Alexander Bukharin

Homework 3

2) First we show our code with the low rank approximation (It is long because I didn’t use any packages) We carry out the normal GMM, except we use a low rank approximation for the covariance matrix:

import numpy as np

import matplotlib.pyplot as plt

import scipy.stats

class EM():

'''

EM Algorithm for MNIST Dataset, each image is of length 784. We distinguish between 2s and 6s

'''

def \_\_init\_\_(self):

# Initialize variable

self.mu2 = np.random.normal(0, 1, 784)

self.mu6 = np.random.normal(0, 1, 784)

self.cov2 = np.identity(784)

self.cov6 = np.identity(784)

self.pi2 = 0.5

self.pi6 = 0.5

self.probs = np.array([1190, 2])

def multivariate\_normal(self, image):

# We use a low rank approximation

diagonal2 = np.diag(self.cov2).copy()

diagonal6 = np.diag(self.cov6).copy()

# Diagonal may contain zeroes, so we take those out

for i in range(len(diagonal2)):

if diagonal2[i] <= 0:

diagonal2[i] = 1.0

if diagonal6[i] <= 0:

diagonal6[i] = 1.0

# Calculate determinants for each covariance matrix

det2 = np.prod(diagonal2)

det6 = np.prod(diagonal6)

# Calculate low-rank approximation of inverses

inv2 = np.diag(1.0 / diagonal2)

inv6 = np.diag(1.0 / diagonal6)

# Calculate constant part of normal

constant6 = 1.0 / np.sqrt(det6)

constant2 = 1.0 / np.sqrt(det2)

# Calculate exponent part of normal

exponent2 = np.exp(-0.5 \* np.matmul(np.matmul((image - self.mu2).transpose(), inv2), (image - self.mu2)))

exponent6 = np.exp(-0.5 \* np.matmul(np.matmul((image - self.mu6).transpose(), inv6), (image - self.mu6)))

return np.array([1 / constant2 \* exponent2, 1 / constant6 \* exponent6])

def calculate\_probs(self, data):

#Data (num images, num classes)

probs = []

for i in range(len(data[0])):

prob = self.multivariate\_normal(data[:, i]) / np.sum(self.multivariate\_normal(data[:, i]))

probs.append(prob / np.linalg.norm(prob))

self.probs = np.array(probs)

def update\_mean(self, data):

# Calculate mean

mu2 = np.sum(self.probs[:, 0] \* data, axis = 1) / np.sum(self.probs[:, 0])

mu6 = np.sum(self.probs[:, 1] \* data, axis = 1) / np.sum(self.probs[:, 1])

self.mu2 = mu2

self.mu6 = mu6

def update\_variance(self, data):

# Calculate covariance

cov2 = np.zeros([784,784])

cov6 = np.zeros([784,784])

reg2 = np.sum(self.probs[:, 0])

reg6 = np.sum(self.probs[:, 1])

for j in range(1190):

cov2 = cov2 + self.probs[j, 0] \* np.outer((self.mu2 - data[:, j]), (self.mu2 - data[:, j])) / reg2

cov6 = cov6 + self.probs[j, 1] \* np.outer((self.mu6 - data[:, j]), (self.mu6 - data[:, j])) / reg6

self.cov2 = cov2

self.cov6 = cov6

#print(self.cov2)

def update\_pis(self, data):

# Calculate pis

self.pi2 = np.sum(self.probs[:, 0]) / len(data[0])

self.pi6 = np.sum(self.probs[:, 1]) / len(data[0])

def loss(self, data):

# Calculate loss

loss2 = 0

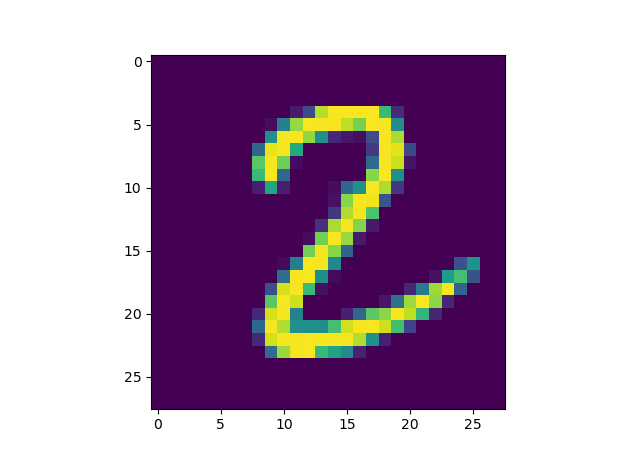
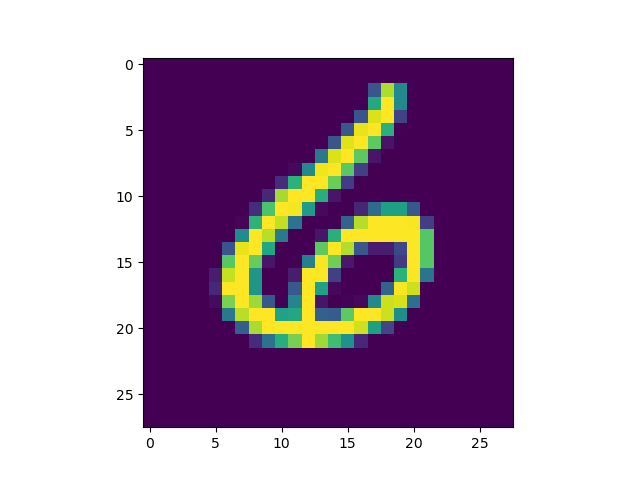
loss6 = 0

for j in range(1990):

loss2 += self.probs[:, 0][j] \* self.pi2 + self.probs[:, 0][j] \* self.multivariate\_normal(data[:, 0])[0]

loss6 += self.probs[:, 1][j] \* self.pi2 + self.probs[:, 1][j] \* self.multivariate\_normal(data[:, 0])[1]

a) Here are visualizations of a two and a six:

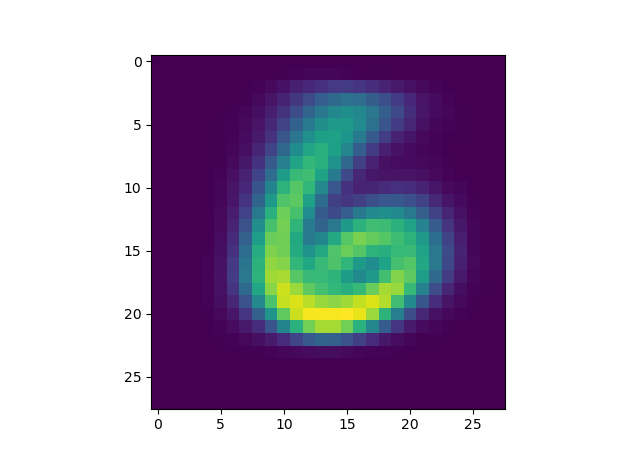
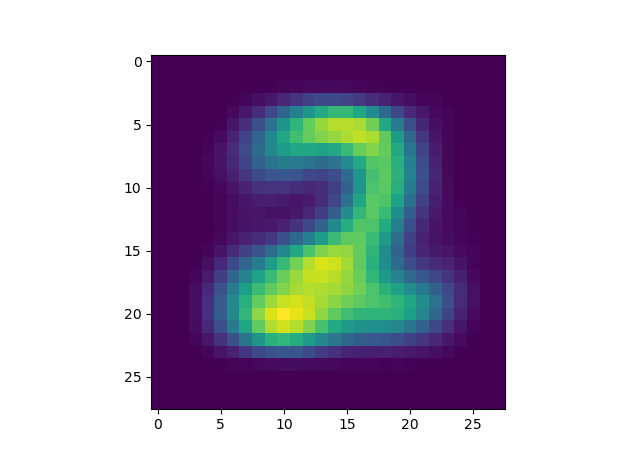


b) We now plot the log-likelihood versus the number of iterations:

A screenshot of a social media post

Description automatically generated

c) Now we plot the mean for the two classes:



So we can see that the GMM can learn the shapes pretty well.