid
lossValue = tempValue

return predclass
```

(g) Let us try the Linear SVM, which fits a hyperplane to separate the dimensionality-reduced data. **Run linear_classification.py and report the Linear SVM confusion matrix for the training data and validation data. You do not need to write any code.**





SVM_Model
training data: Computing Confusion Matrix

$$\begin{bmatrix} 311 & 14 & 0 \\ 10 & 298 & 1 \\ 2 & 1 & 239 \end{bmatrix}$$

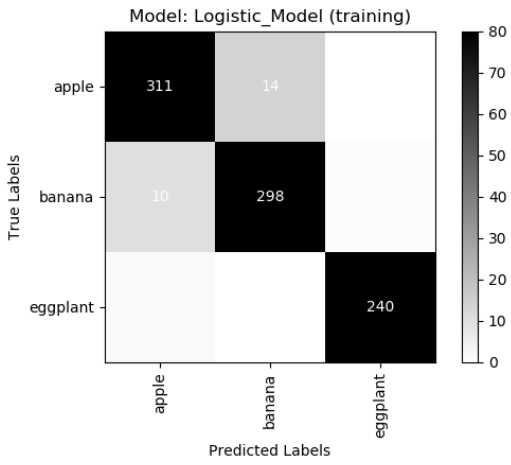
test data: Computing Confusion Matrix

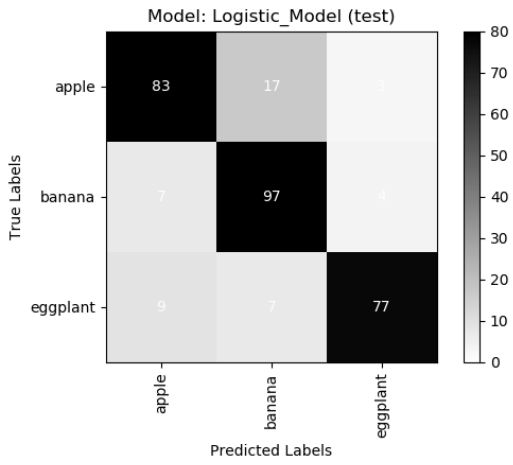
$$\begin{bmatrix} 83 & 17 & 3 \\ 7 & 97 & 4 \\ 7 & 9 & 77 \end{bmatrix}$$

```
from numpy.random import uniform
import random
import time

import matplotlib.pyplot as plt

import numpy as np
import numpy.linalg as LA

import sys

from sklearn.svm import LinearSVC
from projection import Project2D, Projections


class SVM_Model():

def __init__(self,class_labels,projection=None):

self.C = 1.0
```

```
def train_model(self,X,Y):


self.svm = LinearSVC(C=self.C, random_state=10)

self.svm.fit(X,Y)



def eval(self,x):

prediction = self.svm.predict(np.array([x]))

return prediction[0]

def scores(self, x):
return self.svm.decision_function(np.array(x))
```

(h) Let us try logistic regression. **Run linear_classification.py and report the logistic regression confusion matrix for the training data and validation data. You do not need to write any code.**





Logistic_Model
training data: Computing Confusion Matrix

$$\begin{bmatrix} 311 & 14 & 0 \\ 10 & 298 & 1 \\ 2 & 0 & 240 \end{bmatrix}$$

test data: Computing Confusion Matrix

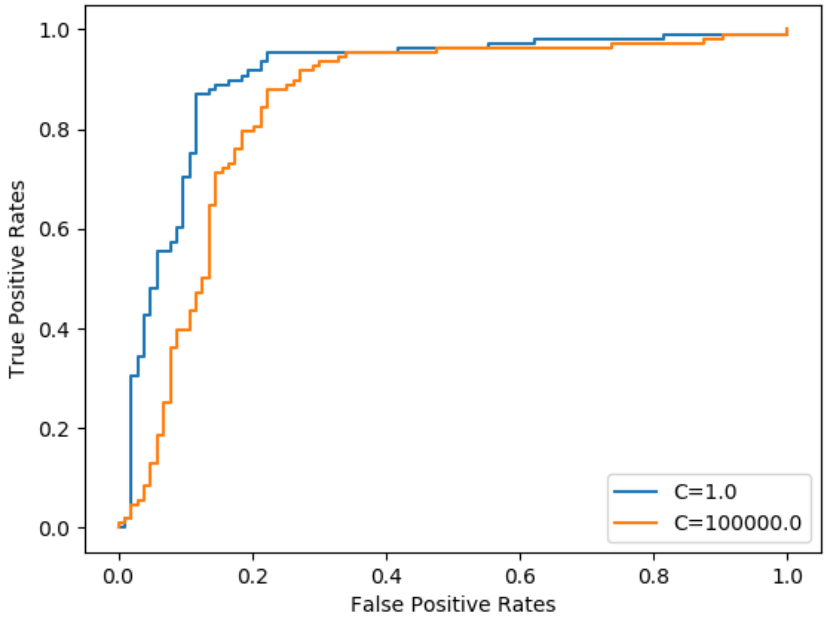
$$\begin{bmatrix} 83 & 17 & 3 \\ 7 & 97 & 4 \\ 9 & 7 & 77 \end{bmatrix}$$

	label=1	label=0
prediction=1	True Positive	False Positive
prediction=0	False Negative	True Negative

(i) In this part, we look at the Receiver Operating Characteristic (ROC), another approach to understanding a classifier's performance. In a two class classification problem, we can compare the prediction to the labels. Specifically, we count the number of True Positives (TP), False Positives (FP), False Negatives (FN) and True Negatives (TN). The true positive rate (TPR) measures how many positive examples out of all positive examples have been detected, concretely, TPR = TP/(TP+FN). The false positive rate (FPR) on the other hand, measures the proportion of negative examples that are mistakenly classified as positive, concretely, FPR = FP/(FP + TN).

A perfect classifier would have TPR = 1.0 and FPR = 0.0. However, in the real world, we usually do not have such a classifier. Requiring a higher TPR usually incurs the cost of a higher FPR, since that makes the classifier predict more positive outcomes. An ROC plots the trade off between TPR and FPR in a graphical manner. One way to get an ROC curve is to first obtain a set of scalar prediction scores, one for each of the validation samples. A higher score means that the classifier believes the sample is more likely to be a positive example. For each of the possible thresholds for the classifier, we can compute the TPR and FPR. After getting all (TPR, FPR) pairs, we can plot the ROC curve. **Finish the ROC function in roc.py. Run roc.py and report the ROC curve for an SVM with different regularization weight C. Which C is better and why?**

As we can see on the graph below, the ROC curve of $C = 1.0$ is above that of $C = 100000.0$. That is for the same amount of increase in FPR, $C = 1.0$ gains more TPR. This can be also measured by "Area under the curve".
Large regularization weight C increases the bias more than the variance decreases giving a overall high error. This is another example of bias-variance tradeoff.

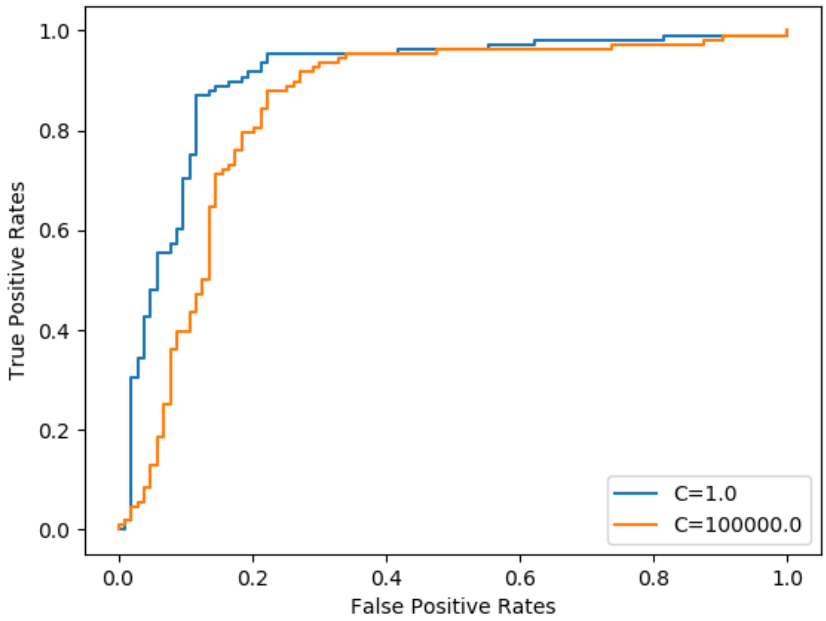


```
from numpy.random import uniform
import random
import time

import matplotlib.pyplot as plt

import numpy as np
import numpy.linalg as LA

import sys

from projection import Project2D, Projections
from confusion_mat import getConfusionMatrixPlot

from ridge_model import Ridge_Model
from qda_model import QDA_Model
from lda_model import LDA_Model
from svm_model import SVM_Model
from logistic_model import Logistic_Model

import matplotlib.pyplot as plt

CLASS_LABELS = ['apple', 'banana']


def compute_tp_fp(thres, scores, labels):
scores = np.array(scores)
prediction = (scores > thres)
tp = np.sum(prediction * labels)
tpr = 1.0 * tp / np.sum(labels)

fp = np.sum(prediction * (1 - labels))
fpr = 1.0 * fp / np.sum(1 - labels)
return tpr, fpr


def plot_ROC(tps, fps):
# plot
plt.plot(fps, tps)
plt.ylabel("True Positive Rates")
plt.xlabel("False Positive Rates")


def ROC(scores, labels):
thresholds = sorted(np.unique(scores))
thresholds = [-float("Inf")] + thresholds + [float("Inf")]
tps = []
fps = []

# student code start here
# TODO: Your code
# student code end here
n = len(labels)
tps = np.zeros(len(thresholds))
fps = np.zeros(len(thresholds))
for ithold, thold in enumerate(thresholds):
#tp = np.count_nonzero((scores > thold) & (labels == 1))
#fp = np.count_nonzero((scores > thold) & (labels == 0))
#tn = np.count_nonzero((scores <= thold) & (labels == 0))
#fn = np.count_nonzero((scores <= thold) & (labels == 1))
#tps[ithold] = tp / (tp + fn)
#fps[ithold] = fp / (fp + tn)
tps[ithold], fps[ithold] = compute_tp_fp(thold, scores, labels)
```

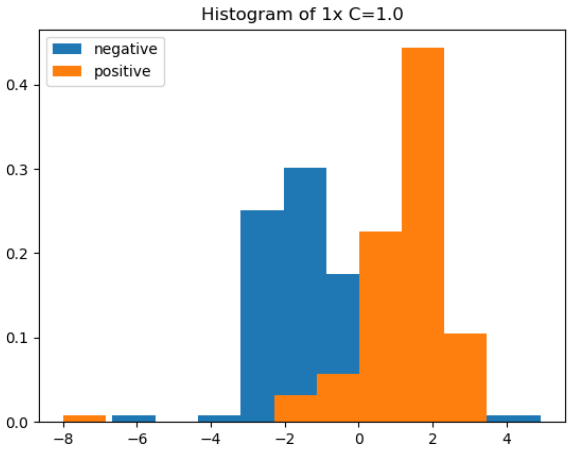
```
return tps, fps


def eval_with_ROC(method, train_X, train_Y, val_X, val_Y, C):
m = method(CLASS_LABELS)
m.C = C
m.train_model(train_X, train_Y)
scores = m.scores(val_X)
#f = open('helloworld.txt','w')
#f.write(str(scores))
#f.close()
# change the scores here
# scores = 10.0 * np.array(scores)

tps, fps = ROC(scores, val_Y)
plot_ROC(tps, fps)


def trim_data(X, Y):
# throw away the 3rd class data
X = np.array(X)
Y = np.array(Y)
retain = (Y < 2)
return X[retain, :], Y[retain]


if __name__ == "__main__":
# Load Training Data and Labels
X = list(np.load('little_x_train.npy'))
Y = list(np.load('little_y_train.npy'))
X, Y = trim_data(X, Y)

# Load Validation Data and Labels
X_val = list(np.load('little_x_val.npy'))
Y_val = list(np.load('little_y_val.npy'))
X_val, Y_val = trim_data(X_val, Y_val)

CLASS_LABELS = ['apple', 'banana']

# Project Data to 200 Dimensions using CCA
feat_dim = max(X[0].shape)
projections = Projections(feat_dim, CLASS_LABELS)
cca_proj, white_cov = projections.cca_projection(X, Y, k=2)

X = projections.project(cca_proj, white_cov, X)
X_val = projections.project(cca_proj, white_cov, X_val)

####RUN SVM REGRESSION#####
eval_with_ROC(SVM_Model, X, Y, X_val, Y_val, 1.0)
eval_with_ROC(SVM_Model, X, Y, X_val, Y_val, 100000.0)
plt.legend(["C=1.0", "C=100000.0"])
#plt.show()
plt.savefig('Figure_3i.png')
```

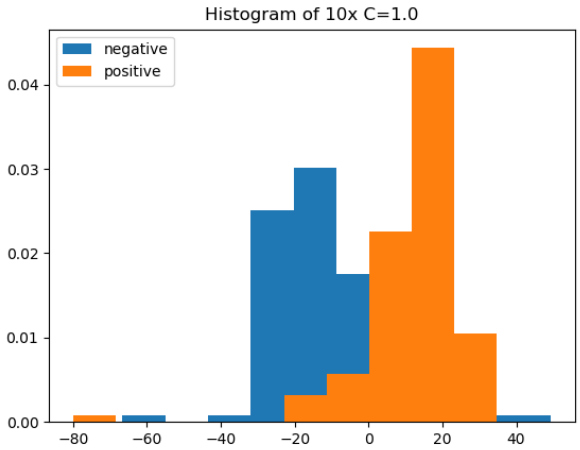
(j) If you multiply the scores output by the classifier by a factor of 10.0, how the ROC curve would change?

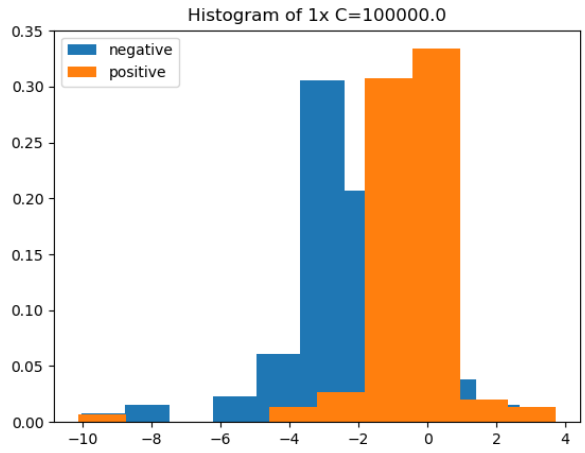
WARNING: DUE TO RANDOMNESS, THE ROC CURVES MIGHT SHIFT.
To avoid this, I set a seed to LinearSVC in svm_model.py.

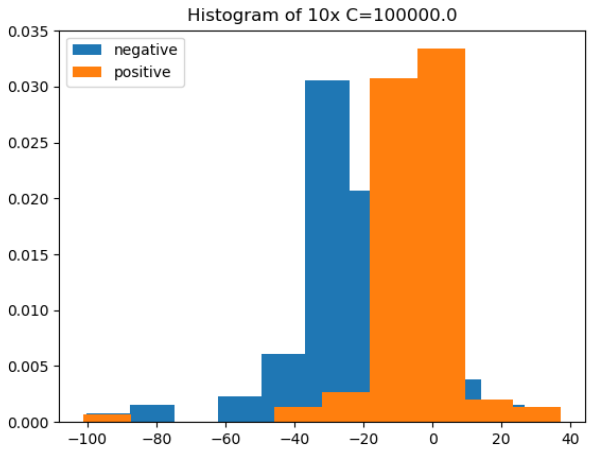


Both ROC curves look exactly the same. This is because rescaling doesn't change the rank order of the scores. This can be seen from the histograms below. The shapes of the histograms before and after rescaling are the same, which means rescaling doesn't affect the count.









```
from numpy.random import uniform
import random
import time

import matplotlib.pyplot as plt

import numpy as np
import numpy.linalg as LA

import sys

from projection import Project2D, Projections
from confusion_mat import getConfusionMatrixPlot

from ridge_model import Ridge_Model
from qda_model import QDA_Model
from lda_model import LDA_Model
from svm_model import SVM_Model
from logistic_model import Logistic_Model

import matplotlib.pyplot as plt

CLASS_LABELS = ['apple', 'banana']


def compute_tp_fp(thres, scores, labels):
scores = np.array(scores)
prediction = (scores > thres)
tp = np.sum(prediction * labels)
tpr = 1.0 * tp / np.sum(labels)

fp = np.sum(prediction * (1 - labels))
fpr = 1.0 * fp / np.sum(1 - labels)
return tpr, fpr


def plot_ROC(tps, fps):
# plot
plt.plot(fps, tps)
plt.ylabel("True Positive Rates")
plt.xlabel("False Positive Rates")


def ROC(scores, labels):
thresholds = sorted(np.unique(scores))
```

```
thresholds = [-float("Inf")] + thresholds + [float("Inf")]
tps = []
fps = []

# student code start here
# TODO: Your code
# student code end here
n = len(labels)
tps = np.zeros(len(thresholds))
fps = np.zeros(len(thresholds))
for ithold, thold in enumerate(thresholds):
tp = np.count_nonzero((scores > thold) & (labels == 1))
fp = np.count_nonzero((scores > thold) & (labels == 0))
tn = np.count_nonzero((scores <= thold) & (labels == 0))
fn = np.count_nonzero((scores <= thold) & (labels == 1))
tps[ithold] = tp / (tp + fn)
fps[ithold] = fp / (fp + tn)
return tps, fps


def eval_with_ROC(method, train_X, train_Y, val_X, val_Y, C):
m = method(CLASS_LABELS)
m.C = C
m.train_model(train_X, train_Y)
scores = m.scores(val_X)
# change the scores here
scores = 10.0 * np.array(scores)

tps, fps = ROC(scores, val_Y)
plot_ROC(tps, fps)


def trim_data(X, Y):
# throw away the 3rd class data
X = np.array(X)
Y = np.array(Y)
retain = (Y < 2)
return X[retain, :], Y[retain]


if __name__ == "__main__":
# Load Training Data and Labels
X = list(np.load('little_x_train.npy'))
Y = list(np.load('little_y_train.npy'))
X, Y = trim_data(X, Y)

# Load Validation Data and Labels
X_val = list(np.load('little_x_val.npy'))
Y_val = list(np.load('little_y_val.npy'))
X_val, Y_val = trim_data(X_val, Y_val)

CLASS_LABELS = ['apple', 'banana']

# Project Data to 200 Dimensions using CCA
feat_dim = max(X[0].shape)
projections = Projections(feat_dim, CLASS_LABELS)
cca_proj, white_cov = projections.cca_projection(X, Y, k=2)

X = projections.project(cca_proj, white_cov, X)
X_val = projections.project(cca_proj, white_cov, X_val)

####RUN SVM REGRESSION#####
eval_with_ROC(SVM_Model, X, Y, X_val, Y_val, 1.0)
```

```
eval_with_ROC(SVM_Model, X, Y, X_val, Y_val, 100000.0)
plt.legend(["C=1.0", "C=100000.0"])
#plt.show()
plt.savefig('Figure_3j.png')
```

(k) We will finally train on the full dataset and compare the different models. We will perform a grid search over the following hyperparameters:

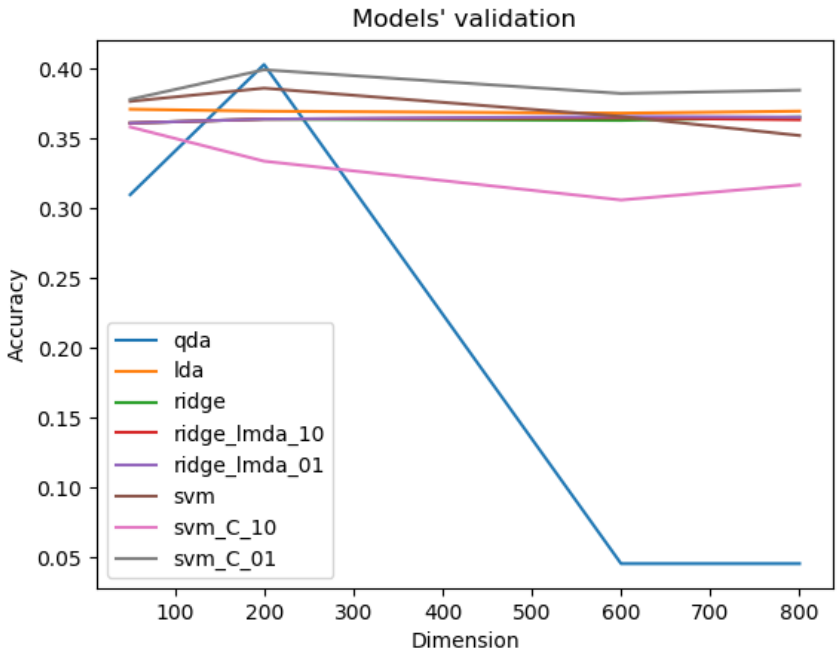
1. The regularization term λ in Ridge Regression.
2. The weighting on slack variables, C in the linear SVM.
3. The number of dimensions, k we project to using CCA.

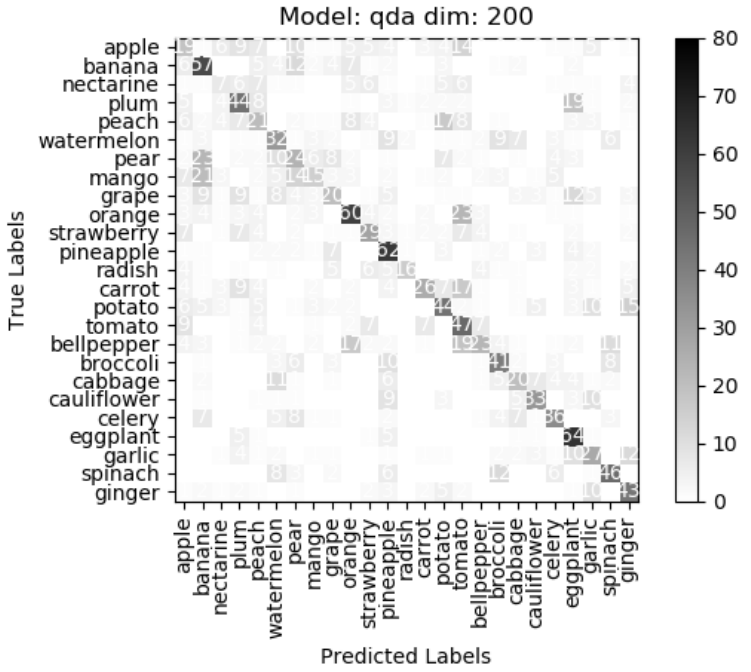
The file hyper_search.py contains the parameters that will be swept over. If the code is correctly implemented in the previous steps, this code should perform a sweep over all parameters and give the best model. **Run hyper_search.py, report the model parameters chosen, report the plot of the models's validation error, and report the best model's confusion matrix for validation data.** WARNING: This can take up to an hour to run.

As we can see from the graphs below, QDR is the best predictor overall. The highest accuracy is achieved at dim = 200.
However, it should be noted that a warning is flagged during the hyperparameter search as following:

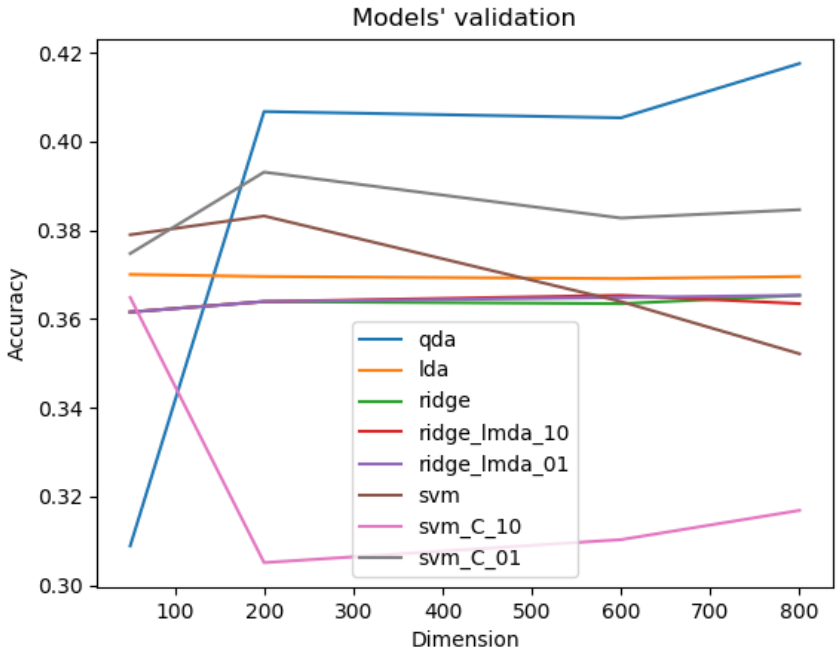
RuntimeWarning: divide by zero encountered in log

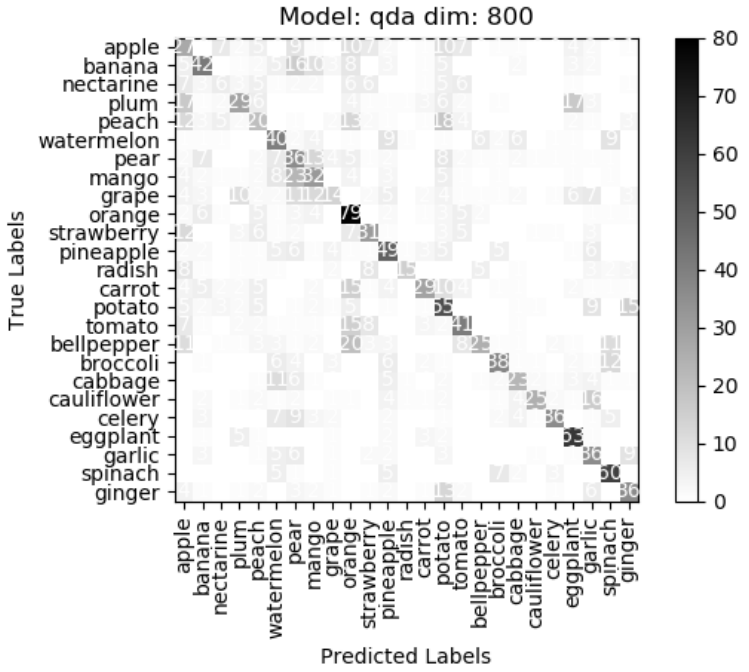
This is due to a small or large determinant of the covariance matrices.





However, if numpy.linalg.slogdet is used, the warning goes away. I think it's due to the fact that we have a tiny determinant.
In this case, QDR is the still best predictor overall. The highest accuracy is achieved at dim = 800.





The giant confusion matrix cannot be readily included here but still attached in the submitted code zip file.

```
import cv2
import IPython
from numpy.random import uniform
import random
import time

import glob
import os
import pickle
import matplotlib.pyplot as plt

import numpy as np
import numpy.linalg as LA

import sys

from projection import Project2D, Projections

from confusion_mat import getConfusionMatrix
from confusion_mat import plotConfusionMatrix

from ridge_model import Ridge_Model
from qda_model import QDA_Model
from lda_model import LDA_Model
from svm_model import SVM_Model

from utils import bmatrix

CLASS_LABELS = ['apple', 'banana', 'nectarine', 'plum', 'peach', 'watermelon', 'pear', 'mango', 'grape',
    'orange',
'strawberry', 'pineapple',
'radish', 'carrot', 'potato', 'tomato', 'bellpepper', 'broccoli', 'cabbage', 'cauliflower', 'celery',
'eggplant', 'garlic', 'spinach', 'ginger']
```

```
def eval_model(X, Y, k, model_key, proj):
# PROJECT DATA
cca_proj, white_cov = proj.cca_projection(X, Y, k=k)

X_p = proj.project(cca_proj, white_cov, X)
X_val_p = proj.project(cca_proj, white_cov, X_val)

# TRAIN MODEL
model = models[model_key]

model.train_model(X_p, Y)
acc, cm = model.test_model(X_val_p, Y_val)

return acc, cm


class Model():
""" Generic wrapper for specific model instance. """

def __init__(self, model):
""" Store specific pre-initialized model instance. """

self.model = model

def train_model(self, X, Y):
""" Train using specific model's training function. """

self.model.train_model(X, Y)

def test_model(self, X, Y):
""" Test using specific model's eval function. """
if hasattr(self.model, "evals"):
labels = np.array(Y)
p_labels = self.model.evals(X)
success = np.sum(labels == p_labels)
total_count = len(X)

else:
labels = [] # List of actual labels
p_labels = [] # List of model's predictions
success = 0 # Number of correct predictions
total_count = 0 # Number of images

for i in range(len(X)):

x = X[i] # Test input
y = Y[i] # Actual label
y_ = self.model.eval(x) # Model's prediction
labels.append(y)
p_labels.append(y_)

if y == y_:
success += 1
total_count += 1

return 1.0 * success / total_count, getConfusionMatrix(labels, p_labels)


if __name__ == "__main__":

orig_stdout = sys.stdout
```

```
f = open('Results_3k.txt', 'a')
sys.stdout = f

# Load Training Data and Labels
X = list(np.load('big_x_train.npy'))
Y = list(np.load('big_y_train.npy'))

# Load Validation Data and Labels
X_val = list(np.load('big_x_val.npy'))
Y_val = list(np.load('big_y_val.npy'))

# Project Data to 200 Dimensions using CCA
feat_dim = max(X[0].shape)
projections = Projections(feat_dim, CLASS_LABELS)

models = {} # Dictionary of key: model names, value: model instance

#########MODELS TO EVALUATE#############
qda_m = QDA_Model(CLASS_LABELS)
models['qda'] = Model(qda_m)

lda_m = LDA_Model(CLASS_LABELS)
models['lda'] = Model(lda_m)

ridge_m = Ridge_Model(CLASS_LABELS)
models['ridge'] = Model(ridge_m)

ridge_m_10 = Ridge_Model(CLASS_LABELS)
ridge_m_10.lmbda = 10.0
models['ridge_lmda_10'] = Model(ridge_m_10)

ridge_m_01 = Ridge_Model(CLASS_LABELS)
ridge_m_01.lmbda = 0.1
models['ridge_lmda_01'] = Model(ridge_m_01)

svm_m = SVM_Model(CLASS_LABELS)
models['svm'] = Model(svm_m)

svm_m_10 = SVM_Model(CLASS_LABELS)
svm_m_10.C = 10.0
models['svm_C_10'] = Model(svm_m_10)

svm_m_01 = SVM_Model(CLASS_LABELS)
svm_m_01.C = 0.1
models['svm_C_01'] = Model(svm_m_01)

#########GRID SEARCH OVER MODELS#############
highest_accuracy = 0 # Highest validation accuracy
best_model_name = None # Best model name
best_model = None # Best model instance
best_k = None
K = [50, 200, 600, 800] # List of dimensions
for model_key in models.keys():
print(model_key + '\\\\')

val_acc = [] # List of model's accuracies for each dimension
for k in K:
print("k =", k, '\\\\')

# Evaluate specific model's validation accuracy on specific dimension
acc, c_m = eval_model(X, Y, k, model_key, projections)
#acc = np.random.rand()
#c_m = np.zeros((len(CLASS_LABELS), len(CLASS_LABELS)))
```

```
val_acc.append(acc)

if acc > highest_accuracy:
highest_accuracy = acc
best_model_name = model_key
best_cm = c_m
best_k = k
# Plot specific model's accuracies across validation error
plt.plot(K, val_acc, label=model_key)

# Display aggregate plot of models across validation error
plt.legend()
plt.xlabel('Dimension')
plt.ylabel('Accuracy')
# plt.show()
plt.title('Models\' validation')
plt.savefig('Figure_3k-validation.png')

# Plot best model's confusion matrix
print("Computing Confusion Matrix \\\\")
# Compute Confusion Matrix
print('\\[')
print(bmatrix(best_cm))
print('\\]')
plt = plotConfusionMatrix(best_cm, CLASS_LABELS)

plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Model: ' + best_model_name + ' dim: ' + str(best_k))
plt.savefig('Figure_3k-bestmodel.png')

sys.stdout = orig_stdout
f.close()
```

Question 4. Expectation Maximization (EM) Algorithm: A closer look!

Hello All! Happy Spring Break! We agree this problem *APPEARS* lengthy. Don't worry! Several parts are of demo nature and several other are tutorial in nature, requiring ample explanation. Hopefully you enjoy working on this homework and learn several concepts in greater detail.

The discussion will also be going over EM using examples similar to this homework.

In this problem, we will work on different aspects of the EM algorithm to reinforce your understanding of this very important algorithm.

For the first few parts, we work with the following one-dimensional mixture model:

$$\begin{aligned} Z &\sim \text{Bernoulli}(0.5) + 1 \\ X|Z=1 &\sim \mathcal{N}(\mu_1, \sigma_1^2), \quad \text{and} \\ X|Z=2 &\sim \mathcal{N}(\mu_2, \sigma_2^2), \end{aligned}$$

i.e., Z denotes the label of the Gaussian distribution from which X is drawn. In other words, we have an equal weighted 2-mixture (since Z takes value 1 or 2 with probability 0.5 each) of Gaussians, where the variances and means for both mixtures are unknown. Note that the mixture weights are given to you. (So there is one less parameter to worry about!) For a given set of parameters, we represent the likelihood of (X, Z) by $p(X, Z; \vec{\theta})$ and its log-likelihood by $l(X, Z; \vec{\theta})$, where $\vec{\theta}$ is used to denote the set of all unknown parameters $\vec{\theta} = \{\mu_1, \mu_2, \sigma_1, \sigma_2\}$.

Given a dataset consisting of only $(x_i, i = 1, \ldots, n)$ (and no labels z_i), our goal is to approximate the maximum-likelihood estimate of the parameters $\vec{\theta}$. In the first few parts, we walk you through one way of achieving this, namely the EM algorithm.

(a) **Write down the expression for the joint likelihood $p(X = x, Z = 1; \vec{\theta})$ and $p(X = x, Z = 2; \vec{\theta})$. What is the marginal likelihood $p(X = x; \vec{\theta})$ and the log-likelihood $l(X = x; \vec{\theta})$?**

The joint likelihood is:

$$p(X = x, Z = 1; \vec{\theta}) = \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} \times \frac{1}{2}$$

$$p(X = x, Z = 2; \vec{\theta}) = \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{-\frac{(x-\mu_2)^2}{2\sigma_2^2}} \times \frac{1}{2}$$

The marginal likelihood is:

$$\begin{aligned} & p(X = x; \vec{\theta}) \\ =& p(X = x, Z = 1; \vec{\theta}) + p(X = x, Z = 2; \vec{\theta}) \\ =& \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} \times \frac{1}{2} + \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{-\frac{(x-\mu_2)^2}{2\sigma_2^2}} \times \frac{1}{2} \end{aligned}$$

The log-likelihood is:

$$\begin{aligned} & l(X = x; \vec{\theta}) \\ =& \log\Big(p(X = x; \vec{\theta})\Big) \end{aligned}$$

$$= \log\left(\frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} \times \frac{1}{2} + \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{-\frac{(x-\mu_2)^2}{2\sigma_2^2}} \times \frac{1}{2}\right)$$

$$= -\log 2 + \log\left(\frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} + \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{-\frac{(x-\mu_2)^2}{2\sigma_2^2}}\right)$$

(b) Now we are given a dataset where the label values Z are unobserved, i.e., we are given a set of n data points $\{x_i, i = 1, \ldots, n\}$. Derive the expression for the log-likelihood $l(X_1 = x_1, \ldots, X_n = x_n)$ of a given dataset $\{x_i\}_{i=1}^n$.

$$
\begin{aligned}
& l(X_1 = x_1, \ldots, X_n = x_n) \\
=& \log\Big(p(X_1 = x_1, \ldots, X_n = x_n; \vec{\theta})\Big) \\
=& -n\log 2 + \sum_{i=1}^{n} \log\left(\frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}} + \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}}\right) \\
=& -n\log 2\sqrt{2\pi} + \sum_{i=1}^{n} \log\left(\frac{1}{\sigma_1} e^{-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}} + \frac{1}{\sigma_2} e^{-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}}\right)
\end{aligned}
$$

(c) Let q denote a distribution on the (hidden) labels $\{Z_i\}_{i=1}^n$ given by

$$q(Z_1 = z_1, \ldots, Z_n = z_n) = \prod_{i=1}^{n} q_i(Z_i = z_i). \tag{1}$$

Note that since $Z \in \{1, 2\}$, q has n parameters, namely $\{q_i(Z_i = 1), i = 1, \ldots, n\}$. More generally if Z took values in $\{1, \ldots, K\}$, q would have $n(K-1)$ parameters. To simplify notation, from now on, we use the notation $l(x; \vec{\theta}) := l(X = x; \vec{\theta})$ and $p(x, k; \vec{\theta}) := p(X = x, Z = k; \vec{\theta})$. Show that for a given point x_i, we have

$$l(x_i; \vec{\theta}) \geq \mathcal{F}_i(\vec{\theta}; q_i) := \underbrace{\sum_{k=1}^{2} q_i(k) \log p(x_i, k; \vec{\theta})}_{\mathcal{L}(x_i; \vec{\theta}, q_i)} + \underbrace{\sum_{k=1}^{2} q_i(k) \log \left(\frac{1}{q_i(k)} \right)}_{H(q_i)}, \tag{2}$$

where $H(q_i)$ denotes the Shannon-entropy of the distribution q_i. Thus conclude that we obtain the following lower bound on the log-likelihood:

$$l(\{x_i\}_{i=1}^n; \vec{\theta}) \geq \mathcal{F}(\vec{\theta}; q) := \sum_{i=1}^{n} \mathcal{F}_i(\vec{\theta}; q_i). \tag{3}$$

Hint: Jensen's inequality, the concave-$\cap$ nature of the log, and reviewing lecture notes might be useful.

Notice that the right hand side of the bound depends on q while the left hand side does not.

$$\begin{aligned}
& l(x_i; \vec{\theta}) \\
=& \log \left(\sum_{z_i \in Z} p(x_i, k; \vec{\theta}) \right) \\
=& \log \left(\sum_{z_i \in Z} \frac{p(x_i, k; \vec{\theta}) q_i(Z_i = z_i)}{q_i(Z_i = z_i)} \right) \\
=& \log \left(\mathbb{E}_q \left[\frac{p(x_i, k; \vec{\theta})}{q_i(Z_i = z_i)} \right] \right) \\
\geq& \mathbb{E}_q \left[\log \left(\frac{p(x_i, k; \vec{\theta})}{q_i(Z_i = z_i)} \right) \right] \\
=& \sum_{z_i \in Z} \left(\log \left(\frac{p(x_i, k; \vec{\theta})}{q_i(Z_i = z_i)} \right) q_i(Z_i = z_i) \right) \\
=& \sum_{z_i \in Z} \Big(q_i(Z_i = z_i) \log p(x_i, k; \vec{\theta}) \Big) + \sum_{z_i \in Z} \left(q_i(Z_i = z_i) \log \frac{1}{q_i(Z_i = z_i)} \right) \\
=& \underbrace{\sum_{k=1}^{2} q_i(k) \log p(x_i, k; \vec{\theta})}_{\mathcal{L}(x_i; \vec{\theta}, q_i)} + \underbrace{\sum_{k=1}^{2} q_i(k) \log \left(\frac{1}{q_i(k)} \right)}_{H(q_i)}
\end{aligned}$$

Because we know

$$l(\{x_i\}_{i=1}^n; \vec{\theta}) = \sum_{i=1}^{n} l(x_i; \vec{\theta})$$

We can conclude that

$$l(\{x_i\}_{i=1}^n; \vec{\theta}) \geq \mathcal{F}(\vec{\theta}; q) := \sum_{i=1}^{n} \mathcal{F}_i(\vec{\theta}; q_i)$$

(d) The EM algorithm can be considered a coordinate-ascent[3] algorithm on the lower bound $\mathcal{F}(\vec{\theta}; q)$ derived in the previous part, where we ascend with respect to $\vec{\theta}$ and q in an alternating fashion. More precisely, one iteration of the EM algorithm is made up of 2-steps:

$$q^{t+1} = \arg\max_{q} \mathcal{F}(\vec{\theta}^t; q) \qquad \textbf{(E-step)}$$

$$\vec{\theta}^{t+1} \in \arg\max_{\vec{\theta}} \mathcal{F}(\vec{\theta}; q^{t+1}). \qquad \textbf{(M-step)}$$

Given an estimate $\vec{\theta}^t$, the previous part tells us that $l(\{x_i\}_{i=1}^n; \vec{\theta}^t) \geq \mathcal{F}(\vec{\theta}^t; q)$. Verify that equality holds in this bound if we plug in $q(Z_1 = z_1, \ldots, Z_n = z_n) = \prod_{i=1}^n p(Z = z_i | X = x_i; \vec{\theta}^t)$ and hence we can conclude that

$$q^{t+1}(Z_1 = z_1, \ldots, Z_n = z_n) = \prod_{i=1}^{n} p(Z = z_i | X = x_i; \vec{\theta}^t). \tag{4}$$

is a valid maximizer for the problem $\max_q \mathcal{F}(\vec{\theta}^t; q)$ and hence a valid E-step update.

$$\begin{aligned}
&\mathcal{F}_i(\vec{\theta}; q_i) \\
&= \sum_{k=1}^{2} q_i(k) \log p(x_i, k; \vec{\theta}) + \sum_{k=1}^{2} q_i(k) \log \left(\frac{1}{q_i(k)} \right) \\
&= \sum_{k=1}^{2} p(Z = k | X = x_i; \vec{\theta}^t) \log p(x_i, k; \vec{\theta}) + \sum_{k=1}^{2} p(Z = k | X = x_i; \vec{\theta}^t) \log \left(\frac{1}{p(Z = k | X = x_i; \vec{\theta}^t)} \right) \\
&= \sum_{k=1}^{2} p(Z = k | X = x_i; \vec{\theta}^t) \log \left(\frac{p(x_i, k; \vec{\theta})}{p(Z = k | X = x_i; \vec{\theta}^t)} \right) \\
&= \sum_{k=1}^{2} p(Z = k | X = x_i; \vec{\theta}^t) \log \left(\frac{p(x_i, k; \vec{\theta})}{p(Z = k | X = x_i; \vec{\theta}^t)} \right) \\
&= \sum_{k=1}^{2} p(Z = k | X = x_i; \vec{\theta}^t) \log \left(\frac{p(Z = k | X = x_i; \vec{\theta}^t) p(X = x_i; \vec{\theta}^t)}{p(Z = k | X = x_i; \vec{\theta}^t)} \right) \\
&= \sum_{k=1}^{2} p(Z = k | X = x_i; \vec{\theta}^t) \log p(X = x_i; \vec{\theta}^t) \\
&= \log p(X = x_i; \vec{\theta}^t) \underbrace{\sum_{k=1}^{2} p(Z = k | X = x_i; \vec{\theta}^t)}_{\textbf{sum of all probabilities is } 1} \\
&= \log p(X = x_i; \vec{\theta}^t) \\
&= l(x_i; \vec{\theta})
\end{aligned}$$

[3] A coordinate-ascent algorithm is just one that fixes some coordinates and maximizes the function with respect to the others as a way of taking iterative improvement steps. (By contrast, gradient-descent algorithms tend to change all the coordinates in each step, just by a little bit.) We will discuss coordinate-ascent and coordinate-descent in more detail later.

Because the above is true for all data points, the following equality is also true:

$$l(\{x_i\}_{i=1}^n; \vec{\theta^t}) = \mathcal{F}(\vec{\theta^t}; q)$$

$$\textbf{where } q(Z_i = z_i) = p(Z = z_i | X = x_i; \vec{\theta^t})$$

(e) Using equation (4) from above and the relation (1), we find that the E-step updates can be re-written as

$$q_i^{t+1}(Z_i = k) = p(Z = k|X = x_i; \vec{\theta}^t).$$

Using this relation, show that the E-step updates for the 2-mixture case are given by

$$q_i^{t+1}(Z_i = 1) = \frac{\frac{1}{\sigma_1}\exp\left(-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}\right)}{\frac{1}{\sigma_1}\exp\left(-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}\right) + \frac{1}{\sigma_2}\exp\left(-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}\right)}, \quad \textbf{and}$$

$$q_i^{t+1}(Z_i = 2) = \frac{\frac{1}{\sigma_2}\exp\left(-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}\right)}{\frac{1}{\sigma_1}\exp\left(-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}\right) + \frac{1}{\sigma_2}\exp\left(-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}\right)} = 1 - q_i^{t+1}(Z_i = 1).$$

Explain intuitively why these updates make sense.

$$\begin{aligned}
& q_i^{t+1}(Z_i = 1) \\
=& p(Z_i = 1|X_i = x_i; \vec{\theta}^t) \\
=& \frac{p(X_i = x_i|Z_i = 1; \vec{\theta}^t)p(Z_i = 1; \vec{\theta}^t)}{p(X_i = x_i; \vec{\theta}^t)} \\
=& \frac{p(X_i = x_i|Z_i = 1; \vec{\theta}^t)p(Z_i = 1; \vec{\theta}^t)}{\sum_{k=1}^{2}\left(p(X_i = x_i|Z = k; \vec{\theta}^t)p(Z_i = k; \vec{\theta}^t)\right)} \\
=& \frac{\frac{1}{2}p(X_i = x_i|Z_i = 1; \vec{\theta}^t)}{\frac{1}{2}\sum_{k=1}^{2}p(X_i = x_i|Z = k; \vec{\theta}^t)} \\
=& \frac{\frac{1}{\sqrt{2\pi\sigma_1^2}}\exp\left(-\frac{(x-\mu_1)^2}{2\sigma_1^2}\right)}{\frac{1}{\sqrt{2\pi\sigma_1^2}}\exp\left(-\frac{(x-\mu_1)^2}{2\sigma_1^2}\right) + \frac{1}{\sqrt{2\pi\sigma_2^2}}\exp\left(-\frac{(x-\mu_2)^2}{2\sigma_2^2}\right)} \\
=& \frac{\frac{1}{\sigma_1}\exp\left(-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}\right)}{\frac{1}{\sigma_1}\exp\left(-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}\right) + \frac{1}{\sigma_2}\exp\left(-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}\right)}
\end{aligned}$$

Similarly, we have:

$$\begin{aligned}
& q_i^{t+1}(Z_i = 2) \\
=& \frac{\frac{1}{\sigma_2}\exp\left(-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}\right)}{\frac{1}{\sigma_1}\exp\left(-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}\right) + \frac{1}{\sigma_2}\exp\left(-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}\right)} = 1 - q_i^{t+1}(Z_i = 1)
\end{aligned}$$

These updates make sense because the best guest for the probability of one data point belonging to a class is the proportion of that data point belonging to that class. This is determined by the fraction of its posterior probability using softmax. In other words, if one wants to guess a probability for a given data point belonging to a class, the best guess is to use the current probabilities of that data point coming from that class.

(f) We now discuss the M-step. Using the definitions from equations (2) and (3), we have that

$$\mathcal{F}(\vec{\theta}; q^{t+1}) = \sum_{i=1}^{n} (\mathcal{L}(x_i; \vec{\theta}, q_i^{t+1}) + H(q_i)) = H(q^{t+1}) + \sum_{i=1}^{n} \mathcal{L}(\vec{x}_i; \vec{\theta}, q_i^{t+1}),$$

where we have used the fact that entropy in this case is given by $H(q^{t+1}) = \sum_{i=1}^{n} H(q_i^{t+1})$. Notice that although (as computed in previous part), q^{t+1} depends on $\vec{\theta}^t$, the M-step only involves maximizing $\mathcal{F}(\vec{\theta}; q^{t+1})$ with respect to just the parameter $\vec{\theta}$ while keeping the parameter q^{t+1} fixed. Now, noting that the entropy term $H(q^{t+1})$ does not depend on the parameter $\vec{\theta}$, we conclude that the M-step simplifies to solving for

$$\arg\max_{\vec{\theta}} \underbrace{\sum_{i=1}^{n} \mathcal{L}(\vec{x}_i; \vec{\theta}, q_i^{t+1})}_{=:\mathcal{L}(\vec{\theta}; q^{t+1})}.$$

For this and the next few parts, we use the simplified notation

$$q_i^{t+1} := q_i^{t+1}(Z_i = 1) \quad \textbf{and} \quad 1 - q_i^{t+1} := q^{t+1}(Z_i = 2)$$

and recall that $\vec{\theta} = (\mu_1, \mu_2, \sigma_1, \sigma_2)$. Show that the expression for $\mathcal{L}(\vec{\theta}; q^{t+1})$ for the 2-mixture case is given by

$$\begin{aligned}
&\mathcal{L}((\mu_1, \mu_2, \sigma_1, \sigma_2); q^{t+1}) \\
&\quad = C - \sum_{i=1}^{n} \left[q_i^{t+1} \left(\frac{(x_i - \mu_1)^2}{2\sigma_1^2} + \log \sigma_1 \right) + (1 - q_i^{t+1}) \left(\frac{(x_i - \mu_2)^2}{2\sigma_2^2} + \log \sigma_2 \right) \right],
\end{aligned}$$

where C is a constant that does not depend on $\vec{\theta}$ or q^{t+1}.

$$\begin{aligned}
&\mathcal{L}((\mu_1, \mu_2, \sigma_1, \sigma_2); q^{t+1}) \\
=&\sum_{i=1}^{n} \mathcal{L}(\vec{x}_i; \vec{\theta}, q_i^{t+1}) \\
=&\sum_{i=1}^{n} \left[q_i^{t+1} \log p(x_i, Z_i = 1; \vec{\theta}) + (1 - q_i^{t+1}) \log p(x_i, Z_i = 2; \vec{\theta}) \right] \\
=&\sum_{i=1}^{n} \left[q_i^{t+1} \log \left(\frac{1}{\sqrt{2\pi\sigma_1^2}} \exp\left(-\frac{(x - \mu_1)^2}{2\sigma_1^2} \right) \right) + (1 - q_i^{t+1}) \log \left(\frac{1}{\sqrt{2\pi\sigma_2^2}} \exp\left(-\frac{(x - \mu_2)^2}{2\sigma_2^2} \right) \right) \right] \\
=&\sum_{i=1}^{n} \left[q_i^{t+1} \log(\frac{1}{\sqrt{2\pi}}) + (1 - q_i^{t+1}) \log(\frac{1}{\sqrt{2\pi}}) \right] - \sum_{i=1}^{n} \left[q_i^{t+1} \left(\frac{(x_i - \mu_1)^2}{2\sigma_1^2} + \log \sigma_1 \right) + (1 - q_i^{t+1}) \left(\frac{(x_i - \mu_2)^2}{2\sigma_2^2} + \log \sigma_2 \right) \right] \\
=&n \log(\frac{1}{\sqrt{2\pi}}) - \sum_{i=1}^{n} \left[q_i^{t+1} \left(\frac{(x_i - \mu_1)^2}{2\sigma_1^2} + \log \sigma_1 \right) + (1 - q_i^{t+1}) \left(\frac{(x_i - \mu_2)^2}{2\sigma_2^2} + \log \sigma_2 \right) \right]
\end{aligned}$$

Therefore, we have proved the above formula with $C = n \log(\frac{1}{\sqrt{2\pi}})$.

(g) Using the expression from the previous part, show that the gradients of $\mathcal{L}(\vec{\theta}; q^{t+1})$ with respect to $\mu_1, \mu_2 \sigma_1, \sigma_2$ are given by

$$\frac{\partial \mathcal{L}}{\partial \mu_1} = -\frac{\sum_{i=1}^n q_i^{t+1}(\mu_1 - x_i)}{\sigma_1^2}, \qquad \frac{\partial \mathcal{L}}{\partial \mu_2} = -\frac{\sum_{i=1}^n (1 - q_i^{t+1})(\mu_2 - x_i)}{\sigma_2^2},$$
$$\frac{\partial \mathcal{L}}{\partial \sigma_1} = \frac{\sum_{i=1}^n q_i^{t+1}(x_i - \mu_1)^2}{\sigma_1^3} - \frac{\sum_{i=1}^n q_i^{t+1}}{\sigma_1}, \quad \frac{\partial \mathcal{L}}{\partial \sigma_2} = \frac{\sum_{i=1}^n (1 - q_i^{t+1})(x_i - \mu_2)^2}{\sigma_2^2} - \frac{\sum_{i=1}^n (1 - q_i^{t+1})}{\sigma_2}.$$

$$\begin{aligned}
&\frac{\partial \mathcal{L}}{\partial \mu_1} \\
=&\frac{\partial}{\partial \mu_1}\left(-\sum_{i=1}^n \left[q_i^{t+1}\left(\frac{(x_i-\mu_1)^2}{2\sigma_1^2} + \log\sigma_1\right) + (1-q_i^{t+1})\left(\frac{(x_i-\mu_2)^2}{2\sigma_2^2} + \log\sigma_2\right)\right]\right) \\
=&\frac{\partial}{\partial \mu_1}\left(-\sum_{i=1}^n \left[q_i^{t+1}\left(\frac{(x_i-\mu_1)^2}{2\sigma_1^2}\right)\right]\right) \\
=&-\sum_{i=1}^n \left[q_i^{t+1}\frac{2(\mu_1 - x_i)}{2\sigma_1^2}\right] \\
=&-\frac{\sum_{i=1}^n q_i^{t+1}(\mu_1 - x_i)}{\sigma_1^2}
\end{aligned}$$

$$\begin{aligned}
&\frac{\partial \mathcal{L}}{\partial \mu_2} \\
=&\frac{\partial}{\partial \mu_2}\left(-\sum_{i=1}^n \left[q_i^{t+1}\left(\frac{(x_i-\mu_1)^2}{2\sigma_1^2} + \log\sigma_1\right) + (1-q_i^{t+1})\left(\frac{(x_i-\mu_2)^2}{2\sigma_2^2} + \log\sigma_2\right)\right]\right) \\
=&\frac{\partial}{\partial \mu_2}\left(-\sum_{i=1}^n \left[(1-q_i^{t+1})\left(\frac{(x_i-\mu_2)^2}{2\sigma_2^2} + \log\sigma_2\right)\right]\right) \\
=&-\sum_{i=1}^n \left[(1-q_i^{t+1})\frac{2(\mu_2 - x_i)}{2\sigma_2^2}\right] \\
=&-\frac{\sum_{i=1}^n (1-q_i^{t+1})(\mu_2 - x_i)}{\sigma_2^2}
\end{aligned}$$

$$\begin{aligned}
&\frac{\partial \mathcal{L}}{\partial \sigma_1} \\
=&\frac{\partial}{\partial \sigma_1}\left(-\sum_{i=1}^n \left[q_i^{t+1}\left(\frac{(x_i-\mu_1)^2}{2\sigma_1^2} + \log\sigma_1\right) + (1-q_i^{t+1})\left(\frac{(x_i-\mu_2)^2}{2\sigma_2^2} + \log\sigma_2\right)\right]\right) \\
=&\frac{\partial}{\partial \sigma_1}\left(-\sum_{i=1}^n \left[q_i^{t+1}\left(\frac{(x_i-\mu_1)^2}{2\sigma_1^2} + \log\sigma_1\right)\right]\right) \\
=&-\sum_{i=1}^n \left[q_i^{t+1}\frac{-2(x_i-\mu_1)^2}{2\sigma_1^3} + \frac{1}{\sigma_1}\right] \\
=&\frac{\sum_{i=1}^n q_i^{t+1}(x_i-\mu_1)^2}{\sigma_1^3} - \frac{\sum_{i=1}^n q_i^{t+1}}{\sigma_1}
\end{aligned}$$

$$
\begin{aligned}
&\frac{\partial \mathcal{L}}{\partial \sigma_2} \\
=&\frac{\partial}{\partial \sigma_2}\left(-\sum_{i=1}^{n}\left[q_i^{t+1}\left(\frac{(x_i-\mu_1)^2}{2\sigma_1^2}+\log\sigma_1\right)+(1-q_i^{t+1})\left(\frac{(x_i-\mu_2)^2}{2\sigma_2^2}+\log\sigma_2\right)\right]\right) \\
=&\frac{\partial}{\partial \sigma_2}\left(-\sum_{i=1}^{n}\left[(1-q_i^{t+1})\left(\frac{(x_i-\mu_2)^2}{2\sigma_2^2}+\log\sigma_2\right)\right]\right) \\
=&-\sum_{i=1}^{n}\left[(1-q_i^{t+1})\frac{-2(x_i-\mu_2)^2}{2\sigma_2^3}+\frac{1}{\sigma_2}\right] \\
=&\frac{\sum_{i=1}^{n}(1-q_i^{t+1})(x_i-\mu_2)^2}{\sigma_2^3}-\frac{\sum_{i=1}^{n}(1-q_i^{t+1})}{\sigma_2}
\end{aligned}
$$

(h) Typically, the M-step updates are computed using the stationary points for $F(\vec{\theta}; q^{t+1})$. Using the expressions from previous parts and setting the gradients to zero, conclude that the M-step updates are given by

$$\mu_1^{t+1} = \frac{\sum_{i=1}^n q_i^{t+1} x_i}{\sum_{i=1}^n q_i^{t+1}}, \qquad \mu_2^{t+1} = \frac{\sum_{i=1}^n (1-q_i^{t+1}) x_i}{\sum_{i=1}^n (1-q_i^{t+1})},$$
$$(\sigma_1^2)^{(t+1)} = \frac{\sum_{i=1}^n q_i^{t+1}(x_i-\mu_1^{t+1})^2}{\sum_{i=1}^n q_i^{t+1}}, \qquad (\sigma_2^2)^{(t+1)} = \frac{\sum_{i=1}^n (1-q_i^{t+1})(x_i-\mu_2^{t+1})^2}{\sum_{i=1}^n (1-q_i^{t+1})}$$

Explain intuitively why these updates make sense.

We copied the first term of complete log likelihood from part(c) and take the derivative:

$$\begin{aligned}
&\frac{\partial}{\partial \mu_1^{t+1}} \sum_{i=1}^n \mathcal{L}(x_i; \vec{\theta}, q_i) \\
=&\frac{\partial}{\partial \mu_1^{t+1}} \sum_{i=1}^n \sum_{k=1}^2 q_i^{t+1} \log p(x_i, k; \vec{\theta}) \\
=&\frac{\partial}{\partial \mu_1^{t+1}} \sum_{i=1}^n \left[q_i^{t+1} \log p(x_i, Z_1 = 1; \vec{\theta}) + (1-q_i^{t+1}) \log p(x_i, Z_1 = 2; \vec{\theta}) \right] \\
=&\frac{\partial}{\partial \mu_1^{t+1}} \sum_{i=1}^n \left[q_i^{t+1} \log \left(\frac{1}{\sqrt{2\pi\sigma_1^2}} \exp\left(-\frac{(x_i-\mu_1^{t+1})^2}{2\sigma_1^2} \right) \right) + (1-q_i^{t+1}) \log \left(\frac{1}{\sqrt{2\pi\sigma_2 2}} \exp\left(-\frac{(x_i-\mu_2)^2}{2\sigma_2^2} \right) \right) \right] \\
=&\frac{\partial}{\partial \mu_1^{t+1}} \sum_{i=1}^n \left[q_i^{t+1} \log \left(\frac{1}{\sqrt{2\pi\sigma_1^2}} \exp\left(-\frac{(x_i-\mu_1^{t+1})^2}{2\sigma_1^2} \right) \right) \right] \\
=&\frac{\partial}{\partial \mu_1^{t+1}} \sum_{i=1}^n \left[q_i^{t+1} \log \left(\frac{1}{\sqrt{2\pi\sigma_1^2}} \right) - q_i^{t+1} \frac{(x_i-\mu_1^{t+1})^2}{2\sigma_1^2} \right] \\
=&-\frac{\partial}{\partial \mu_1^{t+1}} \sum_{i=1}^n \left[q_i^{t+1} \frac{(x_i-\mu_1^{t+1})^2}{2\sigma_1^2} \right] \\
=&-\sum_{i=1}^n \left[q_i^{t+1} \frac{2(\mu_1^{t+1}-x_i)}{2\sigma_1^2} \right] \\
=&-\sum_{i=1}^n \left[q_i^{t+1} \frac{\mu_1^{t+1}-x_i}{\sigma_1^2} \right]
\end{aligned}$$

Set the derivative to be zero and get:

$$\frac{\partial}{\partial \mu_1^{t+1}} \sum_{i=1}^n \mathcal{L}(x_i; \vec{\theta}, q_i) = 0$$
$$-\sum_{i=1}^n \left[q_i^{t+1} \frac{\mu_1^{t+1}-x_i}{\sigma_1^2} \right] = 0$$
$$\sum_{i=1}^n q_i^{t+1} \mu_1^{t+1} = \sum_{i=1}^n q_i^{t+1} x_i$$

$$\mu_1^{t+1} = \frac{\sum_{i=1}^n q_i^{t+1} x_i}{\sum_{i=1}^n q_i^{t+1}}$$

Similarity, we can get:

$$\frac{\partial}{\partial \mu_2^{t+1}} \sum_{i=1}^n \mathcal{L}(x_i; \vec{\theta}, q_i) = 0$$

$$-\sum_{i=1}^n \left[(1 - q_i^{t+1}) \frac{\mu_2^{t+1} - x_i}{\sigma_2^2} \right] = 0$$

$$\sum_{i=1}^n (1 - q_i^{t+1}) \mu_2^{t+1} = \sum_{i=1}^n (1 - q_i^{t+1}) x_i$$

$$\mu_2^{t+1} = \frac{\sum_{i=1}^n (1 - q_i^{t+1}) x_i}{\sum_{i=1}^n (1 - q_i^{t+1})}$$

Now let's take the derivative with respect to σ_1:

$$\begin{aligned}
&\frac{\partial}{\partial \sigma_1^{t+1}} \sum_{i=1}^n \mathcal{L}(x_i; \vec{\theta}, q_i) \\
=&\frac{\partial}{\partial \sigma_1^{t+1}} \sum_{i=1}^n \left[q_i^{t+1} \log \left(\frac{1}{\sqrt{2\pi\sigma_1^2}} \right) - q_i^{t+1} \frac{(x_i - \mu_1^{t+1})^2}{2\sigma_1^2} \right] \\
=&\frac{\partial}{\partial \sigma_1^{t+1}} \sum_{i=1}^n \left[q_i^{t+1} \log \left(\frac{1}{\sigma_1} \right) - q_i^{t+1} \frac{(x_i - \mu_1^{t+1})^2}{2\sigma_1^2} \right] \\
=&\sum_{i=1}^n \left[q_i^{t+1} \frac{2(x_i - \mu_1^{t+1})^2}{2\sigma_1^3} - q_i^{t+1} \frac{1}{\sigma_1} \right] \\
=&\sum_{i=1}^n \left[q_i^{t+1} \frac{(x_i - \mu_1^{t+1})^2}{\sigma_1^3} - q_i^{t+1} \frac{1}{\sigma_1} \right]
\end{aligned}$$

Now we set it to zero and get:

$$\frac{\partial}{\partial \sigma_1^{t+1}} \sum_{i=1}^n \mathcal{L}(x_i; \vec{\theta}, q_i) = 0$$

$$\sum_{i=1}^n \left[q_i^{t+1} \frac{(x_i - \mu_1^{t+1})^2}{\sigma_1^3} - q_i^{t+1} \frac{1}{\sigma_1} \right] = 0$$

$$\sum_{i=1}^n \left[q_i^{t+1} \frac{(x_i - \mu_1^{t+1})^2}{\sigma_1^2} \right] = \sum_{i=1}^n q_i^{t+1}$$

$$(\sigma_1^2)^{(t+1)} = \frac{\sum_{i=1}^n q_i^{t+1} (x_i - \mu_1^{t+1})^2}{\sum_{i=1}^n q_i^{t+1}}$$

Similarly, we have:

$$\frac{\partial}{\partial \sigma_2^{t+1}} \sum_{i=1}^n \mathcal{L}(x_i; \vec{\theta}, q_i) = 0$$

$$\sum_{i=1}^n \left[(1 - q_i^{t+1}) \frac{(x_i - \mu_2^{t+1})^2}{\sigma_2^3} - (1 - q_i^{t+1}) \frac{1}{\sigma_2} \right] = 0$$

$$\sum_{i=1}^{n}\left[(1-q_i^{t+1})\frac{(x_i-\mu_2^{t+1})^2}{\sigma_2^2}\right]=\sum_{i=1}^{n}(1-q_i^{t+1})$$

$$(\sigma_2^2)^{(t+1)}=\frac{\sum_{i=1}^{n}(1-q_i^{t+1})(x_i-\mu_2^{t+1})^2}{\sum_{i=1}^{n}(1-q_i^{t+1})}$$

Intuitively, all these updates make sense because if we know how much a data point belongs to a class, we can just use that proportion of the data point to fit a Gaussian distribution. It's similar to the way we calculate posterior probabilities using priors.

(i) For the next few parts, we simplify the mixture model. We work with the following simpler one-dimensional mixture model that has only a single unknown parameter:

$$\begin{aligned} Z &\sim \textbf{Bernoulli}(0.5) + 1 \\ X|Z = 1 &\sim \mathcal{N}(\mu, 1), \quad \textbf{and} \\ X|Z = 2 &\sim \mathcal{N}(-\mu, 1), \end{aligned}$$

where Z denotes the label of the Gaussian from which X is drawn. Given a set of observations only for X (i.e., the labels are unobserved), our goal is to infer the maximum-likelihood parameter μ. Using computations similar to part (a), we can conclude that the likelihood function in this simpler set-up is given by

$$p(X = x; \mu) = \frac{1}{2}\frac{e^{-\frac{1}{2}(x-\mu)^2}}{\sqrt{2\pi}} + \frac{1}{2}\frac{e^{-\frac{1}{2}(x+\mu)^2}}{\sqrt{2\pi}}.$$

For a given dataset $\{x_i, i = 1, \ldots, n\}$, what is the log-likelihood $l(\{x_i\}_{i=1}^n; \mu)$ as a function of μ?

$$\begin{aligned} & l(\{x_i\}_{i=1}^n; \mu) \\ =& \log\left(p(X_1 = x_1, \ldots, X_n = x_n; \mu)\right) \\ =& -n \log 2\sqrt{2\pi} + \sum_{i=1}^{n} \log\left(e^{-\frac{(x_i-\mu)^2}{2}} + e^{-\frac{(x_i+\mu)^2}{2}}\right) \end{aligned}$$

(j) We now discuss EM updates for the set up introduced in the previous part. Let μ_t denote the estimate for μ at time t. First, we derive the E-step updates. Using part (d) (equation (4)), show that the E-step updates simplify to

$$q_i^{t+1}(Z_i = 1) = \frac{\exp(-(x_i - \mu_t)^2/2)}{\exp(-(x_i - \mu_t)^2/2) + \exp(-(x_i + \mu_t)^2/2)}.$$

Notice that these updates can also be derived by plugging in $\mu_1 = \mu$ and $\mu_2 = -\mu$ in the updates given in part (e).

Copied from part e and set $\mu_1 = \mu$, $\mu_2 = -\mu$, $\sigma_1 = \sigma_2 = 1$.

$$\begin{aligned}
& q_i^{t+1}(Z_i = 1) \\
=& \frac{\frac{1}{\sigma_1} \exp\left(-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}\right)}{\frac{1}{\sigma_1} \exp\left(-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}\right) + \frac{1}{\sigma_2} \exp\left(-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}\right)} \\
=& \frac{\exp\left(-\frac{(x_i-\mu_t)^2}{2}\right)}{\exp\left(-\frac{(x_i-\mu_t)^2}{2}\right) + \exp\left(-\frac{(x_i+\mu_t)^2}{2}\right)} \\
=& \frac{\exp(-(x_i - \mu_t)^2/2)}{\exp(-(x_i - \mu_t)^2/2) + \exp(-(x_i + \mu_t)^2/2)}
\end{aligned}$$

(k) Next, we derive the M-step update. Note that we can NOT simply plug in $\mu_1 = \mu$ in the updates obtained in part (h), because the parameters are shared between the two mixtures. However, we can still make use of some of our previous computations for this simpler case. Plugging in $\mu_1 = \mu$ and $\mu_2 = -\mu$, $\sigma_1 = \sigma_2 = 1$ in part (f), show that the objective for the M-step is given by

$$\mathcal{L}(\mu; q^{t+1}) = C - \sum_{i=1}^{n} \left(q_i^{t+1} \frac{(x_i - \mu)^2}{2} + (1 - q_i^{t+1}) \frac{(x_i + \mu)^2}{2} \right).$$

where C is a constant independent of μ. Compute the expression for the gradient $\frac{d}{d\mu}(\mathcal{L}(\mu; q^{t+1}))$. And by setting the gradient to zero, conclude that the M-step update at time $t+1$ is given by

$$\mu_{t+1} = \frac{\sum_{i=1}^{n} (2q_i^{t+1} - 1)x_i}{\sum_{i=1}^{n} (2q_i^{t+1} - 1)} = \frac{1}{n} \sum_{i=1}^{n} (2q_i^{t+1} - 1)x_i.$$

$$\begin{aligned}
&\mathcal{L}((\mu_1 = \mu, \mu_2 = \mu, \sigma_1 = 1, \sigma_2 = 1); q^{t+1}) \\
=&C - \sum_{i=1}^{n} \left(q_i^{t+1} \left(\frac{(x_i - \mu_1)^2}{2\sigma_1^2} + \log \sigma_1 \right) + (1 - q_i^{t+1}) \left(\frac{(x_i - \mu_2)^2}{2\sigma_2^2} + \log \sigma_2 \right) \right) \\
=&C - \sum_{i=1}^{n} \left(q_i^{t+1} \frac{(x_i - \mu)^2}{2} + (1 - q_i^{t+1}) \frac{(x_i + \mu)^2}{2} \right)
\end{aligned}$$

$$\begin{aligned}
&\frac{\partial \mathcal{L}(\mu; q^{t+1})}{\partial \mu} \\
=& - \frac{\partial}{\partial \mu} \left[\sum_{i=1}^{n} \left(q_i^{t+1} \frac{(x_i - \mu)^2}{2} + (1 - q_i^{t+1}) \frac{(x_i + \mu)^2}{2} \right) \right] \\
=& - \sum_{i=1}^{n} \left(q_i^{t+1} (\mu - x_i) + (1 - q_i^{t+1})(\mu + x_i) \right) \\
=& \sum_{i=1}^{n} \left((2q_i^{t+1} - 1)x_i - \mu \right)
\end{aligned}$$

Set the derivative to zero and get:

$$\begin{aligned}
\frac{\partial \mathcal{L}(\mu_{t+1}; q^{t+1})}{\partial \mu_{t+1}} &= 0 \\
\sum_{i=1}^{n} \left((2q_i^{t+1} - 1)x_i - \mu_{t+1} \right) &= 0 \\
\sum_{i=1}^{n} (2q_i^{t+1} - 1)\vec{x}_i &= \sum_{i=1}^{n} \mu_{t+1} \\
\mu_{t+1} &= \frac{1}{n} \sum_{i=1}^{n} (2q_i^{t+1} - 1)x_i
\end{aligned}$$

(l) Let us now consider a direct optimization method to estimate the MLE for μ: Doing a gradient ascent algorithm directly on the complete log-likelihood function $l(\{x_i\}_{i=1}^n;\mu)/n$ (scaling with n is a bit natural here!). Compute the gradient $\frac{d}{d\mu}(l(\{x_i\}_{i=1}^n;\mu)/n)$ and show that it is equal to

$$\frac{d}{d\mu}\left(\frac{1}{n}l(\{x_i\}_{i=1}^n;\mu)\right)=\left[\frac{1}{n}\sum_{i=1}^n(2w_i-1)x_i\right]-\mu,\quad \textbf{where}\quad w_i(\mu)=\frac{e^{-\frac{(x_i-\mu)^2}{2}}}{e^{-\frac{(x_i-\mu)^2}{2}}+e^{-\frac{(x_i+\mu)^2}{2}}}.$$

Finally conclude that the gradient ascent scheme with step size α is given by

$$\begin{aligned}\mu_{t+1}^{\mathbf{GA}}&=\mu_t^{\mathbf{GA}}+\alpha\frac{d}{d\mu}l(\{x_i\}_{i=1}^n;\mu)\Big|_{\mu=\mu_t^{\mathbf{GA}}}\\&=(1-\alpha)\mu_t^{\mathbf{GA}}+\alpha\left[\frac{1}{n}\sum_{i=1}^n(2w_i(\mu_t^{\mathbf{GA}})-1)x_i\right].\end{aligned}$$

Here we notice that $w_i(\mu)$ is our posterior probability q_i in previous parts. We can take a shortcut here by fixing $w_i(\mu)$ and compute the derivative:

$$\begin{aligned}&\frac{d}{d\mu}\left(\frac{1}{n}l(\{x_i\}_{i=1}^n;\mu)\right)\\=&\frac{d}{d\mu}\left(\frac{1}{n}\mathcal{L}(\mu;\vec{w})+\frac{1}{n}H(\vec{w})\right)\\=&\frac{d}{d\mu}\left(\frac{1}{n}\mathcal{L}(\mu;\vec{w})\right)\\=&\sum_{i=1}^n\left((2q_i^{t+1}-1)x_i-\mu\right)\\=&\frac{1}{n}\sum_{i=1}^n(2w_i(\mu)-1)x_i-\frac{1}{n}\sum_{i=1}^n\mu\\=&\left[\frac{1}{n}\sum_{i=1}^n(2w_i(\mu)-1)x_i\right]-\mu\end{aligned}$$

Alternatively, we could derive it from the original formula:

$$\begin{aligned}&\frac{d}{d\mu}\left(\frac{1}{n}l(\{x_i\}_{i=1}^n;\mu)\right)\\=&\frac{d}{d\mu}\frac{1}{n}\left[-n\log 2\sqrt{2\pi}+\sum_{i=1}^n\log\left(e^{-\frac{(x_i-\mu)^2}{2}}+e^{-\frac{(x_i+\mu)^2}{2}}\right)\right]\\=&\frac{1}{n}\sum_{i=1}^n\left[\frac{d}{d\mu}\log\left(e^{-\frac{(x_i-\mu)^2}{2}}+e^{-\frac{(x_i+\mu)^2}{2}}\right)\right]\\=&\frac{1}{n}\sum_{i=1}^n\left[\frac{1}{e^{-\frac{(x_i-\mu)^2}{2}}+e^{-\frac{(x_i+\mu)^2}{2}}}\left((x_i-\mu)e^{-\frac{(x_i-\mu)^2}{2}}-(x_i+\mu)e^{-\frac{(x_i+\mu)^2}{2}}\right)\right]\\=&\frac{1}{n}\sum_{i=1}^n\left[\frac{1}{e^{-\frac{(x_i-\mu)^2}{2}}+e^{-\frac{(x_i+\mu)^2}{2}}}\left((e^{-\frac{(x_i-\mu)^2}{2}}-e^{-\frac{(x_i+\mu)^2}{2}})x_i-(e^{-\frac{(x_i-\mu)^2}{2}}+e^{-\frac{(x_i+\mu)^2}{2}})\mu\right)\right]\end{aligned}$$

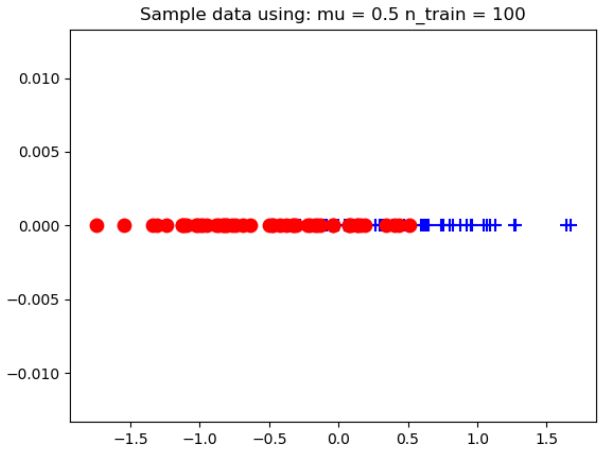
$$
\begin{aligned}
=&\frac{1}{n}\sum_{i=1}^{n}\left[\frac{e^{-\frac{(x_i-\mu)^2}{2}}-e^{-\frac{(x_i+\mu)^2}{2}}}{e^{-\frac{(x_i-\mu)^2}{2}}+e^{-\frac{(x_i+\mu)^2}{2}}}x_i\frac{e^{-\frac{(x_i-\mu)^2}{2}}+e^{-\frac{(x_i+\mu)^2}{2}}}{e^{-\frac{(x_i-\mu)^2}{2}}+e^{-\frac{(x_i+\mu)^2}{2}}}\mu\right]\\
=&\frac{1}{n}\sum_{i=1}^{n}\left[\frac{2e^{-\frac{(x_i-\mu)^2}{2}}-e^{-\frac{(x_i-\mu)^2}{2}}-e^{-\frac{(x_i+\mu)^2}{2}}}{e^{-\frac{(x_i-\mu)^2}{2}}+e^{-\frac{(x_i+\mu)^2}{2}}}x_i-\mu\right]\\
=&\frac{1}{n}\sum_{i=1}^{n}\left[\left(2\frac{e^{-\frac{(x_i-\mu)^2}{2}}}{e^{-\frac{(x_i-\mu)^2}{2}}+e^{-\frac{(x_i+\mu)^2}{2}}}-1\right)x_i\right]-\mu\\
=&\left[\frac{1}{n}\sum_{i=1}^{n}(2w_i(\mu)-1)x_i\right]-\mu
\end{aligned}
$$

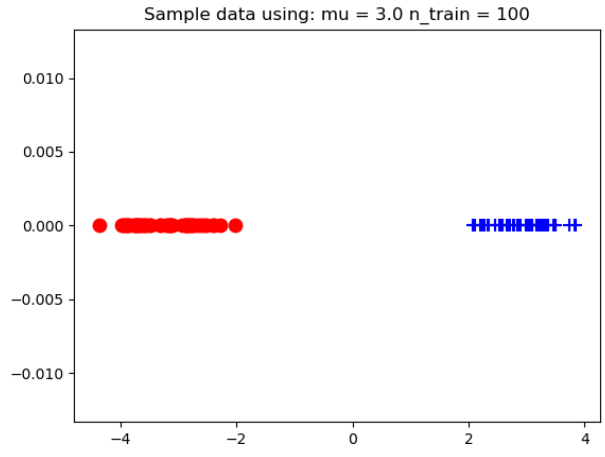
Now we write out the update formula for gradient descent on μ:

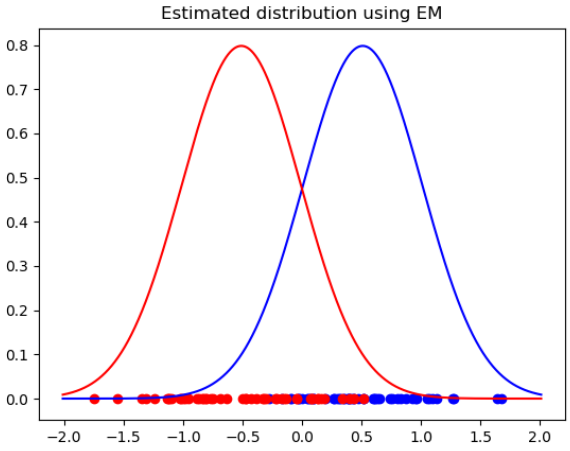
$$
\begin{aligned}
\mu_{t+1}^{\mathbf{GA}}&=\mu_t^{\mathbf{GA}}+\alpha\frac{d}{d\mu}l(\{x_i\}_{i=1}^n;\mu)\Big|_{\mu=\mu_t^{\mathbf{GA}}}\\
&=\mu_t^{\mathbf{GA}}+\alpha\left[\frac{1}{n}\sum_{i=1}^{n}(2w_i(\mu_t^{\mathbf{GA}})-1)x_i\right]-\alpha\mu_t^{\mathbf{GA}}\\
&=(1-\alpha)\mu_t^{\mathbf{GA}}+\alpha\left[\frac{1}{n}\sum_{i=1}^{n}(2w_i(\mu_t^{\mathbf{GA}})-1)x_i\right]
\end{aligned}
$$

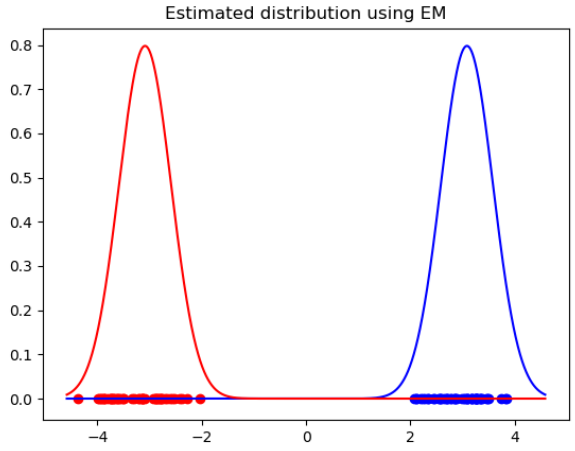
(m) Comment on the similarity or dissimilarity between the EM and gradient ascent (GA) updates derived in the previous two parts. You are given a code `em_gd_km.py` to run the two algorithms for the simpler one-dimensional mixture with a single unknown parameter μ. Set part_m=True in the code and run it. The code first generates a dataset of size 100 for two cases $\mu_{\mathbf{true}} = 0.5$ (i.e., when the two mixtures are close) and the case $\mu_{\mathbf{true}} = 3$ (i.e., when the two mixtures are far). We also plot the labeled dataset to help you visualize (but note that labels are not available to estimate $\mu_{\mathbf{true}}$ and hence we use EM and GA to obtain estimates). Starting at $\mu_0 = 0.1$, the code then computes EM updates and GA updates with step size 0.05 for the dataset. Comment on the convergence plots for the two algorithms. Do the observations match well with the similarity/dissimilarity observed in the updates derived in the previous parts?

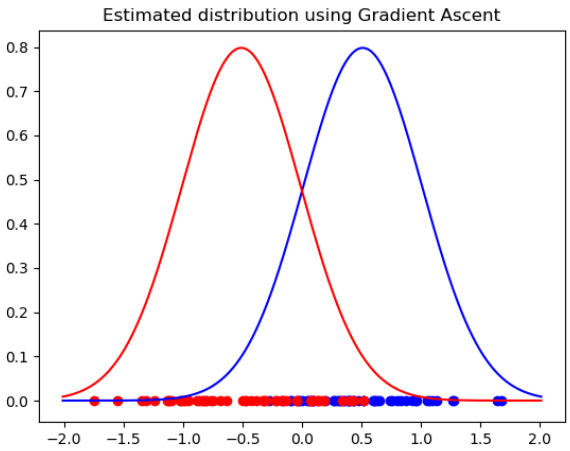
Similarity: They both look for the maximum likelihood point.
Difference: EM break it down to two steps $\vec{\theta}$ and q_i; while Gradient descent synthesize them into one gradient.
Both EM and GD converge to the same optimal point which is the maximum likelihood point.
EM converges with a straight vertical line down. This is because EM optimize $\vec{\theta}$ and q_i separately without any step size. It allows either parameter to move as far as the gradient allows. This will result in big improvements for the first few steps and form a broken line which can be seen on log scale.
GD converges with a shallower slope and bounces back and force around the optimal point.
The observation is consistent with our expectation for the similarity and dissimilarity between EM and GD.

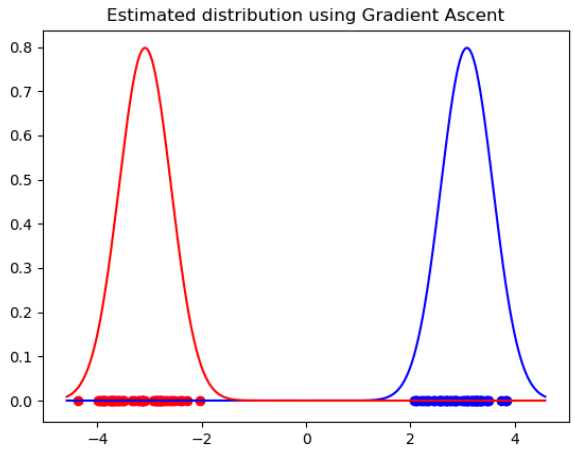


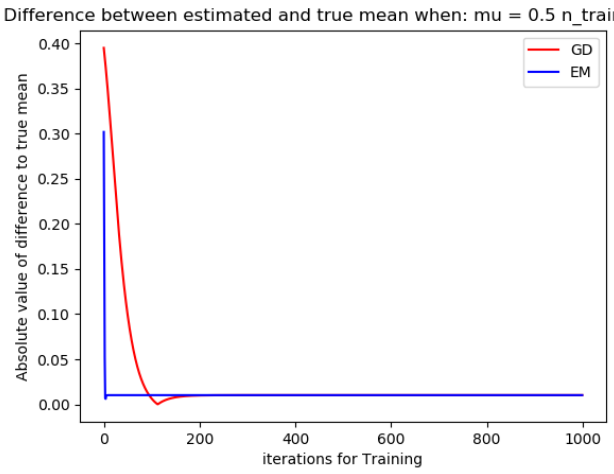


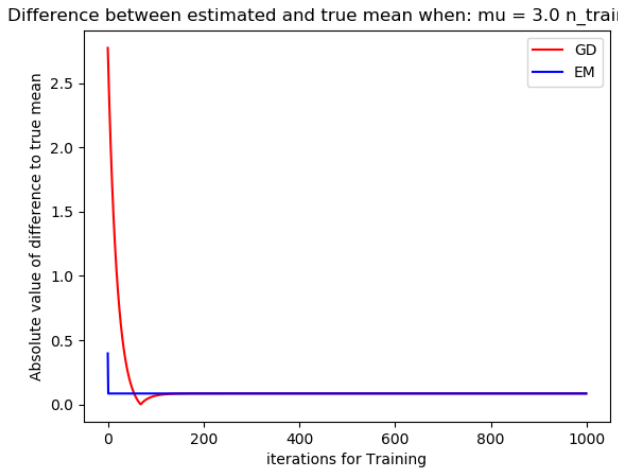

Estimated distribution using EM

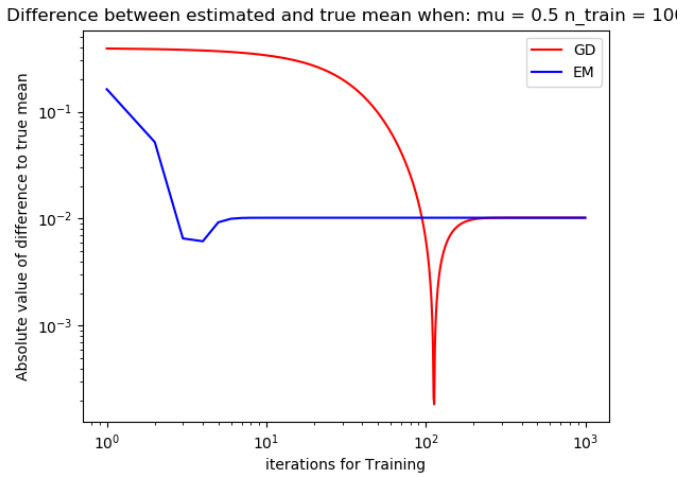

Estimated distribution using EM

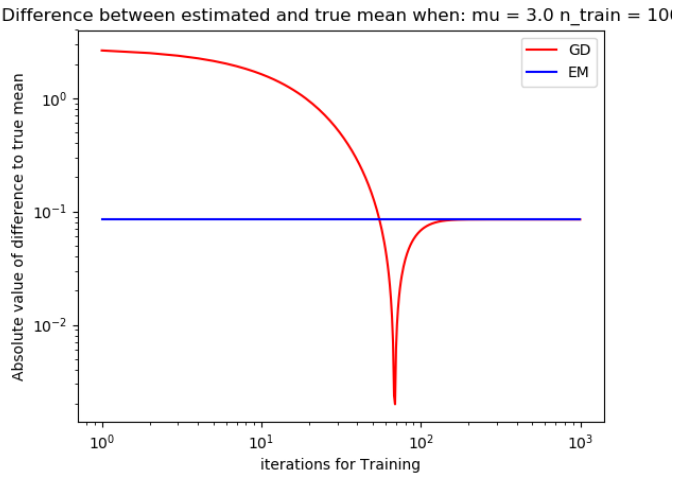

Estimated distribution using Gradient Ascent


Estimated distribution using Gradient Ascent


Difference between estimated and true mean when: mu = 0.5 n_train = 10
GD
EM
Absolute value of difference to true mean
iterations for Training


Difference between estimated and true mean when: mu = 3.0 n_train = 10
GD
EM
Absolute value of difference to true mean
iterations for Training





---

n_points: 100 , True mean:0.500, GA (final) estimate:0.510, EM (final) estimate:0.510

---

n_points: 100 , True mean:3.000, GA (final) estimate:3.085, EM (final) estimate:3.085

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm


def generate_data(mu_1, mu_2, n_examples, sigma_1=0.5, sigma_2=0.5):
# Generate sample data points from 2 distributions: (mu_1, sigma_1) and (mu_2, sigma_2)
d_1 = np.random.normal(mu_1, sigma_1, n_examples)
d_2 = np.random.normal(mu_2, sigma_2, n_examples)
x = np.concatenate([d_1, d_2])
return d_1, d_2, x


def plot_data(d_1, d_2):
# Plot scatter plot of data samples, labeled by class
plt.figure()
plt.scatter(d_1, np.zeros(len(d_1)), c='b', s=80., marker='+')
plt.scatter(d_2, np.zeros(len(d_2)), c='r', s=80.)
plt.title("Sample data using: mu = " + str(mu) + " n_train = " + str(len(d_1) + len(d_2)))
#plt.show()
plt.savefig(whichPart + 'data.png')
plt.close()
return


def plot_data_and_distr(d_1, d_2, mu_1, mu_2, sigma_1=0.5, sigma_2=0.5, title=""):
# Plot scatter plot of data samples overlayed with distribution of estimated means: mu_1 and mu_2
plt.scatter(d_1, np.zeros(len(d_1)), c='b')
scale = [min(mu_1 - 3 * sigma_1, mu_2 - 3 * sigma_2), max(mu_1 + 3 * sigma_1, mu_2 + 3 * sigma_2)]
plt.scatter(d_2, np.zeros(len(d_2)), c='r')
x_axis = np.arange(scale[0], scale[1], 0.001)
plt.plot(x_axis, norm.pdf(x_axis, mu_1, sigma_1), c='b')
x_axis = np.arange(scale[0], scale[1], 0.001)
plt.plot(x_axis, norm.pdf(x_axis, mu_2, sigma_2), c='r')
plt.title(title)
```

```
#plt.show()
plt.savefig(whichPart + title.replace(' ', '_') + '.png')
plt.close()


def grad_ascent(x, mu, mu_true, sigma=0.5, iterations=1000):
# Run gradient ascent on the likelihood of a point belonging to a class and compare the estimates of the
    mean
#    at each iteration with the true mean of the distribution
# Note: the original dataset comes from 2 distributions centered at mu and -mu, which this takes into
    account
#    with each update
diff_mu = []
alpha = 0.05
for i in range(iterations):
phi_1 = np.exp(-np.square(x - mu) / (2 * np.square(sigma)))
phi_2 = np.exp(-np.square(x + mu) / (2 * np.square(sigma)))
w = phi_1 / (phi_1 + phi_2)
em = (1 / len(x)) * np.sum((2 * w - 1) * x)
mu = mu * (1 - alpha) + alpha * em
diff_mu.append(np.abs(mu - mu_true))
return mu, sigma, diff_mu


def em(x, mu, mu_true, sigma=0.5, iterations=1000):
# Run the EM algorithm to find the estimated mean of the dataset
# Note: the original dataset comes from 2 distributions centered at mu and -mu, which this takes into
    account
#    with each update
diff_mu = np.zeros(iterations)
for i in range(iterations):
phi_1 = np.exp(-np.square(x - mu) / (2 * np.square(sigma)))
phi_2 = np.exp(-np.square(x + mu) / (2 * np.square(sigma)))
w = phi_1 / (phi_1 + phi_2)

mu = (1 / len(x)) * np.sum((2 * w - 1) * x)
diff_mu[i] = np.abs(mu - mu_true)
return mu, sigma, diff_mu


def kmeans(x, mu, mu_true, sigma=0.5, iterations=1000):
# Run the K means algorithm to find the estimated mean of the dataset
# Note: the original dataset comes from 2 distributions centered at mu and -mu, which this takes into
    account
#    with each update
diff_mu = np.zeros(iterations)

for i in range(iterations):
mu_1 = mu
mu_2 = -mu
set1 = []
set2 = []
for x_i in x:
if np.abs(x_i - mu_1) <= np.abs(x_i - mu_2):
set1.append(x_i)
else:
set2.append(x_i)
mu_1_new = np.mean(set1)
mu_2_new = np.mean(set2)
# Estimates two means and combines them to get mu for the next iteration
mu = np.abs(mu_1_new - mu_2_new) / 2
diff_mu[i] = np.abs(mu - mu_true)
return mu, sigma, diff_mu
```

```
def plot_differences(iterations, diff_mu_ga, diff_mu_em, mu, n_examples):
# Make plot comparing convergence of means to true mean for gradient descent and EM
plt.plot(np.arange(iterations), diff_mu_ga, c='r', label='GD')
plt.plot(np.arange(iterations), diff_mu_em, c='b', label='EM')
plt.legend()
plt.title("Difference between estimated and true mean when: mu = " + str(mu) + " n_train = " +
    str(n_examples))
plt.xlabel("iterations for Training")
plt.ylabel("Absolute value of difference to true mean")
# plt.show()
plt.savefig(whichPart + 'difference.png')
plt.close()


def plot_ll(x, scale=[-5, 5]):
# if you want to visualize the likelihood function as a function of mu
mus = np.linspace(scale[0], scale[1], 200)
ll = np.zeros(mus.shape)
for j, mu in enumerate(mus):
ll[j] = 0.
for xi in x:
ll[j] += np.log(np.exp(-(xi - mu) ** 2 / 2) + np.exp(-(xi + mu) ** 2 / 2)) - np.log(2 * np.sqrt(2 * np.pi))
plt.plot(mus, ll)
plt.show()
return


################################################################################
part_m = True
part_n = 1 - part_m

# Set this to True if you want to visualize the distribution estimated by each method
visualize_distr = True
################################################################################

np.random.seed(12312)
mu_list = [.5, 3.]
n_train_list = [50]
mu_start = 0.1
whichPart = ''

for mu in mu_list:
for n_train in n_train_list:
d_1, d_2, x = generate_data(mu, -mu, n_examples=n_train)
if part_m:
whichPart = 'Figure_3m-mu' + str(mu) + '_ntrain' + str(n_train) + '_'
if part_n:
whichPart = 'Figure_3n-mu' + str(mu) + '_ntrain' + str(n_train) + '_'
plot_data(d_1, d_2)

# Code for part m
if part_m:
mu_ga, sigma_gd, diff_mu_ga = grad_ascent(x, mu_start, mu)
mu_em, sigma_em, diff_mu_em = em(x, mu_start, mu)
if visualize_distr:
plot_data_and_distr(d_1, d_2, mu_ga, -mu_ga, title="Estimated distribution using Gradient Ascent")
plot_data_and_distr(d_1, d_2, mu_em, -mu_em, title="Estimated distribution using EM")
print("------------")
print("n_points:", n_train * 2,
", True mean:{:.3f}, GA (final) estimate:{:.3f}, EM (final) estimate:{:.3f}".format(mu, mu_ga,
mu_em)) # , "GA (final) mean: {.3f}", mu_ga, "EM (final) mean: {.3f}", mu_em)
```

```
print("------------")
plot_differences(1000, diff_mu_ga, diff_mu_em, mu, 2 * n_train)

# Code for part n
if part_n:
mu_ga, sigma_gd, diff_mu_ga = grad_ascent(x, mu_start, mu)
mu_em, sigma_em, diff_mu_em = em(x, mu_start, mu)
mu_k, sigma_k, diff_mu_k = kmeans(x, mu_start, mu)
if visualize_distr:
plot_data_and_distr(d_1, d_2, mu_ga, -mu_ga, title="Estimated distribution using Gradient Ascent")
plot_data_and_distr(d_1, d_2, mu_em, -mu_em, title="Estimated distribution using EM")
plot_data_and_distr(d_1, d_2, mu_k, -mu_k, title="Estimated distribution using K Means")
print("------------")
print(
"True mean:{:.3f}, GA (final) estimate:{:.3f}, EM (final) estimate:{:.3f}, K-Means (final)
    estimate:{:.3f}".format(
mu, mu_ga, mu_em, mu_k))
print("------------")
```

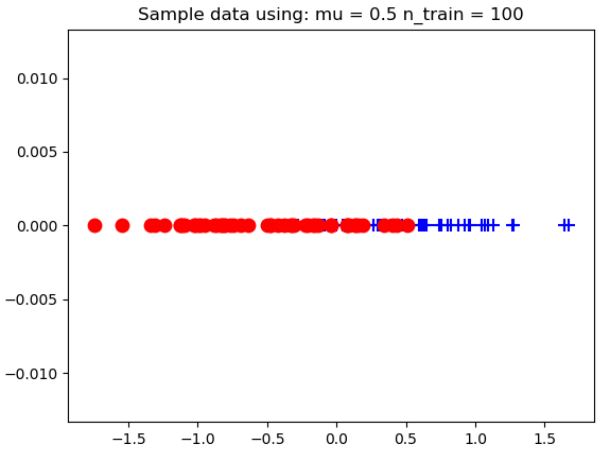
(n) Suppose we decided to use the simplest algorithm to estimate the parameter μ: K Means! Because the parameter is shared, we can estimate $\mu = \frac{1}{2}|\hat{\mu}_1 - \hat{\mu}_2|$ (assuming $\mu > 0$) where $\hat{\mu}_1$ and $\hat{\mu}_2$ denote the cluster centroids determined by the K means. Do you think this strategy will work well? Does your answer depend on whether μ is large or small? To help you answer the question, we have given a numerical implementation for this part as well. Run the code `em_gd_km.py` with part_n=True. The code then plots a few data-points where we also plot the hidden labels to help you understand the dataset. Also code provides the final estimates of μ by EM, Gradient Ascent (GA) and K Means. Use the plots and the final answers to provide an intuitive argument for the questions asked above in this part. Do not spend time on being mathematically rigorous.

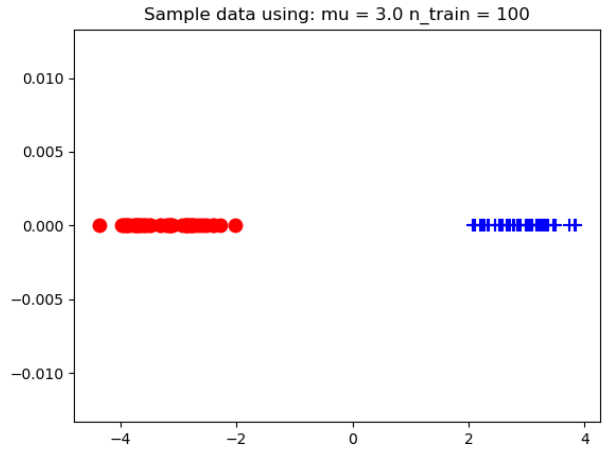
Hopefully you are able to learn the following take away messages: For the simple one-dimensional mixture model, we have that

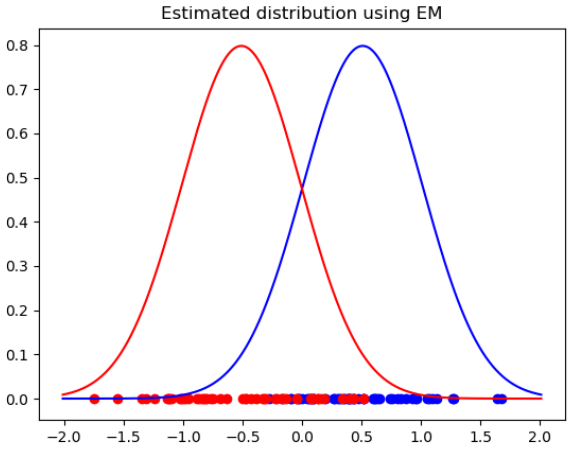
- EM works well: It converges to a good estimate of μ pretty quickly.
- Gradient ascent is a weighted version of EM: It converges to a good estimate of μ, but is slower than EM.
- K Means: Because of the hard thresholding, it converges to a biased estimate of μ if the two distributions overlap.

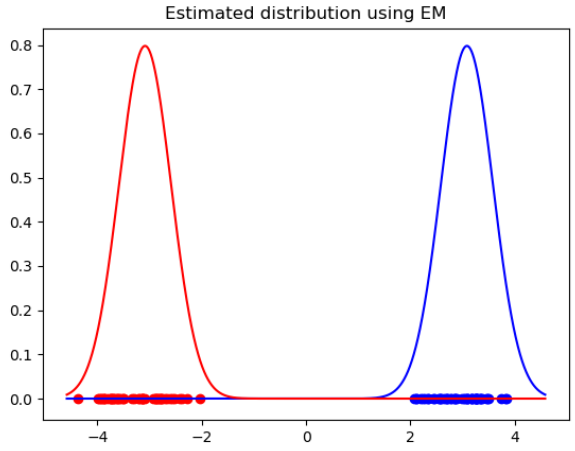
I have learned the take away messages:For the simple one-dimensional mixture model, we have that

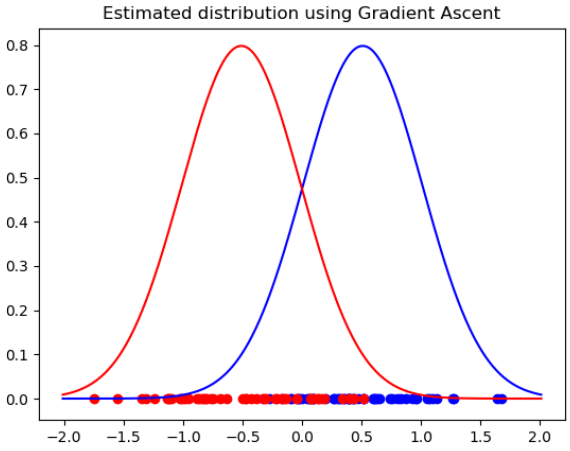
- EM works well: It converges to a good estimate of μ pretty quickly.
- Gradient ascent is a weighted version of EM: It converges to a good estimate of μ, but is slower than EM.
- K Means: Because of the hard thresholding, it converges to a biased estimate of μ if the two distributions overlap.

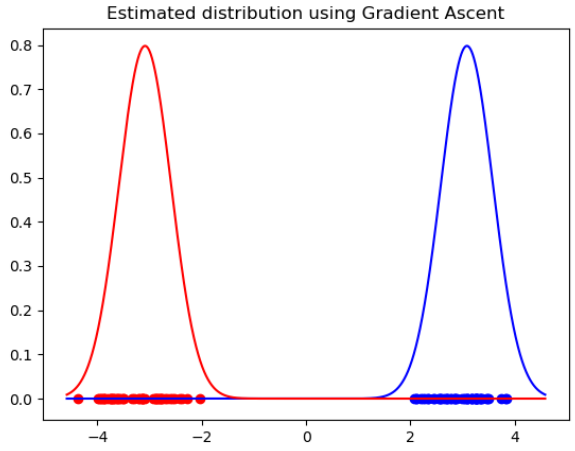


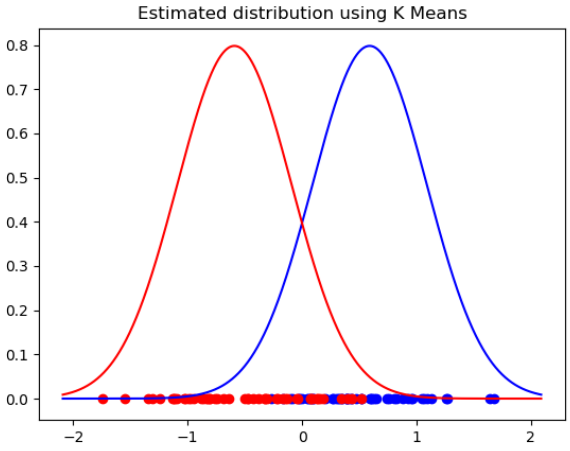


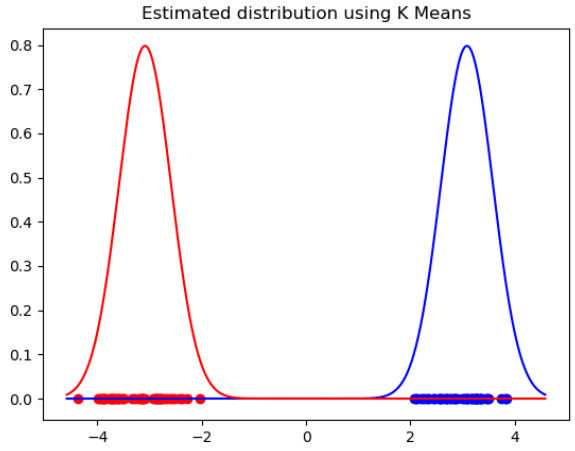












True mean:0.500, GA (final) estimate:0.510, EM (final) estimate:0.510, K-Means (final) estimate:0.591

True mean:3.000, GA (final) estimate:3.085, EM (final) estimate:3.085, K-Means (final) estimate:3.085

Question 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 $(\vec{X}, Z)$ where $\vec{X}$ takes values in $\mathbb{R}^d$ and Z takes value in the set $[K] = \{1, \ldots, K\}$. The joint-distribution of the pair $(\vec{X}, Z)$ is given to us as follows:

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

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

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

Suppose we are given a dataset $\{\vec{x}_i\}_{i=1}^n$ without their labels. Our goal is to identify the K-clusters of the data. To do this, we are going to estimate the parameters $\vec{\mu}_k, \mathbf{\Sigma}_k$ 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:

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

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

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

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:

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

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

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

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

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

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

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

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

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

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

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

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)
```

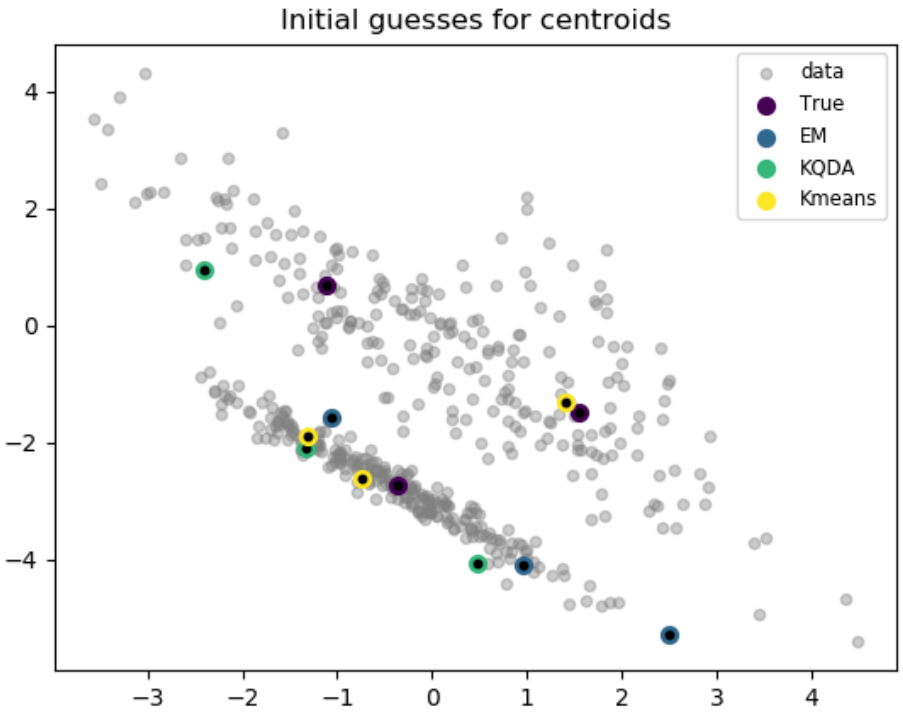
We assume that $\vec{x} \in \mathbb{R}^2$, and the default settings are number of samples is 500 ($n = 500$), and the number of clusters is 3 ($K = 3$). Notice that `seed` will determine the randomness and `factor` will determine how far apart are the clusters.

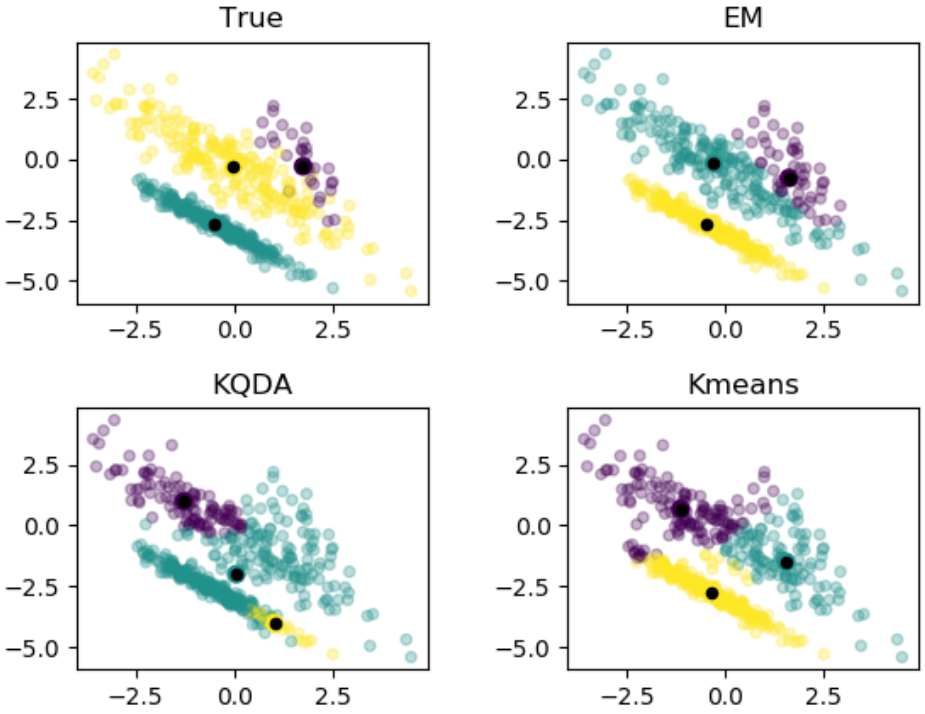
(a) Run the following setting:

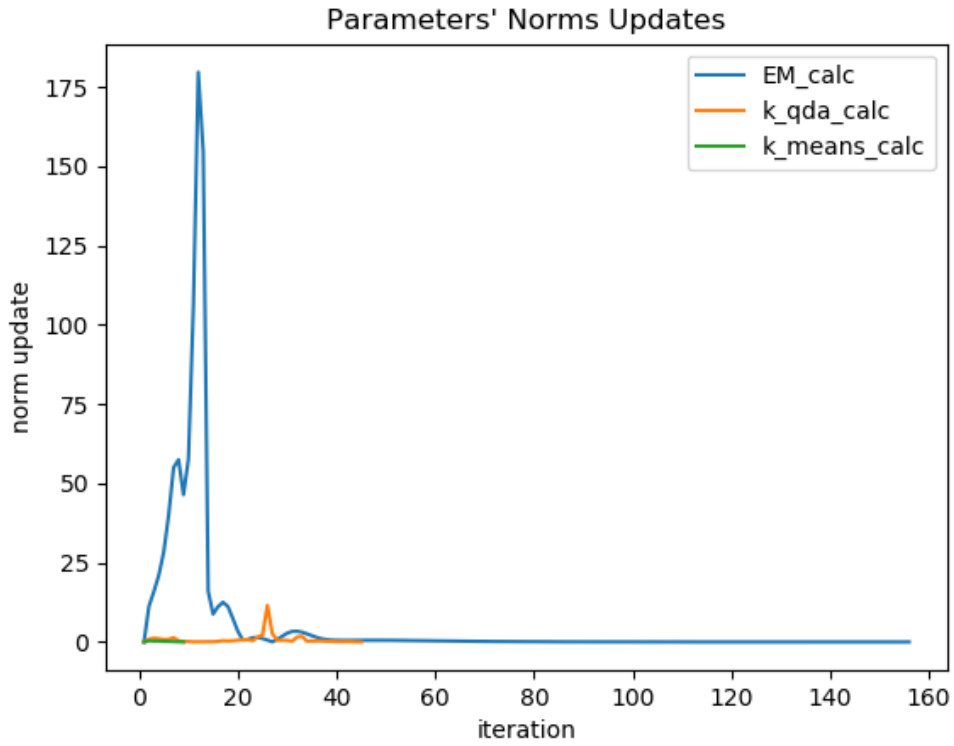
```
experiments(seed=11, factor=1, num_samples=500, num_clusters=3)
```

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

The difference between the initial guesses and the final predictions is largest in EM AND closest in K-means.
EM is not very sensitive to the initial guesses because it will look for the maximum likelihood point anyway. On the other hand, K-QDA and K-means tend to cut the data into pieces using the initial guesses.
However, K-means converges the fastest and EM is the slowest.
The final prediction of EM is the best; K-means and K-QDA are equally bad.
This is because EM has more parameters to allow for uneven and overlapping distributions.


Initial guesses for centroids
data
True
EM
KQDA
Kmeans
4
2
0
−2
−4
−3
−2
−1
0
1
2
3
4


True
EM
KQDA
Kmeans
2.5
0.0
−2.5
−5.0
−2.5
0.0
2.5



Below is the printed results of updates:

EM_calc
iteration 1, update 0.0002
iteration 2, update 11.221832782345018
iteration 3, update 15.895028006874554
iteration 4, update 21.016630288918122
iteration 5, update 28.359858328967448
iteration 6, update 40.00662487046384
iteration 7, update 55.00513458715068
iteration 8, update 57.4749240715264
iteration 9, update 46.44690175881692
iteration 10, update 57.612594671861416
iteration 11, update 105.52357138695561
iteration 12, update 179.64358613881802
iteration 13, update 154.4558020125114
iteration 14, update 15.993699598010949
iteration 15, update 8.798845255353513
iteration 16, update 11.134736782698042
iteration 17, update 12.517797342970198
iteration 18, update 11.181050144282835
iteration 19, update 7.408627432780122
iteration 20, update 3.4195379060790856
iteration 21, update 0.6500952485214384
iteration 22, update 0.7911550419216837

```
iteration 23, update 1.3497117646018069
iteration 24, update 1.3884709058149838
iteration 25, update 1.095589588690018
iteration 26, update 0.5961764094877253
iteration 27, update 0.03085253676931643
iteration 28, update 0.8170462505020168
iteration 29, update 1.8045244094141708
iteration 30, update 2.7697476971384276
iteration 31, update 3.3229256653798984
iteration 32, update 3.379259573431341
iteration 33, update 3.0830976043414466
iteration 34, update 2.567166201080113
iteration 35, update 1.9679244249673502
iteration 36, update 1.4222823915617937
iteration 37, update 1.00963101169134
iteration 38, update 0.7399354610487308
iteration 39, update 0.5853259319694644
iteration 40, update 0.5097530562185284
iteration 41, update 0.4826893643922858
iteration 42, update 0.48230916945840363
iteration 43, update 0.49443644209122795
iteration 44, update 0.5104878782793776
iteration 45, update 0.525608044986825
iteration 46, update 0.5372869342609192
iteration 47, update 0.5444272833648256
iteration 48, update 0.5467491971563732
iteration 49, update 0.5444244203832795
iteration 50, update 0.5378585612405686
iteration 51, update 0.5275650182459231
iteration 52, update 0.5140939056279876
iteration 53, update 0.49799291042359073
iteration 54, update 0.4797860795814586
iteration 55, update 0.4599623680113609
iteration 56, update 0.4389694157055146
iteration 57, update 0.4172102022630497
iteration 58, update 0.39504146041531385
iteration 59, update 0.37277336772399394
iteration 60, update 0.35067031683263394
iteration 61, update 0.32895264951321224
iteration 62, update 0.307799232439379
iteration 63, update 0.2873507157919448
iteration 64, update 0.2677132833031237
iteration 65, update 0.24896268921304454
iteration 66, update 0.2311483854615517
iteration 67, update 0.21429756776569775
iteration 68, update 0.19841900481981156
iteration 69, update 0.18350655433528118
```

iteration 70, update 0.1695423073414304
iteration 71, update 0.15649933500037605
iteration 72, update 0.14434403803943496
iteration 73, update 0.13303811761807083
iteration 74, update 0.1225401987285295
iteration 75, update 0.11280714380620793
iteration 76, update 0.10379509681024501
iteration 77, update 0.09546029747036755
iteration 78, update 0.08775970283954848
iteration 79, update 0.0806514499623745
iteration 80, update 0.07409518898668921
iteration 81, update 0.06805231234943676
iteration 82, update 0.06248610127840948
iteration 83, update 0.05736180761596188
iteration 84, update 0.05264668549114049
iteration 85, update 0.048309984899447045
iteration 86, update 0.04432291667035315
iteration 87, update 0.04065859662250659
iteration 88, update 0.037291974920890425
iteration 89, update 0.03419975546921705
iteration 90, update 0.03136030901646336
iteration 91, update 0.02875358295091246
iteration 92, update 0.026361009929246393
iteration 93, update 0.02416541701688857
iteration 94, update 0.022150936561274648
iteration 95, update 0.0203029197862179
iteration 96, update 0.018607853585422163
iteration 97, update 0.017053281075618543
iteration 98, update 0.015627726091452132
iteration 99, update 0.014320621812089485
iteration 100, update 0.013122243485781837
iteration 101, update 0.01202364531764033
iteration 102, update 0.011016601352821453
iteration 103, update 0.010093550277474606
iteration 104, update 0.009247543991477869
iteration 105, update 0.008472199746847764
iteration 106, update 0.007761655720628369
iteration 107, update 0.007110529789770226
iteration 108, update 0.0065138813674820994
iteration 109, update 0.005967176089825443
iteration 110, update 0.005466253162467183
iteration 111, update 0.0050072952171831275
iteration 112, update 0.004586800490415044
iteration 113, update 0.004201557173701076
iteration 114, update 0.003848619764767136
iteration 115, update 0.003525287314232628
iteration 116, update 0.0032290833419210685

iteration 117, update 0.0029577374328937367
iteration 118, update 0.00270916826480061
iteration 119, update 0.002481467983784569
iteration 120, update 0.002272887901199283
iteration 121, update 0.0020818253201468906
iteration 122, update 0.001906811408844078
iteration 123, update 0.0017465001249092893
iteration 124, update 0.0015996580020782858
iteration 125, update 0.0014651547720632152
iteration 126, update 0.0013419547946114108
iteration 127, update 0.0012291091723000136
iteration 128, update 0.0011257484873112844
iteration 129, update 0.001031076212939297
iteration 130, update 0.0009443625792755483
iteration 131, update 0.0008649390423443037
iteration 132, update 0.0007921931166947616
iteration 133, update 0.0007255637195839881
iteration 134, update 0.0006645368606541524
iteration 135, update 0.0006086416889274915
iteration 136, update 0.0005574468918894127
iteration 137, update 0.0005105573638957139
iteration 138, update 0.000467611202907392
iteration 139, update 0.00042827688582747214
iteration 140, update 0.0003922507562492683
iteration 141, update 0.0003592546613617742
iteration 142, update 0.00032903383282700815
iteration 143, update 0.0003013548947592426
iteration 144, update 0.0002760041069223007
iteration 145, update 0.0002527856762526426
iteration 146, update 0.00023152028097683797
iteration 147, update 0.00021204366294114152
iteration 148, update 0.0001942053931998089
iteration 149, update 0.00017786766443350643
iteration 150, update 0.0001629042772037792
iteration 151, update 0.00014919963041393203
iteration 152, update 0.00013664784967204469
iteration 153, update 0.0001251519678362456
iteration 154, update 0.0001146231611528492
iteration 155, update 0.00010498008862214192
iteration 156, update 9.614824045911519e-05

k_qda_calc
iteration 1, update 2e-05
iteration 2, update 0.868668229713116
iteration 3, update 1.187621311151393
iteration 4, update 1.0490700628561325
iteration 5, update 0.7740484418989847

iteration 6, update 0.8402336989655679
iteration 7, update 1.3662413000416103
iteration 8, update 0.38328270434379186
iteration 9, update 0.11360650979227344
iteration 10, update 0.15356953613001426
iteration 11, update 0.03235186754648143
iteration 12, update 0.0761484541653795
iteration 13, update 0.050727064736127425
iteration 14, update 0.10323551453662327
iteration 15, update 0.0794871794588851
iteration 16, update 0.1891854905163854
iteration 17, update 0.4133069994833283
iteration 18, update 0.28642468135443555
iteration 19, update 0.3824576289258276
iteration 20, update 0.5500149965428822
iteration 21, update 0.7171421489654106
iteration 22, update 0.7435076676468257
iteration 23, update 0.38629802001472613
iteration 24, update 1.5139034578126345
iteration 25, update 2.161598610654874
iteration 26, update 11.53441555538621
iteration 27, update 2.614227144037055
iteration 28, update 0.5217674979961711
iteration 29, update 0.512615995998146
iteration 30, update 0.5003472649054552
iteration 31, update 0.22831211435370757
iteration 32, update 1.5478535126286563
iteration 33, update 1.7574690856577264
iteration 34, update 0.2666634943687821
iteration 35, update 0.29110053033568895
iteration 36, update 0.32612071679657995
iteration 37, update 0.34105053235693406
iteration 38, update 0.23966767782618098
iteration 39, update 0.20660385298304274
iteration 40, update 0.11922883832367386
iteration 41, update 0.10700447959212347
iteration 42, update 0.12331632427533137
iteration 43, update 0.13122684085570105
iteration 44, update 0.02670664321016654
iteration 45, update 0.0

k_means_calc
iteration 1, update 2e-05
iteration 2, update 0.3830443376421983
iteration 3, update 0.2673453932623169
iteration 4, update 0.2910058665662966
iteration 5, update 0.19894005591745575

iteration 6, update 0.1739005429189131
iteration 7, update 0.11948187431415665
iteration 8, update 0.04418597867214031
iteration 9, update 0.0

Here is my modified code:

```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
import os
import inspect

# This function generate random mean and covariance
def gauss_params_gen(num_clusters, num_dims, factor):
mu = np.random.randn(num_clusters, num_dims) * factor
sigma = np.random.randn(num_clusters, num_dims, num_dims)
for k in range(num_clusters):
sigma[k] = np.dot(sigma[k], sigma[k].T)

return (mu, sigma)


# Given mean and covariance generate data
def data_gen(mu, sigma, num_clusters, num_samples):
labels = []
X = []
cluster_prob = np.array([np.random.rand() for k in range(num_clusters)])
cluster_num_samples = (num_samples * cluster_prob / sum(cluster_prob)).astype(int)
cluster_num_samples[-1] = num_samples - sum(cluster_num_samples[:-1])

for k, ks in enumerate(cluster_num_samples):
labels.append([k] * ks)
X.append(np.random.multivariate_normal(mu[k], sigma[k], ks))

# shuffle data
randomize = np.arange(num_samples)
np.random.shuffle(randomize)
X = np.vstack(X)[randomize]
labels = np.array(sum(labels, []))[randomize]

return X, labels


def data2D_plot(ax, x, labels, centers, cmap, title):
data = {'x0': x[:, 0], 'x1': x[:, 1], 'label': labels}
ax.scatter(data['x0'], data['x1'], c=data['label'], cmap=cmap, s=20, alpha=0.3)
ax.scatter(centers[:, 0], centers[:, 1], c=np.arange(np.shape(centers)[0]), cmap=cmap, s=50, alpha=1)
ax.scatter(centers[:, 0], centers[:, 1], c='black', cmap=cmap, s=20, alpha=1)
ax.title.set_text(title)


def plot_init_means(x, mus, algs, fname):
import matplotlib.cm as cm
fig = plt.figure()
plt.scatter(x[:, 0], x[:, 1], c='gray', cmap='viridis', s=20, alpha=0.4, label='data')
for mu, alg, clr in zip(mus, algs, cm.viridis(np.linspace(0, 1, len(mus)))):
plt.scatter(mu[:, 0], mu[:, 1], c=clr, s=50, label=alg)
plt.scatter(mu[:, 0], mu[:, 1], c='black', s=10, alpha=1)
legend = plt.legend(loc='upper right', fontsize='small')
plt.title('Initial guesses for centroids')
fig.savefig(fname)
```

```
def loss_plot(loss, title, xlabel, ylabel, fname):
fig = plt.figure(figsize=(13, 6))
plt.plot(np.array(loss))
plt.title(title)
plt.xlabel(xlabel)
plt.ylabel(ylabel)
fig.savefig(fname)


def gaussian_pdf(X, mu, sigma):
# Gaussian probability density function
return np.linalg.det(sigma) ** -.5 ** (2 * np.pi) ** (-X.shape[1] / 2.) \
* np.exp(-.5 * np.einsum('ij, ij -> i', \
X - mu, np.dot(np.linalg.inv(sigma), (X - mu).T).T))


def EM_initial_guess(num_dims, data, num_samples, num_clusters):
# randomly choose the starting centroids/means
# as num_clusters of the points from datasets
mu = data[np.random.choice(num_samples, num_clusters, False), :]

# initialize the covariance matrice for each gaussian
sigma = [np.eye(num_dims)] * num_clusters

# initialize the probabilities/weights for each gaussian
# begin with equal weight for each gaussian
alpha = [1. / num_clusters] * num_clusters

return mu, sigma, alpha


def EM_E_step(num_clusters, num_samples, data, mu, sigma, alpha):
## Vectorized implementation of e-step equation to calculate the
## membership for each of k -gaussians
Q = np.zeros((num_samples, num_clusters))
for k in range(num_clusters):
Q[:, k] = alpha[k] * gaussian_pdf(data, mu[k], sigma[k])

## Normalize so that the responsibility matrix is row stochastic
Q = (Q.T / np.sum(Q, axis=1)).T

return Q


def EM_M_step(num_clusters, num_dims, num_samples, Q, data):
# M Step
## calculate the new mean and covariance for each gaussian by
## utilizing the new responsibilities
mu = np.zeros((num_clusters, num_dims))
sigma = np.zeros((num_clusters, num_dims, num_dims))
alpha = np.zeros(num_clusters)

## The number of datapoints belonging to each gaussian
num_samples_per_cluster = np.sum(Q, axis=0)

for k in range(num_clusters):
## means
mu[k] = 1. / num_samples_per_cluster[k] * np.sum(Q[:, k] * data.T, axis=1).T
centered_data = np.matrix(data - mu[k])

## covariances
```

```
sigma[k] = np.array(
1. / num_samples_per_cluster[k] * np.dot(np.multiply(centered_data.T, Q[:, k]), centered_data))

## and finally the probabilities
alpha[k] = 1. / (num_clusters * num_samples) * num_samples_per_cluster[k]

return mu, sigma, alpha


def EM_log_likelihood_calc(num_clusters, num_samples, data, mu, sigma, alpha):
L = np.zeros((num_samples, num_clusters))
for k in range(num_clusters):
L[:, k] = alpha[k] * gaussian_pdf(data, mu[k], sigma[k])
return np.sum(np.log(np.sum(L, axis=1)))


def EM_calc(num_dims, num_samples, num_clusters, x):
print(r'\hfill \linebreak')
print('\\textbf{{' + str(inspect.stack()[0][3]).replace('_', '\\_') + "}} \\\\")
log_likelihoods = []
labels = []
iter_cnt = 0
epsilon = 0.0001
max_iters = 200
update = 2 * epsilon

# initial guess
mu, sigma, alpha = EM_initial_guess(num_dims, x, num_samples, num_clusters)
mus = [mu]
sigmas = [sigma]
alphas = [alpha]
plot_iter_cnts = []
plot_updates = []

while (update > epsilon) and (iter_cnt < max_iters):
iter_cnt += 1

# E - Step
Q = EM_E_step(num_clusters, num_samples, x, mu, sigma, alpha)

# M - Step
mu, sigma, alpha = EM_M_step(num_clusters, num_dims, num_samples, Q, x)

mus.append(mu)
sigmas.append(sigma)
alphas.append(alpha)

# Likelihood computation
log_likelihoods.append(EM_log_likelihood_calc(num_clusters, num_samples, x, mu, sigma, alpha))

# check convergence
if iter_cnt >= 2:
update = np.abs(log_likelihoods[-1] - log_likelihoods[-2])

# logging
print("iteration {}, update {} \\\\".format(iter_cnt, update))

# print current iteration
labels.append(np.argmax(Q, axis=1))

plot_iter_cnts.append(iter_cnt)
plot_updates.append(update)
```

```
plt.plot(plot_iter_cnts, plot_updates, label=str(inspect.stack()[0][3]))
plt.title('Parameters\' Norms Updates')

return labels, log_likelihoods, {'mu': mus, 'sigma': sigmas, 'alpha': alphas}


def kmeans_initial_guess(data, num_samples, num_clusters):
# randomly choose the starting centroids/means
# as num_clusters of the points from datasets
mu = data[np.random.choice(num_samples, num_clusters, False), :]
return mu


def kmeans_get_labels(num_clusters, num_samples, num_dims, data, mu):
# set all dataset points to the best cluster according to minimal distance
# from centroid of each cluster
dist = np.zeros((num_clusters, num_samples))
for k in range(num_clusters):
dist[k] = np.linalg.norm(data - mu[k], axis=1)

labels = np.argmin(dist, axis=0)

return labels


def kmeans_get_means(num_clusters, num_dims, data, labels):
# Compute the new means given the reclustering of the data
mu = np.zeros((num_clusters, num_dims))
for k in range(num_clusters):
idx_list = np.where(labels == k)[0]
if (len(idx_list) == 0):
r = np.random.randint(len(data))
mu[k] = data[r, :]
else:
mu[k] = np.mean(data[idx_list], axis=0)
return mu


def kmeans_calc_loss(num_clusters, num_samples, data, mu, labels):
dist = np.zeros((num_samples, num_clusters))
for j in range(num_samples):
for k in range(num_clusters):
if (labels[j] == k):
dist[j, k] = np.linalg.norm(data[j] - mu[k])
return sum(sum(dist))


def k_means_calc(num_dims, num_samples, num_clusters, x):
print(r'\hfill \linebreak')
print('\\textbf{{' + str(inspect.stack()[0][3]).replace('_', '\\_') + "}} \\\\")

loss = []
labels = []
iter_cnt = 0
epsilon = 0.00001
max_iters = 100
update = 2 * epsilon

# initial guess
mu = [kmeans_initial_guess(x, num_samples, num_clusters)]
plot_iter_cnts = []
plot_updates = []
while (update > epsilon) and (iter_cnt < max_iters):
```

```
iter_cnt += 1
# Assign labels to each datapoint based on centroid
labels.append(kmeans_get_labels(num_clusters, num_samples, num_dims, x, mu[-1]))

# Assign centroid based on labels
mu.append(kmeans_get_means(num_clusters, num_dims, x, labels[-1]))
# check convergence
if iter_cnt >= 2:
update = np.linalg.norm(mu[-1] - mu[-2], None)

# Print distance to centroids vs iteration
loss.append(kmeans_calc_loss(num_clusters, num_samples, x, mu[-1], labels[-1]))

# logging
print("iteration {}, update {} \\\\".format(iter_cnt, update))
plot_iter_cnts.append(iter_cnt)
plot_updates.append(update)

plt.plot(plot_iter_cnts, plot_updates, label=str(inspect.stack()[0][3]))
plt.title('Parameters\' Norms Updates')

return labels, loss, mu


def k_qda_initial_guess(num_dims, data, num_samples, num_clusters):
# randomly choose the starting centroids/means
# as num_clusters of the points from datasets
mu = data[np.random.choice(num_samples, num_clusters, False), :]

# initialize the covariance matrice for each gaussian
sigma = [np.eye(num_dims)] * num_clusters

return mu, sigma


def k_qda_get_parms(num_clusters, num_dims, data, labels):
## calculate the new mean and covariance for each gaussian
mu = np.zeros((num_clusters, num_dims))
sigma = np.zeros((num_clusters, num_dims, num_dims))

for k in range(num_clusters):
c_k = labels == k
if (len(data[c_k]) == 0):
r = np.random.randint(len(data))
mu[k] = data[r, :]
else:
mu[k] = np.mean(data[c_k], axis=0)

if (len(data[c_k]) > 1):
centered_data = np.matrix(data[c_k] - mu[k])
sigma[k] = np.array(1. / len(data[c_k]) * np.dot(centered_data.T, centered_data))
else:
sigma[k] = np.eye(num_dims)

return mu, sigma


def k_qda_get_labels(num_clusters, num_samples, mu, sigma, data):
# set all dataset points to the best cluster according to best
# probability given calculated means and covariances
dist = np.zeros((num_clusters, num_samples))
for k in range(num_clusters):
data_center = (data - mu[k])
```

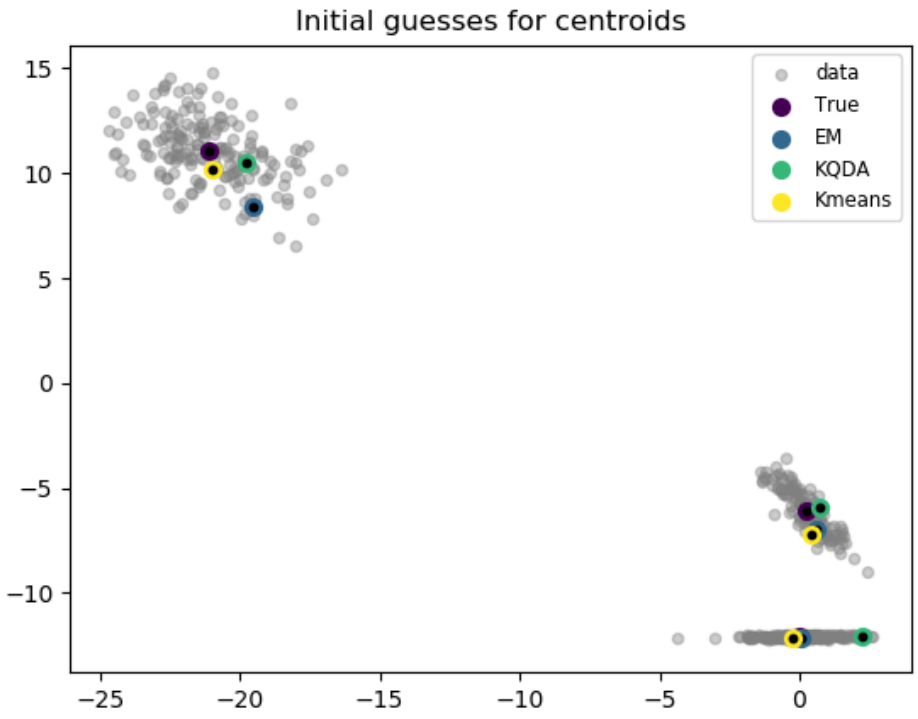
```
dist[k] = np.einsum('ij, ij -> i', data_center, np.dot(np.linalg.inv(sigma[k]), data_center.T).T)
labels = np.argmin(dist, axis=0)
return labels


def k_qda_calc(num_dims, num_samples, num_clusters, x):
print(r'\hfill \linebreak')
print('\\textbf{{' + str(inspect.stack()[0][3]).replace('_', '\\_') + "}} \\\\")
loss = []
labels = []
iter_cnt = 0
epsilon = 0.00001
max_iters = 100
update = 2 * epsilon

# initial guess
mu, sigma = k_qda_initial_guess(num_dims, x, num_samples, num_clusters)
mus = [mu]
sigmas = [sigma]
plot_iter_cnts = []
plot_updates = []
while (update > epsilon) and (iter_cnt < max_iters):
iter_cnt += 1

# Assign labels to each datapoint based on probability
labels.append(k_qda_get_labels(num_clusters, num_samples, mus[-1], sigmas[-1], x))

# Assign centroid and covarince based on labels
mu, sigma = k_qda_get_parms(num_clusters, num_dims, x, labels[-1])

mus.append(mu)
sigmas.append(sigma)

# check convergence
if iter_cnt >= 2:
update = np.linalg.norm(mus[-1] - mus[-2], None)
update += np.linalg.norm(sigmas[-1] - sigmas[-2], None)

# logging
print("iteration {}, update {} \\\\".format(iter_cnt, update))
plot_iter_cnts.append(iter_cnt)
plot_updates.append(update)

plt.plot(plot_iter_cnts, plot_updates, label=str(inspect.stack()[0][3]))
plt.xlabel('iteration')
plt.ylabel('norm update')
plt.title('Parameters\' Norms Updates')

return labels, {'mu': mus, 'sigma': sigmas}


def experiments(seed, factor, dir='plots', num_samples=500, num_clusters=3):
if not os.path.exists(dir):
os.makedirs(dir)

np.random.seed(seed)
num_dims = 2

# generate data samples
(mu, sigma) = gauss_params_gen(num_clusters, num_dims, factor)
x, true_labels = data_gen(mu, sigma, num_clusters, num_samples)

plt.figure()
```

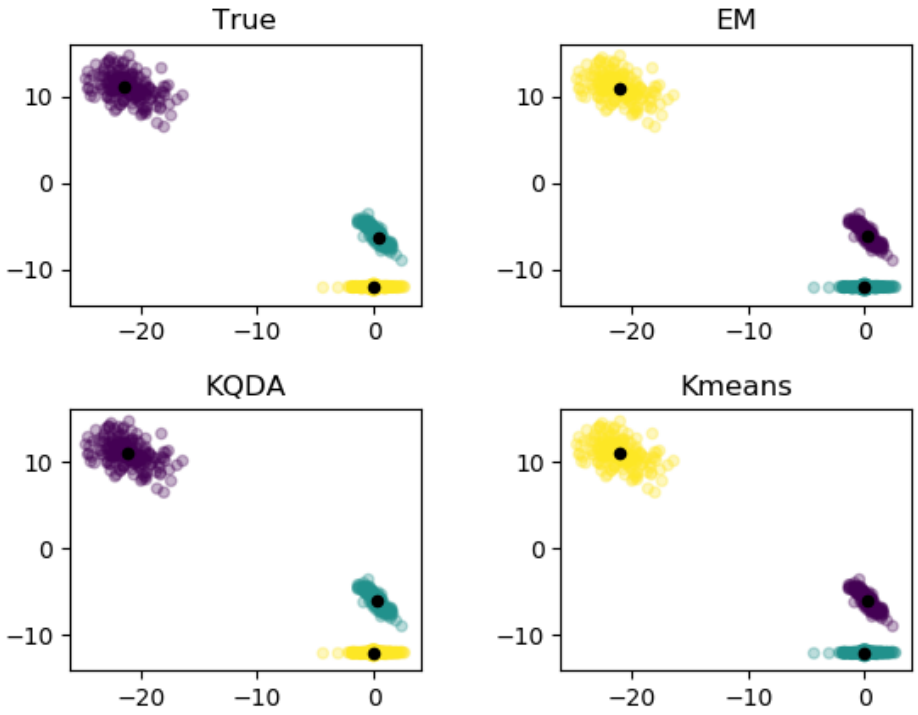
```
#### Expectation-Maximization
EM_labels, log_likelihoods, EM_parms = EM_calc(num_dims, num_samples, num_clusters, x)
#### K QDA
kqda_labels, kqda_parms = k_qda_calc(num_dims, num_samples, num_clusters, x)
#### K means
kmeans_labels, loss, kmean_mus = k_means_calc(num_dims, num_samples, num_clusters, x)

plt.legend()
plt.savefig(os.path.join(dir, 'Updates_s{}_f{}_n{}_k{}.png'.format(seed, factor, num_samples,
    num_clusters)))

# Collect all results
labels = [true_labels, EM_labels[-1], kqda_labels[-1], kmeans_labels[-1]]
mus_fin = np.array([mu, EM_parms['mu'][-1], kqda_parms['mu'][-1], kmean_mus[-1]])
algs = np.array(['True', 'EM', 'KQDA', 'Kmeans'])

#### Plot
fig = plt.figure()
fig.subplots_adjust(hspace=0.4, wspace=0.4)
for i, (lbl, alg, mu) in enumerate(zip(labels, algs, mus_fin)):
ax = fig.add_subplot(2, 2, i + 1)
data2D_plot(ax, x, lbl, mu, 'viridis', alg)

fname = os.path.join(dir, 'Results_s{}_f{}_n{}_k{}.png'.format(seed, factor, num_samples, num_clusters))
fig.savefig(fname)

mus_init = np.array([mu, EM_parms['mu'][0], kqda_parms['mu'][0], kmean_mus[0]])
init_mu_fname = os.path.join(dir, 'init_mu_s{}_f{}_n{}_k{}.png'.format(seed, factor, num_samples,
    num_clusters))
plot_init_means(x, mus_init, algs, init_mu_fname)


if __name__ == "__main__":
#experiments(seed=63, factor=10)
#5a
experiments(seed=11, factor=1, num_samples=500, num_clusters=3)
#5b
experiments(seed=63, factor=10, num_samples=500, num_clusters=3)
```

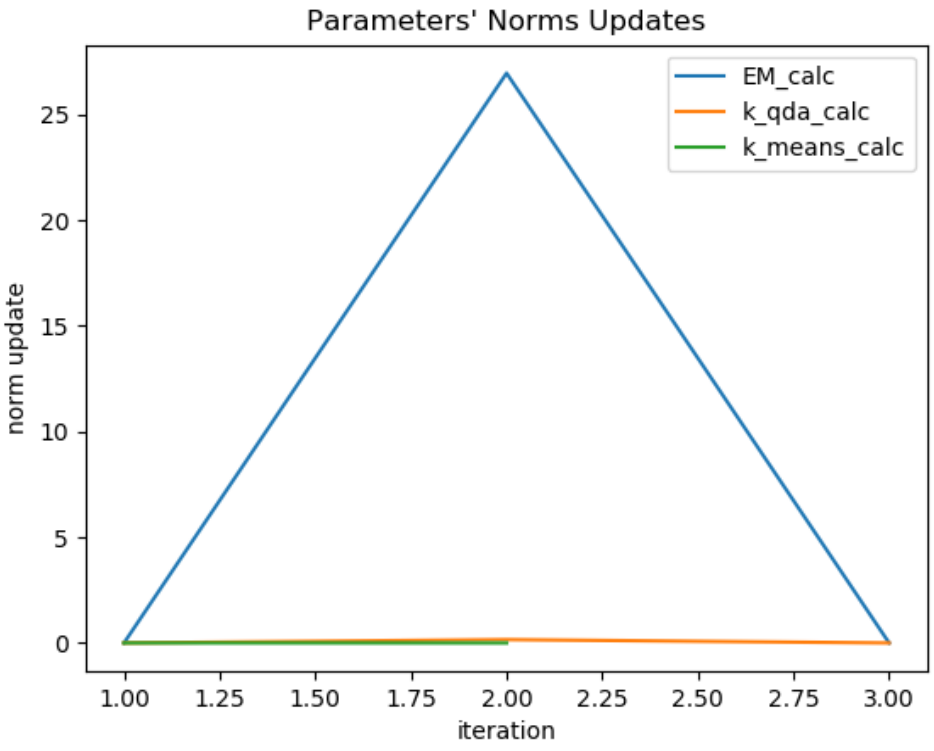
(b) **Comment on the results obtained for the following setting:**

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

In this example, all clusters are well separated so they all perform really well and converge really fast. It's interesting that at iteration 2 EM seems to swap the labels of two clusters.
All in all, if there is overlap between clusters, EM tends to beat K-QDA and K-means but it runs for more iterations. However, if clusters are well separated, there isn't much difference in terms of the predictions among these three methods.







Here are the printed results:

EM_calc
iteration 1, update 0.0002
iteration 2, update 26.950255622375835

iteration 3, update 1.8201262719230726e-09

k_qda_calc
iteration 1, update 2e-05
iteration 2, update 0.15960650153459244
iteration 3, update 0.0

k_means_calc
iteration 1, update 2e-05
iteration 2, update 0.0

Question 6. Your Own Question

Write your own question, and provide a thorough solution.

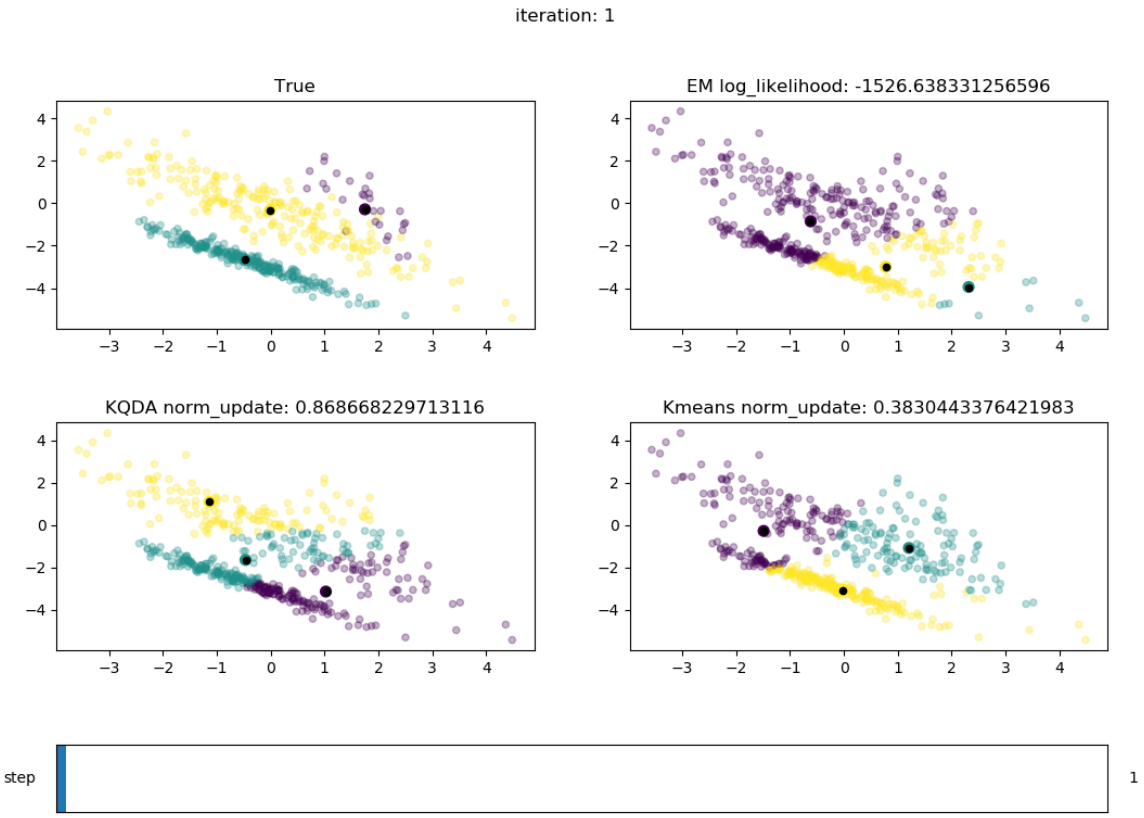
Writing your own problems is a very important way to really learn the material. The famous "Bloom's Taxonomy" that lists the levels of learning is: Remember, Understand, Apply, Analyze, Evaluate, and Create. Using what you know to create is the top-level. We rarely ask you any HW questions about the 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.

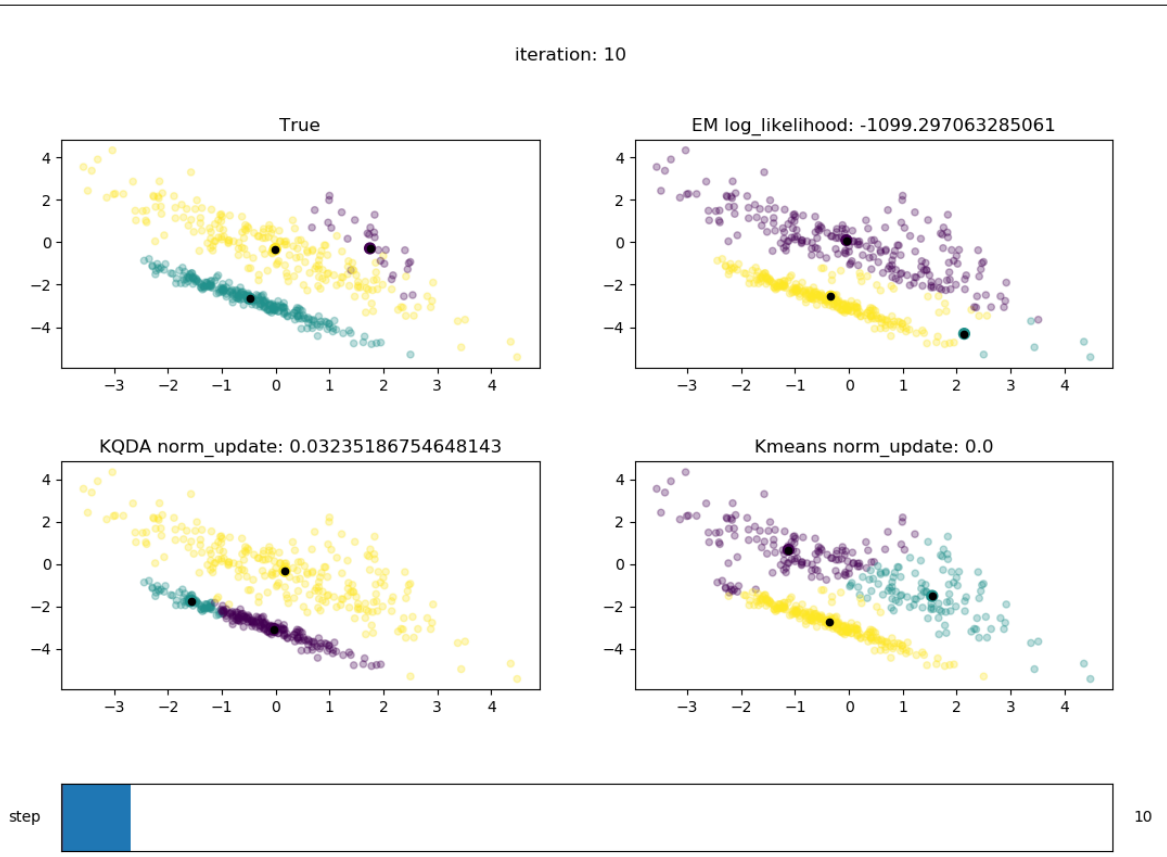
As a practical matter, having some practice at trying to create problems helps you study for exams much better than simply counting on solving existing practice problems. This is because thinking about how to create an interesting problem forces you to really look at the material from the perspective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

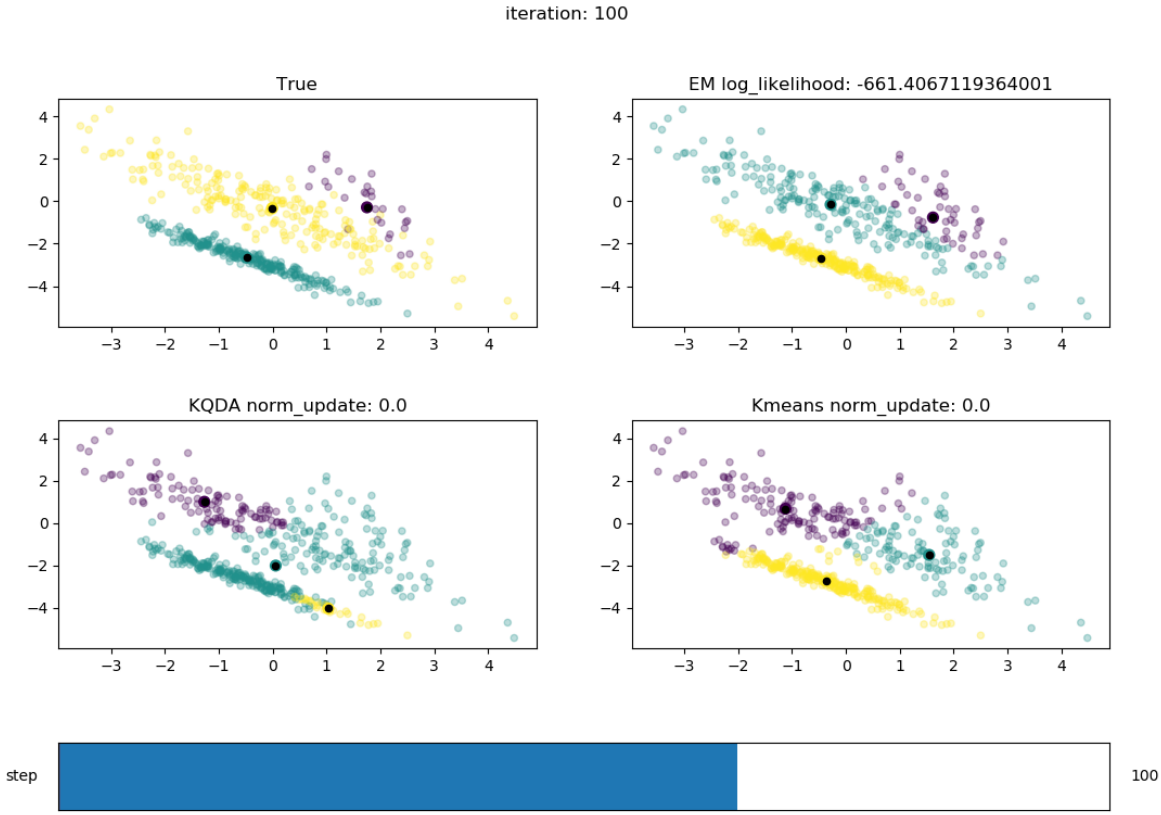
Besides, this is fun. If you want to make a boring problem, go ahead. That is your prerogative. But it is more fun to really engage with the material, discover something interesting, and then come up with a problem that walks others down a journey that lets them share your discovery. You don't have to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

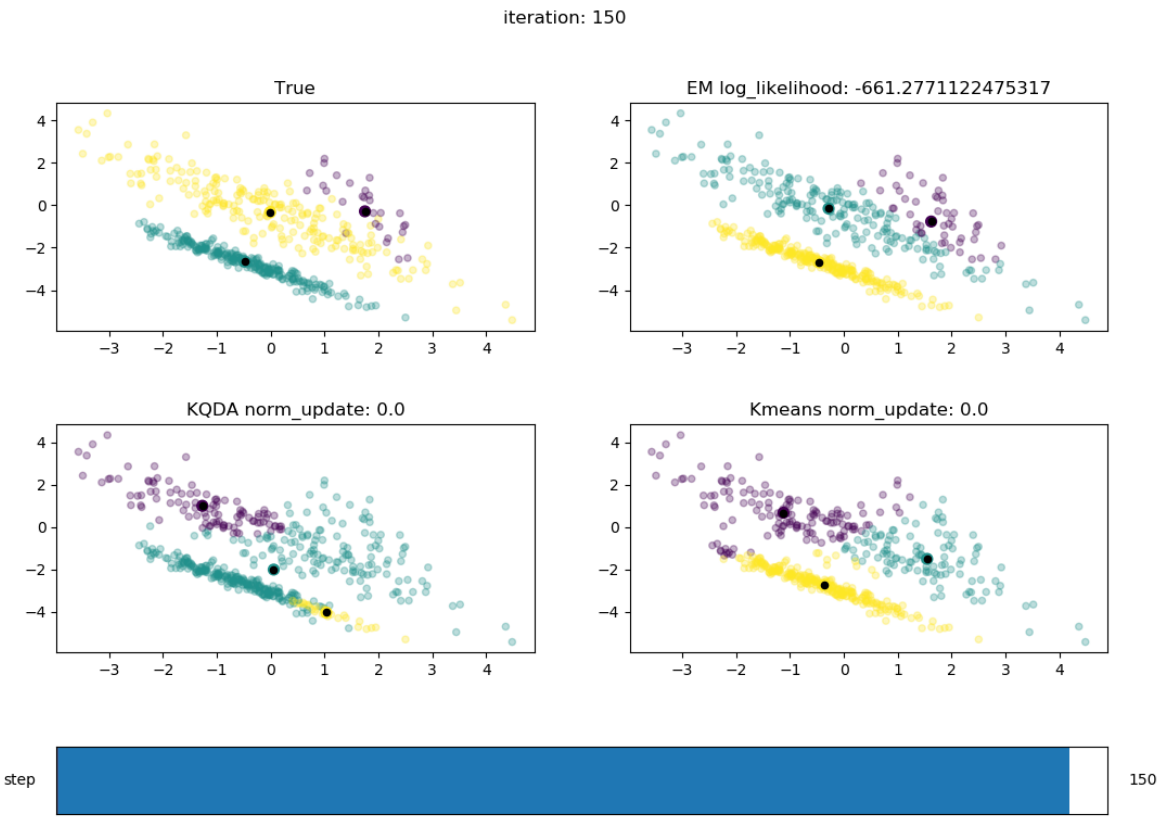
Use the provided start code in Q5 and visualize the entire updating processes of all three methods.

I tried to save gif but failed many many times so I gave up. Instead, I used the slider to manually create animation. It works well for my purpose to visualize the entire process of all three algorithms. The parameters are the same as in 5a. Here are some snapshots:




iteration: 10
True
EM log_likelihood: -1099.297063285061
KQDA norm_update: 0.03235186754648143
Kmeans norm_update: 0.0
step
10


iteration: 100
True
EM log_likelihood: -661.4067119364001
KQDA norm_update: 0.0
Kmeans norm_update: 0.0
step
100



As we can see, K-means converges really fast, followed by K-QDA. EM spends many iterations try to estimate the prior probabilities while maintain the same labels for all points after iteration 100.

```
from __future__ import division
import numpy as np
from matplotlib.widgets import Slider
import matplotlib.pyplot as plt


class ParamsWrapper(object):
def __init__(self):
self.fargs = ()

def set_params(self, fargs):
self.fargs = fargs


plot_params = ParamsWrapper()


# This function generate random mean and covariance
def gauss_params_gen(num_clusters, num_dims, factor):
mu = np.random.randn(num_clusters, num_dims) * factor
sigma = np.random.randn(num_clusters, num_dims, num_dims)
for k in range(num_clusters):
sigma[k] = np.dot(sigma[k], sigma[k].T)

return (mu, sigma)
```

```
# Given mean and covariance generate data
def data_gen(mu, sigma, num_clusters, num_samples):
labels = []
X = []
cluster_prob = np.array([np.random.rand() for k in range(num_clusters)])
cluster_num_samples = (num_samples * cluster_prob / sum(cluster_prob)).astype(int)
cluster_num_samples[-1] = num_samples - sum(cluster_num_samples[:-1])

for k, ks in enumerate(cluster_num_samples):
labels.append([k] * ks)
X.append(np.random.multivariate_normal(mu[k], sigma[k], ks))

# shuffle data
randomize = np.arange(num_samples)
np.random.shuffle(randomize)
X = np.vstack(X)[randomize]
labels = np.array(sum(labels, []))[randomize]

return X, labels


def data2D_plot(ax, x, labels, centers, cmap, title):
data = {'x0': x[:, 0], 'x1': x[:, 1], 'label': labels}
ax.scatter(data['x0'], data['x1'], c=data['label'], cmap=cmap, s=20, alpha=0.3)
ax.scatter(centers[:, 0], centers[:, 1], c=np.arange(np.shape(centers)[0]), cmap=cmap, s=50, alpha=1)
ax.scatter(centers[:, 0], centers[:, 1], c='black', cmap=cmap, s=20, alpha=1)
ax.title.set_text(title)


def plot_init_means(x, mus, algs, fname):
import matplotlib.cm as cm
fig = plt.figure()
plt.scatter(x[:, 0], x[:, 1], c='gray', cmap='viridis', s=20, alpha=0.4, label='data')
for mu, alg, clr in zip(mus, algs, cm.viridis(np.linspace(0, 1, len(mus)))):
plt.scatter(mu[:, 0], mu[:, 1], c=clr, s=50, label=alg)
plt.scatter(mu[:, 0], mu[:, 1], c='black', s=10, alpha=1)
legend = plt.legend(loc='upper right', fontsize='small')
plt.title('Initial guesses for centroids')
fig.savefig(fname)


def loss_plot(loss, title, xlabel, ylabel, fname):
fig = plt.figure(figsize=(13, 6))
plt.plot(np.array(loss))
plt.title(title)
plt.xlabel(xlabel)
plt.ylabel(ylabel)
fig.savefig(fname)


def gaussian_pdf(X, mu, sigma):
# Gaussian probability density function
return np.linalg.det(sigma) ** -.5 ** (2 * np.pi) ** (-X.shape[1] / 2.) \
* np.exp(-.5 * np.einsum('ij, ij -> i', \
X - mu, np.dot(np.linalg.inv(sigma), (X - mu).T).T))


def EM_initial_guess(num_dims, data, num_samples, num_clusters):
# randomly choose the starting centroids/means
# as num_clusters of the points from datasets
mu = data[np.random.choice(num_samples, num_clusters, False), :]
```

```
# initialize the covariance matrice for each gaussian
sigma = [np.eye(num_dims)] * num_clusters

# initialize the probabilities/weights for each gaussian
# begin with equal weight for each gaussian
alpha = [1. / num_clusters] * num_clusters

return mu, sigma, alpha


def EM_E_step(num_clusters, num_samples, data, mu, sigma, alpha):
## Vectorized implementation of e-step equation to calculate the
## membership for each of k -gaussians
Q = np.zeros((num_samples, num_clusters))
for k in range(num_clusters):
Q[:, k] = alpha[k] * gaussian_pdf(data, mu[k], sigma[k])

## Normalize so that the responsibility matrix is row stochastic
Q = (Q.T / np.sum(Q, axis=1)).T

return Q


def EM_M_step(num_clusters, num_dims, num_samples, Q, data):
# M Step
## calculate the new mean and covariance for each gaussian by
## utilizing the new responsibilities
mu = np.zeros((num_clusters, num_dims))
sigma = np.zeros((num_clusters, num_dims, num_dims))
alpha = np.zeros(num_clusters)

## The number of datapoints belonging to each gaussian
num_samples_per_cluster = np.sum(Q, axis=0)

for k in range(num_clusters):
## means
mu[k] = 1. / num_samples_per_cluster[k] * np.sum(Q[:, k] * data.T, axis=1).T
centered_data = np.matrix(data - mu[k])

## covariances
sigma[k] = np.array(
1. / num_samples_per_cluster[k] * np.dot(np.multiply(centered_data.T, Q[:, k]), centered_data))

## and finally the probabilities
alpha[k] = 1. / (num_clusters * num_samples) * num_samples_per_cluster[k]

return mu, sigma, alpha


def EM_log_likelihood_calc(num_clusters, num_samples, data, mu, sigma, alpha):
L = np.zeros((num_samples, num_clusters))
for k in range(num_clusters):
L[:, k] = alpha[k] * gaussian_pdf(data, mu[k], sigma[k])
return np.sum(np.log(np.sum(L, axis=1)))


def EM_calc(num_dims, num_samples, num_clusters, x):

log_likelihoods = []
labels = []
iter_cnt = 0
epsilon = 0.0001
max_iters = 200
```

```
update = 2 * epsilon

# initial guess
mu, sigma, alpha = EM_initial_guess(num_dims, x, num_samples, num_clusters)
mus = [mu]
sigmas = [sigma]
alphas = [alpha]
updates = [np.nan]
while (update > epsilon) and (iter_cnt < max_iters):
iter_cnt += 1

# E - Step
Q = EM_E_step(num_clusters, num_samples, x, mu, sigma, alpha)

# M - Step
mu, sigma, alpha = EM_M_step(num_clusters, num_dims, num_samples, Q, x)

mus.append(mu)
sigmas.append(sigma)
alphas.append(alpha)

# Likelihood computation
log_likelihoods.append(EM_log_likelihood_calc(num_clusters, num_samples, x, mu, sigma, alpha))

# check convergence
if iter_cnt >= 2:
update = np.abs(log_likelihoods[-1] - log_likelihoods[-2])
updates.append(update)

# print current iteration
labels.append(np.argmax(Q, axis=1))

return labels, log_likelihoods, {'mu': mus, 'sigma': sigmas, 'alpha': alphas}, updates


def kmeans_initial_guess(data, num_samples, num_clusters):
# randomly choose the starting centroids/means
# as num_clusters of the points from datasets
mu = data[np.random.choice(num_samples, num_clusters, False), :]
return mu


def kmeans_get_labels(num_clusters, num_samples, num_dims, data, mu):
# set all dataset points to the best cluster according to minimal distance
# from centroid of each cluster
dist = np.zeros((num_clusters, num_samples))
for k in range(num_clusters):
dist[k] = np.linalg.norm(data - mu[k], axis=1)

labels = np.argmin(dist, axis=0)

return labels


def kmeans_get_means(num_clusters, num_dims, data, labels):
# Compute the new means given the reclustering of the data
mu = np.zeros((num_clusters, num_dims))
for k in range(num_clusters):
idx_list = np.where(labels == k)[0]
if (len(idx_list) == 0):
r = np.random.randint(len(data))
mu[k] = data[r, :]
else:
```

```
mu[k] = np.mean(data[idx_list], axis=0)
return mu


def kmeans_calc_loss(num_clusters, num_samples, data, mu, labels):
dist = np.zeros((num_samples, num_clusters))
for j in range(num_samples):
for k in range(num_clusters):
if (labels[j] == k):
dist[j, k] = np.linalg.norm(data[j] - mu[k])
return sum(sum(dist))


def k_means_calc(num_dims, num_samples, num_clusters, x):

loss = []
labels = []
iter_cnt = 0
epsilon = 0.00001
max_iters = 100
update = 2 * epsilon

# initial guess
mu = [kmeans_initial_guess(x, num_samples, num_clusters)]
plot_mus = []
updates = [np.nan]
while (update > epsilon) and (iter_cnt < max_iters):
iter_cnt += 1
# Assign labels to each datapoint based on centroid
labels.append(kmeans_get_labels(num_clusters, num_samples, num_dims, x, mu[-1]))

# Assign centroid based on labels
mu.append(kmeans_get_means(num_clusters, num_dims, x, labels[-1]))
# check convergence
if iter_cnt >= 2:
update = np.linalg.norm(mu[-1] - mu[-2], None)
updates.append(update)
# Print distance to centroids vs iteration
loss.append(kmeans_calc_loss(num_clusters, num_samples, x, mu[-1], labels[-1]))

# saving update params
plot_mus.append(mu[-1])

return labels, loss, mu, updates


def k_qda_initial_guess(num_dims, data, num_samples, num_clusters):
# randomly choose the starting centroids/means
# as num_clusters of the points from datasets
mu = data[np.random.choice(num_samples, num_clusters, False), :]

# initialize the covariance matrice for each gaussian
sigma = [np.eye(num_dims)] * num_clusters

return mu, sigma


def k_qda_get_parms(num_clusters, num_dims, data, labels):
## calculate the new mean and covariance for each gaussian
mu = np.zeros((num_clusters, num_dims))
sigma = np.zeros((num_clusters, num_dims, num_dims))

for k in range(num_clusters):
```

```
c_k = labels == k
if (len(data[c_k]) == 0):
r = np.random.randint(len(data))
mu[k] = data[r, :]
else:
mu[k] = np.mean(data[c_k], axis=0)

if (len(data[c_k]) > 1):
centered_data = np.matrix(data[c_k] - mu[k])
sigma[k] = np.array(1. / len(data[c_k]) * np.dot(centered_data.T, centered_data))
else:
sigma[k] = np.eye(num_dims)

return mu, sigma


def k_qda_get_labels(num_clusters, num_samples, mu, sigma, data):
# set all dataset points to the best cluster according to best
# probability given calculated means and covariances
dist = np.zeros((num_clusters, num_samples))
for k in range(num_clusters):
data_center = (data - mu[k])
dist[k] = np.einsum('ij, ij -> i', data_center, np.dot(np.linalg.inv(sigma[k]), data_center.T).T)
labels = np.argmin(dist, axis=0)
return labels


def k_qda_calc(num_dims, num_samples, num_clusters, x):

loss = []
labels = []
iter_cnt = 0
epsilon = 0.00001
max_iters = 100
update = 2 * epsilon

# initial guess
mu, sigma = k_qda_initial_guess(num_dims, x, num_samples, num_clusters)
mus = [mu]
sigmas = [sigma]
updates = [np.nan]
while (update > epsilon) and (iter_cnt < max_iters):
iter_cnt += 1

# Assign labels to each datapoint based on probability
labels.append(k_qda_get_labels(num_clusters, num_samples, mus[-1], sigmas[-1], x))

# Assign centroid and covarince based on labels
mu, sigma = k_qda_get_parms(num_clusters, num_dims, x, labels[-1])

mus.append(mu)
sigmas.append(sigma)

# check convergence
if iter_cnt >= 2:
update = np.linalg.norm(mus[-1] - mus[-2], None)
update += np.linalg.norm(sigmas[-1] - sigmas[-2], None)
updates.append(update)

return labels, {'mu': mus, 'sigma': sigmas}, updates


def plot_update(iframe):
```

```
iframe = int(iframe)
(mu, sigma, x, true_labels,
EM_labels, log_likelihoods, EM_parms, EM_update,
kqda_labels, kqda_parms, kqda_update,
kmeans_labels, loss, kmean_mus, kmean_update, plotaxes) = plot_params.fargs

fig = plt.gcf()
fig.suptitle('iteration: ' + str(iframe))

iEM = iframe if iframe < len(EM_labels) else -1
ikqda = iframe if iframe < len(kqda_labels) else -1
ikmeans = iframe if iframe < len(kmeans_labels) else -1

# Collect all results
labels = [true_labels, EM_labels[iEM], kqda_labels[ikqda], kmeans_labels[ikmeans]]
mus_fin = np.array([mu, EM_parms['mu'][iEM], kqda_parms['mu'][ikqda], kmean_mus[ikmeans]])
algs = np.array(['True', 'EM', 'KQDA', 'Kmeans'])

#### Plot
for i, (lbl, alg, mu) in enumerate(zip(labels, algs, mus_fin)):
ax = plotaxes[i]
ax.cla()
data2D_plot(ax, x, lbl, mu, 'viridis', alg)
if alg == 'EM':
ax.set_title(alg + ' log_likelihood: ' + str(log_likelihoods[iEM]))
if alg == 'KQDA':
ax.set_title(alg + ' norm_update: ' + str(kqda_update[ikqda]))
if alg == 'Kmeans':
ax.set_title(alg + ' norm_update: ' + str(kmean_update[ikmeans]))


def experiments(seed, factor, dir='plots', num_samples=500, num_clusters=3):
#if not os.path.exists(dir):
#    os.makedirs(dir)

np.random.seed(seed)
num_dims = 2

# generate data samples
(mu, sigma) = gauss_params_gen(num_clusters, num_dims, factor)
x, true_labels = data_gen(mu, sigma, num_clusters, num_samples)

#### Expectation-Maximization
EM_labels, log_likelihoods, EM_parms, EM_update = EM_calc(num_dims, num_samples, num_clusters, x)
#### K QDA
kqda_labels, kqda_parms, kqda_update = k_qda_calc(num_dims, num_samples, num_clusters, x)
#### K means
kmeans_labels, loss, kmean_mus, kmean_update = k_means_calc(num_dims, num_samples, num_clusters, x)


plt.figure(figsize=(12.5, 8.5))
ax1 = plt.subplot2grid((9, 2), (0, 0), rowspan=3)
ax2 = plt.subplot2grid((9, 2), (0, 1), rowspan=3)
ax3 = plt.subplot2grid((9, 2), (4, 0), rowspan=3)
ax4 = plt.subplot2grid((9, 2), (4, 1), rowspan=3)
ax_slider = plt.subplot2grid((9, 2), (8, 0), colspan=2)

plot_params.set_params(
(mu, sigma, x, true_labels,
EM_labels, log_likelihoods, EM_parms, EM_update,
kqda_labels, kqda_parms, kqda_update,
kmeans_labels, loss, kmean_mus, kmean_update,
(ax1, ax2, ax3, ax4))
```

```
)
plot_update(0)
nframes = max([len(EM_labels), len(kqda_labels), len(kmeans_labels)])
step_slider = Slider(ax_slider, 'step', valmin=0, valmax=nframes,
valinit=0, valfmt="%d")
step_slider.on_changed(plot_update)
plt.show()

if __name__ == "__main__":
#experiments(seed=63, factor=10)
#5a
experiments(seed=11, factor=1, num_samples=500, num_clusters=3)
#5b
#experiments(seed=63, factor=10, num_samples=500, num_clusters=3)
```