University of Toronto Mississauga
Department of Mathematical and Computational Sciences
CSC 311 - Introduction to Machine Learning, Fall 2020

Assignment 3

Due date: Tuesday Deccember 8, 11:59pm.
No late assignments will be accepted.

As in all work in this course, 20% of your grade is for quality of presentation, including the use of good English, properly commented and easy-to-understand programs, and clear proofs. In general, short, simple answers are worth more than long, complicated ones. Unless stated otherwise, all answers should be justified. The TA has a limited amount of time to devote to each assignment, so what you hand in should be legible (either typed or *neatly* hand-written), well-organized and easy to evaluate. (An employer would demand no less.) All computer problems are to be done in Python with the NumPy, SciPy and scikit-learn libraries. Unless specified otherwise, programs will be graded primarily by the output they produce.

Hand in five files: The source code of all your programs (functions and script) in a single Python file, a pdf file of figures generated by the programs, a pdf file of all printed output, a pdf file of answers to all the non-programming questions (such as proofs and explanations), and a scanned, signed copy of the cover sheet at the end of the assignment. All proofs should be typed. (Word, Latex and many other programs have good facilities for typing equations.)

Be sure to indicate clearly which question(s) each program and piece of output refers to. All the Python code (functions and script) for a given question should appear in one location in your source file, along with a comment giving the question number. All material in all files should appear in order; *i.e.*, material for Question 1 before Question 2 before Question 3, etc. It should be easy for the TA to identify the material for each question. In particular, all figures should be titled, and all printed output should be identified with the Question number. The five files should be submitted electronically as described on the course web page. In addition, if we run your source file, it should not produce any errors, it should produce all the output that you hand in (figures and print outs), and it should be clear which question each piece of output refers to. *Output that is not labelled with the Question number will not be graded. Programs that are suppose to produce output, but don't, will not be graded.*

Style: Use the solutions to Assignment 1 and 2 and the midterm as a guide/model for how to present your solutions to this assignment.

I don't know policy: If you do not know the answer to a question (or part), and you write "I don't know", you will receive 20% of the marks of that question (or part). If you just leave a question blank with no such statement, you get 0 marks for that question.

No more questions will be added.

Tips on Scientific Programming in Python

If you haven't already done so, please read the NumPy tutorial on the course web page.

Special numbers. The term `numpy.inf` represents infinity. It results from dividing by 0 in numpy. It can also result from overflow (*i.e.*, from numbers that are too large to represent in the computer, like 10^{1000}). The term `numpy.nan` stands for "not a number", and it results from doing 0/0, inf/inf or inf-inf in numpy.

Indexing. Array indexing begins at 0, not 1. Thus, if `A` is a matrix, then `A[7,0]` is the element in row 7 and column 0. Likewise, `A[0,4]` is the element in row 0 and column 4. We use both indices and ordinal numbers to refer to rows and columns. Thus, the first row is row 0, the second row is row 1, etc. Slicing allows large segments of an array to be referenced. For example, `A[:,5]` returns column 5 of matrix `A`, and `A[7,[3,6,8]]` returns elements 3, 6 and 8 of row 7. Similarly, if `v` is a vector, then the statement `A[6,:]=v` copies `v` into row 6 of matrix `A`. You can read more about indexing in the following Numpy tutorial: `https://numpy.org/doc/stable/reference/arrays.indexing.html`

Vectorized code. Unless otherwise specified, *do not use loops* for numerical computations, as they are very slow in Python. In particular, avoid iterating over the elements of a large vector or matrix. Instead, use NumPy's vector and matrix operations, which are much faster and can be executed in parallel on a gpu. This is called *vectorized* code. For example, if `A` is a matrix and `v` is a column vector, then `A+v` will add `v` to every column of `A`. Likewise for rows and row vectors. Note that if `A` and `B` are matrices, then `A*B` performs *element-wise* multiplication, *not* matrix multiplication. To perform matrix multiplication, you can use `numpy.matmul(A,B)`, or `A@B` in Python 3. Also, the functions `sum` and `mean` in `numpy` are useful for summing or averaging over all or part of an array. Many NumPy functions that are defined for single numbers can be passed lists, vectors and matrices instead. For example, $f([x_1, x_2, ..., x_n])$ returns the list $[f(x_1), f(x_2), ..., f(x_n)]$. The same is true for many user-defined functions.

Likewise, unless specified otherwise, do not use recursion, or any Python function that operates on lists, such as `zip` or `map`, or any higher-order function that operates on numpy arrays. This includes many numpy functions listed under "functional programming", such as `apply-along-axis`. These functions are often loops in disguise and can be very slow. With few exceptions, arrays should be the only large objects in your program, and you should only operate on them with NumPy functions.

Sometimes, you will need loops to iterate over short lists or to implement iterative algorithms, such as gradient descent, or you may need recursion to traverse a small graph. You may also want to append results to a list during each iteration of a loop. This is OK. Typically, this represents a tiny fraction of total computation time, since large arrays are processed at each iteration of a loop or at each node in graph. It is these compute-intensive operations on large arrays that must be vectorized. In general, all linear-algebra computations should be vectorized, that is, implemented using Numpy's matrix and vector functions. In fact, one of the goals of this course is to teach you to write vectorized code, since it is ubiquitous in machine learning.

To give you maximum practice, all your vectorized code should only use basic operations of linear algebra, such as matrix addition, multiplication, inverse and transpose, unless specified otherwise. The point here is for you to implement vectorized code yourself, not to use complex Numpy procedures that solve most of a problem for you. You may, of course, use Numpy's array-indexing facilities to vectorize operations on all or part of an array.

Broadcasting. Another index-related feature in Numpy for vectorization is *broadcasting*, which combines arrays of different shapes. As an example, suppose `A` and `B` are Numpy arrays, where `shape(A) = [I,J,K]` and `shape(B) = [I,K]`. And suppose we want to define a new array, `C`, where `shape(C) = [I,J,K]` and $C_{ijk} = A_{ijk}B_{ik}$ for all i, j, k. We can do this with the following Numpy statements:

```
    B = np.reshape(B,[I,1,K]}
    C = A*B
```

If we want to compute $D_{ij} = \sum_k A_{ijk}B_{ik} = \sum_k C_{ijk}$, we can simply add the statement

```
    D = np.sum(C,axis=2)
```

Similarly, suppose `shape(A) = [I,K]` and `shape(B) = [J,K]`, and we want to define `C`, where `shape(C) = [I,J,K]` and $C_{ijk} = A_{ik} + B_{jk}$ for all i, j, k. We can do this with the following Numpy statements:

```
    A = np.reshape(A,[I,1,K]}
    B = np.reshape(B,[1,J,K]}
    C = A+B
```

You can read more about broadcasting in the following Numpy tutorial: `https://numpy.org/doc/stable/user/basics.broadcasting.html`

Plotting. For generating and annotating plots, the following functions in `matplotlib.pyplot` are used frequently: `plot`, `scatter`, `figure`, `xlabel`, `ylabel`, `title`, `suptitle`, `xlim` and `ylim`. The functions `semilogx`, `semilogy` and `loglog` generate plots with a log scale on one or both axes. You can use Google to conveniently look up these functions. e.g., Google "pyplot suptitle". To plot a smooth function, $y = f(x)$, you compute y for many closely-spaced values of x, and then plot all the x, y pairs. For example, the following code plots the function $y = \sin x$ for x between 0 and 10 by plotting 1000 values of y at 1000 evenly-spaced values of x.

```
import numpy as np
import matplotlib.pyplot as plt
xmin = 0
xmax = 10
```

```
xList = np.linspace(xmin,xmax,1000)
yList = np.sin(xList)
plt.plot(xList,yList)
```

The `plot` function draws a tiny line segment between consecutive (x, y) pairs, giving the illusion of a smooth curve.

Printouts. Finally, if a program prints any output, you should identify the question (and part) that it comes from by preceding all code for that part with lines like the following:

```
    print('\n')
    print('Question 3(d).')
    print('-------------')
```

If a program is not suppose to print anything, then do not include these lines in your program, so as to reduce clutter in your output. In any case, you do not have to include these lines in your line-counts of code.

Unless otherwise specified, you may assume in this assignment that all inputs are correct and no error-checking is required.

Tips on Proving Theorems

When proving theorems, all steps should be justified. Appeals to intuition and leaps of logic are not allowed. Explanations in English should be minimized and must not replace careful logical inference. Trivial or obvious steps can be skipped (but if you have to think about something for more than a few seconds, then it is not obvious). Everything should be proved from scratch, ie, from basic results and definitions. Unless specified otherwise, you should not use any powerful theorems or results from the lecture slides, notes, books or any other source. The point is to prove everything yourself. Proofs should be clear and concise. Use the proofs in the solutions to Assignment 1 and the midterm as a guide to what proofs should look like. Proofs like this will receive full marks. Note, in particular, that *every* step is justified with a short explanation (*e.g.*, *by the definition of matrix multiplication*, or *since* $X_{ij} = x_j^{(i)}$, or *by Equation (1)*). Unless otherwise specified, you should never write out the contents of large vectors or matrices, as in

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_{11} & \dots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \dots & x_{nm} \end{pmatrix} \begin{pmatrix} w_1 \\ \vdots \\ w_m \end{pmatrix}$$

Instead, you should write $y_i = \sum_j x_{ij} w_j$, which is more compact and leads to much clearer proofs. Often, you can simply write $y = Xw$, which is the same thing in vectorized form. When you need to refer to matrix elements, it is convenient to use the notation $[A]_{ij}$ to refer to the ij^{th} element of matrix A, and $[V]_i$ for the i^{th} element of vector V. In some cases, the square brackets are unecessary. For example, $[w]_j = w_j$ in the equations above. However, when A or V is a complicated expression, we do need them. For example, here are some equalities you may find useful:

$$\begin{aligned}
[AB]_{ij} &= \sum_k A_{ik}B_{kj} && \text{matrix multiplication} \\
[Aw]_i &= \sum_j A_{ij}w_j && \text{matrix-vector multiplication} \\
\left[A^T\right]_{ij} &= A_{ji} && \text{matrix transpose} \\
\left[\frac{\partial f}{\partial w}\right]_j &= \frac{\partial f}{\partial w_j} && \text{gradient}
\end{aligned}$$

You may need other such equalities in your proofs. Some may be trivial (such as $[w]_j = w_j$). All others should be justified.

Finally, to speed up grading, all proofs should be typed.

1. *Principle Components Analysis (PCA).* (26 points total)

   In this question, you will be implementing and applying PCA to the MNIST data. Some of your computations will be quite sensitive to numerical error, so you will have use 64-bit arithmetic. The simplest way to ensure this is to convert the MNIST data to 64-bit floating-point numbers immediately after reading it from file, as follows:

```
with open('mnistTVT.pickle','rb') as f:
    Xtrain,Ttrain,Xval,Tval,Xtest,Ttest = pickle.load(f)

Xtrain = Xtrain.astype(np.float64)
Xval = Xval.astype(np.float64)
Xtest = Xtest.astype(np.float64)
```

   Since PCA is unsupervised, you will not be using the labels—`Ttrain`, `Tval` and `Ttest`—in this question.

   Tips. When performing matrix multiplication, be sure to get the order of the matrices correct, *e.g.* AB v.s. BA. Besides being incorrect, the wrong order can generate gargantuan matrices that take forever to compute or do not fit in your computer memory. Likewise, when computing covariance matrices from a data matrix, X, be careful not to compute the covariance of, X^T, which besides giving the wrong answer, can be huge. Finally, if your computer has difficulty doing PCA with the entire MNIST training set, which has 50,000 rows, then you may use just the first 5,000 rows. This may affect some of your answers, so be sure to indicate the change by printing the following message at the start of the output for this question:

   **** Using only 5,000 MNIST training points ****

In the questions below, do not use any Python loops unless otherwise specified. You will be using some functions from `sklearn`, but only those that are specified. The point is to implement everything else yourself from scratch. All code should be vectorized.

(a) (3 points) Use PCA to project the MNIST data onto a 30-dimensional subspace. Specifically, use the Python class `PCA` in `sklearn.decomposition` to create a pca instance that keeps 30 components. Fit the instance to the MNIST training data. Use the `transform` method to reduce the MNIST test data to 30 dimensions. We shall call this the reduced data. Use the method `inverse_transform` to transform the reduced data back to the original space. We shall refer to the result as the projected data, because it is the result of projecting the MNIST data onto a 30-dimensional subspace. Like the MNIST data, the projected data has 784 components, but the reduced data has 30 components.

Display the first 25 digits of the projected MNIST test data using a greyscale colormap. Display them in a single figure in a 5x5 grid (much as you did in Assignment 1). Title the figure, *Question 1(a): MNIST test data projected onto 30 dimensions*. Figure 2 at the end of the assignment shows what the second 25 projected digits look like if everything is working properly. Notice how each digit can be clearly identified despite the distortion.

Design your code to help you in the questions below. Do not use any loops, except for generating the 5x5 grid of images.

(b) (3 points) Repeat part (a), but project the MNIST test data onto a 3-dimensional subspace. Title the figure appropriately. This can be done in at most 2 lines of highly-readible code. Figure 3 at the end of the assignment shows what the second 25 projected digits look like if everything is working properly. Notice that many digits can no longer be identified.

(c) (3 points) Repeat part (a), but project the MNIST test data onto a 300-dimensional subspace. Title the figure appropriately. This can be done in at most 2 lines of highly-readible code. Figure 4 at the end of the assignment shows what the second 25 projected digits look like if everything is working properly. Notice how most of the images are now fairly clear and crisp, though a pale grey background remains.

(d) (12 points total for parts (d) and (f)) Implement PCA. That is, define a python function `myPCA(X,K)` that uses principle components analysis to project the data in matrix `X` onto a `K`-dimensional subspace. As usual, each row in `X` is a data point. Unlike part (a), there is no distinction between training and test data. Instead, `X` shall act as both. Your function should return `Xp`, the projected version of `X`. Note that if `X` is a NxM matrix, then so is `Xp`.

Do not use any functions in `sklearn`. Instead, you should implement everything yourself from scratch. As described in the lecture slides, you will need to compute the mean of the data points, and the eigenvectors of the covariance matrix. You may use the numpy function `mean`, but for full marks, do not use the function `cov`, or any other function that computes a covariance matrix. Instead, you should compute it yourself from more-basic numpy functions. Because the covariance matrix is symmetric, you should use the numpy function `eigh` to compute its eigenvectors. You can easily define `myPCA` in 9 lines of highly-readible code. Do not use any loops.

Hint 1: carefully study Lecture 8 and Tutorial 7 and the numpy user guide for `eigh`. Hint 2: first get your function working with the `cov` function, and then try constructing your own covariance matrix.

(e) (5 points) The formula for the covariance matrix of a data set is given at the top of slide 25/40 in lecture 8. Propose and prove a vectorized version of this formula. Use this version to compute the covariance matrix in part (d). In addition, prove that the covariance matrix is symmetric. (This is a non-programming question.)

ANSWER: From the equation on Slide 25 of Lecture 8, the covariance matrix is given by

$$\hat{\Sigma} \;=\; \frac{1}{N}\sum_n (x^{(n)} - \hat{\mu})(x^{(n)} - \hat{\mu})^T \tag{1}$$

Here, the $x^{(n)}$ are column vectors, and their average, $\hat{\mu}$, is also a column vector. Let X be a data matrix with N rows whose n^{th} row is the transpose of $x^{(n)}$. Then $X_{ni} = x_n^{(i)}$. Also, let X^C be a centered version of X in which the n^{th} row is the transpose of $x^{(n)} - \hat{\mu}$. Thus, $X^C_{ni} = x_i^{(n)} - \hat{\mu}_i$. (This is easily implemented in Numpy with broadcasting.) Also note that $[yy^T]_{ij} = y_i y_j$, for any column vector, y. Keeping these points in mind,

$$\begin{aligned}
[\hat{\Sigma}]_{ij} &= \frac{1}{N}\sum_n [(x^{(n)} - \hat{\mu})(x^{(n)} - \hat{\mu})^T]_{ij} && \text{by the definition of matrix addition}\\
&= \frac{1}{N}\sum_n [x^{(n)} - \hat{\mu}]_i [x^{(n)} - \hat{\mu}]_j && \text{as noted above}\\
&= \frac{1}{N}\sum_n (x_i^{(n)} - \hat{\mu}_i)(x_j^{(n)} - \hat{\mu}_j) && \text{by the definition of vector subtraction}\\
&= \frac{1}{N}\sum_n X^C_{ni} X^C_{nj} && \text{by the definition of } X^C\\
&= \frac{1}{N}\sum_n [(X^C)^T]_{in} X^C_{nj} && \text{by the definition of matrix transpose}\\
&= \frac{1}{N}[(X^C)^T X^C]_{ij} && \text{by the definition of matrix multiplication}
\end{aligned}$$

Thus, $\hat{\Sigma} = \frac{1}{N}(X^C)^T X^C$ since all their components are equal. This is the vectorized version of Equation (1).

To prove symmetry, first recall a few basic results about matrix transposition:

- $(AB)^T = B^T A^T$ for any two compatible matrices, A and B.
- $(A^T)^T = A$.

- A matrix, C, is symmetric iff $C^T = C$.

With these in mind, we note that that any matrix of the form B^TB is symmetric, since $(B^TB)^T = B^T(B^T)^T = B^TB$. We proved above that the covariance matrix $\hat{\Sigma}$ has this form. It is therefore symmetric.

(f) Use the function `myPCA` from part (d) to project the MNIST training data onto a 100-dimensional subspace. Call the projected data `myXtrainP`. Display the first 25 digits in `myXtrainP` in a 5x5 grid. Title the figure, *Question 1(f): MNIST data projected onto 100 dimensions (mine).* The images should look in-between those of parts (a) and (c).

Do the same thing using `sklearn`. That is, use the class `PCA` in `sklearn.decomposition` to project the MNIST training data onto a 100-dimensional subspace. Call the projected data `XtrainP`. The `PCA` class provides a number of algorithms for doing PCA, including fast approximate algorithms. Be sure to specify `svd_solver='full'`, which specifies the exact method and is comparable to what you implemented in part (d). Display the first 25 digits in `XtrainP` in a 5x5 grid. Title the figure, *Question 1(f): MNIST data projected onto 100 dimensions (sklearn).* The figures for `XtrainP` and `myXtrainP` should look identical.

Compute and print out the RMS difference between `XtrainP` and `myXtrainP`. That is compute `RMS(XtrainP-myXtrainP)`, where

$$RMS(X) \;=\; \sqrt{\sum_{ij} X_{ij}^2/N}$$

for any $N \times M$ matrix X. Do not use any loops in computing `RMS`. If everthing is working correctly the value of `RMS(XtrainP-myXtrainP)` should be less than 10^{-12}.

2. *Regularization.* (36 points total)

In this question, you will use dimensionality reduction to reduce overfitting and increase accuracy when training a classifier on MNIST data. We thus use dimensionality reduction as a kind of regularization, and we shall see that it can be much more effective than more traditional forms of regularization.

Overfitting occurs when there is not enough data and too many parameters. In this question, therefore, we shall use a small fraction of the MNIST training data. We call this the small training set, and it consists of just the first 200 training points. In addition, we shall make use of a slightly larger data set consisting of the first 300 training points. We call this the debugging data set, since it is meant to help you debug your programs.

Occasionally, I shall describe the correct answer to a question. Normally, these descriptions refer to answers based on the small data set of 200 training points. The only exception is a number of graphs at the end of this assignment that are based on the debugging data set of 300 training points. You should be able to duplicate these graphs, but do not hand them in. In fact, what you hand in should make no mention of the debugging data set. Everything handed in for Questions 2 and 3 should be for the small data set consisting of the first 200 MNIST training points.

You shall use the small data set to train a classifier for Gaussian Discriminant analysis (QDA), as in Assignment 2. Recall that QDA has many more parameters than naive Bayes or logistic regression, so it will more-easily exhibit overfittng. In particular, since the MNIST data has 784 features and ten classes, QDA requires estimating ten 784×784 covariance matrices. This means estimating approximately $10 \times 784^2/2 = 3,073,280$ parameters.[1] It is impossible to accurately estimate this many parameters with just 200 data points. You shall therefore explore possibilities for regularization, which reduces the effective number of parameters.

The traditional way to regularize QDA is to modify the covariance matrix to shrink the importance of the estimated parameters. There are a variety of regularization methods of this kind, referred to as "shrinkage" methods. The typical approach is to combine the estimated covariance matrix with a matrix that has fewer (if any) parameters. For example,

$$\Sigma_{shrinkage} = (1-\lambda)\hat{\Sigma} + \lambda I$$

where I is the identity matrix, λ is a number between 0 and 1, and $\hat{\Sigma}$ is the estimated covariance matrix, as described in Question 1(e). $\Sigma_{shrinkage}$ is thus a weighted average of $\hat{\Sigma}$ and I. It is used as the covariance matrix in QDA.

Notice, that if $\lambda = 0$, then $\Sigma_{shrinkage} = \hat{\Sigma}$, so there is no reduction in the number of parameters to estimate, and no regularization. At the other extreme, if $\lambda = 1$, then $\Sigma_{shrinkage} = I$, so there are no parameters to estimate, which represents maximum regularization. In general, as λ increases, so does regularization, so overfitting decreases and underfitting increases. The trick is to find a good value for λ. This is typically done using validation data, which you will do in this question.

The class `QuadraticDiscriminantAnalysis` in `sklearn` has a built-in mechanism for regularization similar to (but somewhat more sophisticated than) the one just described. To invoke it with a value of $\lambda = 0.3$, say, you set `reg_param=0.3` when creating the QDA classifier. If you find that fitting QDA to data creates frequent warning messages about convergence, then you can suppress them. To do this, you simply replace statements like `clf.fit(X,T)` by the following:

```
from sklearn.utils.testing import ignore_warnings
ignore_warnings(clf.fit)(X,T)
```

Of course, the import only needs to be done once.

In the questions below, do not use any Python loops unless otherwise specified. You will be using some functions from `sklearn`, but only those that are specified. The point is to implement everything else yourself from scratch. All code should be vectorized.

[1]Because a covariance matrix is symmetric, it has about half as many independent parameters as an arbitrary matrix.

(a) (3 points) Since the MNIST data contains 10 classes of roughly equal size, we would expect that randomly guessing the class of an input image would be correct about 10% of the time. In this question, you shall show that training a QDA classifier on the small data set without regularization is little better than random guessing.

In particular use the Python class `QuadraticDiscriminantAnalysis` in the module `sklearn.discriminant_analysis` to define a QDA classifier. Fit the classifier to the small training set without specifying any regularization. Use the `score` method to compute the accuracy of the classifier on the small training set and the (full) test set. Print the training accuracy and the test accuracy. (Print the test accuracy to at least 4 significant digits.) If everything is working corrctly the training accuracy should be 1.0, and the test acuracy should be about 0.14. This demonstrates severe overfitting, since the classifier fits the training data perfectly but is little better than random guessing on the test data.

In this question, there is no regularization and no dimensionality reduction, so extreme overfitting is the result, more extreme than anywhere else in the assignment. Because of this, the entire system is highly sensitive to tiny differences in numerical error, so your test accuracy may deviate from mine (0.14) more than anywhere else in this assignment. This is OK as long as your test accuracy is not much better than random guessing (0.1) and your training accuracy is 1.

(b) (5 points) Using the class `QuadraticDiscriminantAnalysis` in `sklearn`, write a Python program that uses the validation data to find a good value of the regularization parameter for QDA. Specifically, define QDA classifiers with `reg_param` $= 2^{-n}$, for $n = 0, 1, .., 20$. Train each of the classifiers on the small training set. Use the `score` method to compute the accuracy of the classifier on the small training set and on the (full) validation set. Print out the maximum validation accuracy, and the corresponding values of training accuracy and regularization parameter. Print the accuracies to at least 4 significant digits. You should find that the maximum validation accuracy is about 0.24, which is significantly greater than in part (a), but still not great.

In addition, plot training accuracy v.s. regularization parameter as a blue curve, and plot validation accuracy v.s. regularization parameter as a red curve. Plot both curves on a single pair of axes. Use `semilogx` to put a log scale on the horizongtal axis. Title the figure, *Question 2(b): Training and Validation Accuracy for Regularized QDA*. Label the horizontal axis, *Regularization parameter*, and label the vertical axis, *Accuracy*. Figure 5 at the end of the assignment shows what the graph should look like for the debugging data set of 300 MNIST points. Your program may use 1 loop, but no nested loops.

(c) (4 points total) For what values of the regularization parameter in part (b) does overfitting occur? (1 point) How is this indicated in the figure? (1 point) For what values of the regularization parameter does underfitting occur? (1 point) How is this indicated in the figure? (1 point)

ANSWER: As shown in part (b), the validation accuracy is maximal when the value of the regularization parameter is 0.00390625, which corresponds to the peak of the red curve in the graph. Below this value, the figure shows that the training accuracy gets increasingly big while the test accuracy gets increasingly small, which indicates overfitting. Above this value, the training and test accuracy both get increasingly small, which indicates underfitting.

(d) (10 points) In this question, you will try to avoid overfitting by using PCA to reduce the dimensionality of the MNIST data before applying QDA. Recall that for data of dimensionality M, the covariance matrix has dimensions $M \times M$, and thus has M^2 entries. If we can reduce M by a factor of 10, say, then we can reduce the number of parameters that have to be learned by a factor of $10^2 = 100$, which could significantly alleviate overfitting. The trick is to find the best dimension for the reduced data. You shall use the validation data for this purpose.

Suppose `X` is a data matrix and `T` is a vector of target values. Using the PCA and QDA classes in sklearn, define a Python function `train2d(K,X,T)` that uses PCA to reduce the data in `X` to `K` dimensions, and then trains a QDA classifier on the reduced data. Use `svd_solver='full'` with PCA. Your function should return the PCA instance, the QDA instance, and the training accuracy. You may use whatever methods of the PCA and QDA classes that you like. Note that if `X` has dimensions NxM, then the training data for QDA has dimensions NxK. Also note that this function does not project the reduced data back to the original space, as in Question 1, so it does not use the `inverse_transform` method of the PCA class.

In addition, suppose that `pca` and `qda` are instances returned by `train2d`. Together, they define a classifier. Define a Python function `test2d(pca,qda,X,T)` that computes and returns the accuracy of this classifier on the data `X,T`. The function should use the `pca` instance to reduce the dimensionality of the data in `X`, then compute the accuracy of the `qda` instance on the reduced data. Note that `test2d` only does testing and does not retrain `pca` or `qda`.

Using `train2d`, train classifiers on the small MNIST training set for `K` = 1,2,...,50. Using `test2d`, compute the accuracy of these classifiers on the MNIST validation data. Print out the maximum validation accuracy, and the corresponding values of `K` and training accuracy. Print the accuracies to at least 4 significant digits. You should find that the maximum validation accuracy is about 0.69, which is much greater than in part (b).

In addition, plot training accuracy v.s. `K` as a blue curve, and plot validation accuracy v.s. `K` as a red curve. Plot both curves on a single pair of axes, using the `plot` function. Title the figure, *Question 2(d): Training and Validation Accuracy for PCA + QDA*. Label the horizontal axis, *Reduced dimension*, and label the vertical axis, *Accuracy*. Figure 6 at the end of the assignment shows what the graph should look like for the debugging data set of 300 MNIST points.

Notice that above about `K` $= 30$, the training accuracy flattens out at a value of 1, which indicates extreme over fitting (though not nearly as extreme as in part (a)). At this point, the entire system is highly sensitive to even tiny differences in numerical error, which is why the validation accuracy is erratic. There can also be noticeable differences between machines, so your graph of validation accuracy may deviate slightly from the one in Figure 6 in this region. (But the graph of training accuracy should be flat at a constant value of 1 in this region.)

Your program may use 1 loop, but no nested loops.

(e) (6 points total) For what values of `K` in part (d) does overfitting occur? (1 point) How is this indicated in the figure? (1 point) For what values of `K` does underfitting occur? (1 point) How is this indicated in the figure? (1 point) You should find that the validation accuracy is low but somewhat erratic in the right half of the curve. Explain what you think is going on here. (2 points, for going beyond the explanation at the end of part (d))

ANSWER: As shown in part (d), the validation accuracy is maximal when the data is reduced to `K` $= 8$ dimensions, which corresponds to the peak of the red curve in the graph. Above this value, the figure shows that the training accuracy gets increasingly big while the test accuracy gets increasingly small, which indicates overfitting. Below this value, the training and test accuracy both get increasingly small, which indicates underfitting.

In QDA, predictions are made by using Bayes rule to compute posterior probabilities. This in turn requires computing class-conditional probabilities. As described in slide 42 of Lecture 7, in QDA, the class-conditional probabilities of a K-dimensional point, x, are given by

$$P(x|k) = \frac{e^{-(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k)/2}}{(2\pi)^{K/2}|\Sigma_k|^{1/2}}$$

where Σ_k is the covariance matrix for class k, Σ_k^{-1} is its inverse, and $|\Sigma_k|$ is its determinant. For this to be well-defined, the covariance matrix must be non-singular, so that it's inverse exists and its determinant (which we divide by) is non-zero. If the covariance matrix is singular, we can expect things to be unpredictable. Even if it is almost singular, things will be erratic, since the computed probabilities will be very sensitive to numerical error.

This will happen if the covariance matrix is estimated by too few data points. In particular, if a $K \times K$ covariance matrix is estimated by fewer than K data points, then it will be singular.[2] Since we have 10 classes and 200 data points, there are about 20 data points per class, *i.e.*, 20 points for estimating each covariance matrix. Thus, if $K > 20$, there will be too few points to estimate the covariance matrices, and they will be singular. This is precisely the region of

[2]Intuitively, K K-dimensional points are needed to specify a $K \times K$ matrix.

the graph, where the test accuracy becomes erratic and the training accuracy is constantly 1, *i.e.*, where extreme over-fitting occurs.

(f) (8 points) In this question you will combine the approaches of parts (b) and (d) to reduce overfitting even more. In particular, write a Python program that finds a combination of dimensionality reduction and regularization that maximizes validation accuracy. You should consider the same values of `reg_param` as in part (b), and the same values of `K` as in part (d). For all possible combinations of these values, you should train a regularized QDA classifier on reduced data from the small training set, and you should compute the validation accuracy of this classifier. Let `accMax` be the maximum validation accuracy over all combinations of values of `reg_param` and `K`. Print out `accMax`, as well as the corresponding values of training accuracy, `reg_param` and `K`. Print the accuracies to at least 4 significant digits. You should find that the maximum validation accuracy is about 0.85, which is much higher than in part (d).

In addition, for each value of `K`, let `accMaxK` be the maximum validation accuracy for all values of `reg_param`. Note that unlike `accMax`, the value of `accMaxK` depends on `K`. Plot `accMaxK` as a function of `K`. Title the figure, *Question 2(f): Maximum validation accuracy for QDA*. Label the horizontal axis, *Reduced dimension*, and label the vertical axis, *maximum accuracy.* Figure 7 at the end of the assignment shows what the graph should look like for the debugging data set of 300 MNIST points.

Your program may use one doubly-nested loop. In addition, your program should not use the functions `train2d` and `test2d` from part (d), as this would do a lot of redundant dimensionality reduction. Your program should avoid this redundancy.

3. *Bagging.* (47 points total)

This question builds on Question 2, augmenting the techniques developed there with bagging, which can reduce overfitting and increase accuracy even further. You will use the same MNIST data sets as Question 2, including the small training set. Because bagging is inherently random, your results may vary slightly from run to run.

In the questions below, you will be using some Python loops, since they are inherent to bagging, but use them only when specified. You will be using some functions from `sklearn`, but only those that are specified. The point is to implement everything else yourself from scratch. All code should be vectorized.

(a) (5 points) Define a Python function `myBootstrap(X,T)` that returns a bootstrap sample of the data matrix `X` and target values `T`. Only return bootstrap samples that contain at least 3 training points for each class (*i.e.*, at least three examples of each digit.) Without at least three points, QDA will produce erratic estimates of the covariance matrix of a digit, and without at least two points, it will raise an error, since it is impossible to estimate the covariance. Use the function `resample` in `sklearn.utils` to generate an arbitrary bootstrap sample. The sample should be drawn with replacement and should have the same size as `X,T` (*i.e.*, the matrix dimensions should be the same). Test the sample to make sure it has at least three points from each class. If not, use `resample` to generate another bootstrap sample, and test it. Repeat this procees until the bootstrap sample has at least 3 points in each class, and return this sample. Your program may, of course, use one loop (but no nested loops). Hint: create a one-hot encoding of `T` or use the function `numpy.unique`.

Note: in the vast majority of cases, the first bootstrap sample wil be returned, but occasionally, a second iteration will be required, and very rarely, a third. However, you will be generating so many bootstrap samples below, that you will inevitably generate some bootstrap samples that do not have at least two or three examples of some digit, which can cause your programs to fail. Hence the need for this function.

(b) (5 points) In this question, you will write a Python program that creates a QDA classifier for MNIST data, and bootstraps it 50 times. You may use any `sklearn` methods that come with your QDA classifiers.

First, use `sklearn` to create a QDA classifier, and set the regularization parameter to 0.004. Fit the classifier to the small MNIST training set. Compute and print out the validation accuracy of this classifier. We shall call this the validation accuracy of the base classifier.

Next, use the function `myBootstrap` from part (a) to create a bootstrap sample of the small training set. Create another QDA classifier with the same value of the regularization parameter, and fit it to the bootstrap sample. Use the `sklearn` method `predict_proba` to compute the predicted probabilities of the classifier on the validation data. Note that the input to this method is a data matrix, and the output is a matrix of probabilities. If the data matrix has dimensions $N \times M$, then the probability matrix has dimensions $N \times K$, where K is the number of classes.

Repeat this process 50 times, generating 50 different QDA classifiers and 50 different probability matrices. Compute the average of the probability matrices, and use this average matrix to create a vector of predictions, just as you did in Assignment 2. Note that if the validation data matrix has dimensions $N \times M$, then the prediction vector has dimension N. Compute and print out the accuracy of these predictions. We call this the validation accuracy of the bagged classifier, and we say that the bagged classifier is the average of 50 different QDA classifiers.

You should find that the bagged classifier has a validation accuracy that is 2-3 times greater than the base classifier. You should also find that each of the 50 iterations takes less than 1 second, so the entire process completes in under a minute, depending on your computer. Your program may, of course, use one loop (but no nested loops).

(c) (5 points) This question is similar to part (b) except that you will create 500 bootstrap samples, and you will monitor how the validation accuracy of the bagged classifier increases as the number of bootstrap samples increases. Of course, you will create the 500 QDA classifiers one after another in a loop. After each new one is created, compute the validation accuracy of the bagged classifier that is the average of all the QDA classifiers created so far. Thus, after creating 500 bootstrap samples and 500 QDA classifiers, you will have 500 estimates of validation accuracy.

Generate a plot of validation accuracy v.s. number of bootstrap samples. Title the figure, *Question 3(c): Validation accuracy.* Label the horizontal axis, *Number of bootstrap samples*, and label the vertical axis, *Accuracy*. Figure 8 at the end of the assignment shows approximately what the graph should look like for the debugging data set of 300 MNIST points.

In addition, create a second version of the plot using a log scale on the horizontal axis. Label the figure, *Question 3(c): Validation accuracy (log scale)*, and label the axes appropriately. The second plot should show clear increases in accuracy even when the first plot begins to flatten out.

Your program may, of course, use one loop (but no nested loops).

(d) (5 points) Define a Python function `train3d(K,R,X,T)` that uses both PCA and QDA to train a classifier on data `X,T`. Specifically, the function should use PCA to reduce the data in `X` to `K` dimensions, and then fit a QDA classifier to the reduced data using `reg_param=R`. This function is simlar to the function `train2d` in Question 2(d), but does not compute accuracy. Instead, it simply returns the PCA and QDA instances. You may use any `sklearn` methods that come with the PCA and QDA instances. Do not use any loops.

Also define a Python function `proba3d(pca,qda,X)` that uses the PCA and QDA instances returned by `train3d` to compute predicted probabilities for the data in `X`. The function should first use the PCA instance to reduce the dimensionality of the data. Then use the `predict_proba` method of the QDA instance to compute probabilities from the reduced data. Note that `proba3d` only computes probabilities and does not retrain `pca` or `qda`. Do not use any loops.

(e) (5 points total for parts (e) and (f))) Define a Python function `myBag(K,R)` that uses PCA and QDA to create classifiers for MNIST data, and bootstraps them 200 times. Assume the MNIST data is stored in global variables, using your choice of variable names. The function is similar to your program in part (b), but includes dimensionality reduction, as in Question 2. Here, `K` and `R` specify the dimension of the reduced data and the regularization parameter for QDA, respectively. They are used whenever the function `train3d` is called. You may use any methods in `sklearn` that come with PCA and QDA.

First, use the function `train3d` to create a classifier and fit it to the small MNIST training set. Use the returned PCA instance to reduce the dimensionality of the validation data, and then compute the accuracy of the QDA classifier on the reduced data. We shall call this the validation accuracy of the base classifier.

Next, use the function `myBootstrap` to generate a bootstrap sample of the small MNIST training set. Use `train3d` to create another classifier and fit it to the bootstrap sample. Then use `proba3d` to compute a matrix of predicted probabilities from the MNIST validation data.

Repeat this process 200 times, creating 200 classifiers and 200 probabiity matrices. Compute the average of the probability matrices, and use it to make predictions and compute accuracy, as in part (b). We shall call this the validation accuracy of the bagged classifier.

Return the validation accuracies of the base and bagged classifiers. Your program may, of course, use one loop (but no nested loops).

(f) Call `myBag` with `K` =100 and `R` = 0.01, and print both returned accuracies. You should find that the validation accuracy of the bagged classifier is a bit more than twice that of the base classifier. Also, the function should take about a minute to execute, depending on your machine.

(g) (5 points) Write a Python program that calls `myBag(K,R)` 50 times using random values of `K` and `R`. Possible values of `K` should be uniformly distributed between 1 and 10, inclusive, and possible values of `R` should uniformly distributed between 0.2 and 1.0. This will result in 50 values of validation accuracy for the base classifier, and 50 corresponding values of validation accuracy for the bagged classifier. Using blue dots, draw a scatter plot of bagged validation accuracy v.s. base validation accuracy. Title the figure, *Question 3(g): Bagged v.s. base validation accuracy.* Label the horizontal axis, *Base validation accuracy*, and label the vertical axis, *Bagged validation accuracy.* Set the lower and upper limits on both axes to 0 and 1, respectively. Finally, draw a diagonal red line across the plot from the point (0,0) to (1,1). You should find that the blue dots lie close to and slightly above the red line. Figure 9 at the end of the assignment shows approximately what the graph should look like for the debugging data set of 300 MNIST points.

Your program may use one loop, but no nested loops. You may find the functions `rand` and `randint` in `numpy.random` useful.

(h) (5 points) Repeat part (g), but use values of `K` uniformly distributed between 50 and 200, inclusive, and values of `R` uniformly distributed between 0 and 0.05. Title the figure and label the axes appropriately. Set the lower and upper limits of the y axis to be 0 and 1, respectively. Do not set any limits on the x axis. Instead of a diagonal line, draw a red horizonal line whose height is the maximum value of the bagged validation accuracy. You should find that the blue dots all lie fairly close to the red horizontal line. Print out the maximum bagged validation accuracy. It should be about 0.85, like the maximum validation accuracy in Question 2(f). Figure 10 at the end of the assignment shows approximately what the graph should look like for the debugging data set of 300 MNIST points.

(i) (12 points total) Interpret and explain the results in parts (g) and (h). Your answer should address the following questions:

- (5 points) What is the significance of the red diagonal line in the figure of part (g)? Why are the blue dots close to the diagonal red line? Why do they occur above the line, but not beneath it? Why does this happen for the values of `K` and `R` given in part (g).

  ANSWER: The red diagonal line indicates when bagging has no effect. Any point that lies on the red diagonal line represents a bagged classifier that has the same accuarcy as the base classifier, meaning that bagging did not increase the accuracy of the classifier. When a point lies above the red line, the bagged classifier has greater accuracy than the base classifier. The points in part (g) lie on or just above the red diagonal line, meaning that bagging had little or no effect. They do not lie below the red line because bagging does not decrease the accuracy of a classifier.
  The values of `K` and `R` used in part (g) cause substantial underfitting. In particular, the values of `K` are small, near or below the optimal value of 8 found in Question 2(d). And the values of `R` are large, much larger than the optimal value of 0.0039 found in Question 2(b). By themselves, these values of `R` cause underfitting. When, in addition, `K` is reduced from 784 to 10 or less, we get substantial underfitting.
  Substantial underfitting means that the decision boundary is smooth and stable, and does not change much from one bootstrap sample to another. More generally, the output probabilities do not change much from one bootstrap sample to another. This means that for any given input point, x, the average output probability over many bootstrap samples is about the same as the output probability of the base classifier. This means that bootstrapping has little, if any, effect.

- (5 points) The figure in part (h) implies that regardless of the accuracy of the base classifier, the bagged classifiers all have the same accuracy. How is this possible? Why does it happen for the values of `K` and `R` given in part (h). Why doesn't it happen in part (g)? Hint: See the slides for Tutorial 11.

  ANSWER: The values of `K` and `R` used in part (h) cause substantial overfitting. In particular, the values of `K` are large, much larger than the optimal value of 18 found in Question 2(f). And the values of `R` are small, much smaller than the optimal value of 0.25 found in Question 2(f).
  Substantial overfitting means the decision boundary is complex and erratic, and varies randomly from one bootstrap sample to the next. More generally, the output probabilities vary randomly from one bootstrap sample to the next. This means that for any given input point, x, the average output prob-

ability is an average of many samples of a random variable. These are exactly the conditions under which bagging does well, so we expect the accuarcy of the bagged classifiers to be much greater than that of the base classifiers.
Moreover, with enough bootstrap samples, the accuracy of a bagged classifier should converge on the Bayes optimal accuracy, regardless of the accuracy of the base classifier. (See the slides for Tutorial 11.) This is why the blue dots in the figure for part (h) all have a height of about 0.85, regardless of the base accuracy: after 200 bootstrap samples, the bagged classifiers are converging on the Bayes optimal accuracy, which is the best they can achieve.
This does not happen in part (g) because bagging is ineffective there, because of underfitting. Because of this, the accuracy of the bagged classifiers is barely better than that of the base classifiers, which puts them nowhere near the Bayes optimal of 0.85.

- (2 points) The maximum validation accuracy of the bagged classifier in part (h) is the same as the maximum validation accuracy of the classifier in Question 2(f). Why is this not a coincidence?

  ANSWER: The classifier in Question 2(f) searches for an optimal combination of `K` and `R`. It has apparently found a combination that achieves the Bayes optimal accuarcy, about 0.85, the best it can hope to achieve.

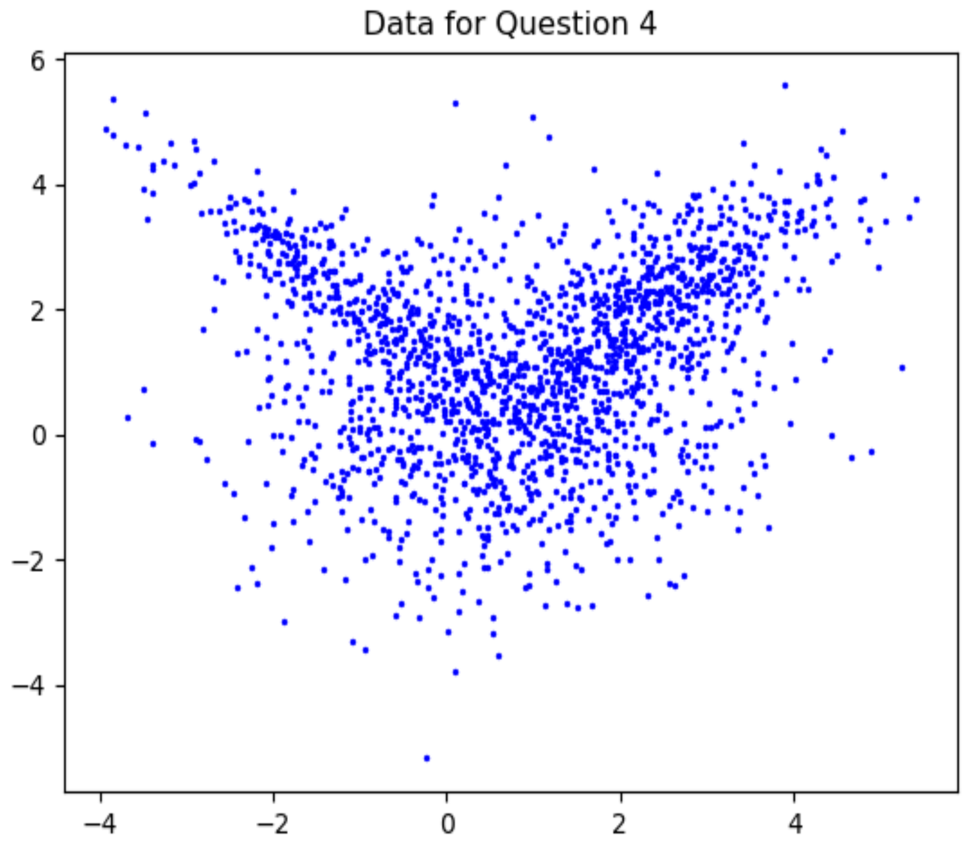
4. *Clustering.* (55 points total)

In this question, you will write Python programs to find the three clusters in the 2-dimensional data from Question 2 of Assignment 2. The first step, therefore, is to reload the data from the the file `dataA2Q2.pickle`. Because this is unsupervised learning, you will will ignore the labels — `Ttrain` and `Ttest` — and use only the input vectors — `Xtrain` and `Xtest`. Figure 1 shows what the unlabelled training data looks like. Your job is to find clusters in this data using (hard) K-means and Gaussian mixture models (GMM). In all cases, you will be looking for three clusters.

Although you will be testing your programs on 2-dimensional data with three clusters, your programs should work for data with any number of dimensions and clusters, unless specified otherwise. In the questions below, `X` is a NxM data matrix, `R` is a NxK matrix of soft assignments, and `Mu` is a KxM matrix of cluster centers. Here, N is the number of points in the data matrix, M is the number of features in each data point, and K is the number of clusters. Row k of `Mu` is the center of cluster k. Row n of `R` describes how much responsibility each cluster takes for the data point in row n of `X`. These are the responsibilities computed by the EM algorithm for a Gaussian mixture model. The entries of `R` are non-negative, and each row sums to 1. A one-hot encoding is a special case. A hard assignment of points to clusters (as computed by hard K-means) can thus be interpreted as responsibilities of 0 or 1.

In the questions below, you should not use any Python loops, unless specified otherwise. You will be using some functions from `sklearn`, but only those that are specified. The point is to implement everything else yourself from scratch. All code should be

Figure 1:



vectorized.

(a) *Data visualization.* (5 points)

Define a Python function `plot_clusters(X,R,Mu)` that displays the data points in `X` and the clusters defined by `R` and `Mu`. Here, `X` is a Nx2 data matrix, `R` is a Nx3 matrix of responsibilites, and `Mu` is a 3x2 matrix of cluster centers. Your function should use the following command to display the data points:

```
plt.scatter(X[:, 0], X[:, 1], color=R, s=5)
```

This will display each point as a dot of size of 5. More importantly, the color of each point will be a combination of the three primary colors, red, blue and green, depending on the responsibilty that each cluster takes for it. In particular, if (r_1, r_2, r_3) is the responsibility vector for a point, then the colors red, blue and green are given weights r_1, r_2 and r_3, respectively, when coloring the point. For example, if cluster 2 takes 100% responsibility for a point, then the point is colored blue. Likewise, if cluster 3 takes 100% responsibility for the point, then it is colored green. But, if clusters 2 and 3 each take 50% responsibilty, then the point is colored blue-green.

Before doing any of the above, first add up the total responsibility in each column of `R`. Then sort the columns of `R` in ascending order, so the column with least total responsibility is first. Use the sorted version of `R` above when calling scatter. This will ensure that the smallest cluster is always red, the largest is blue, and the medium one is green. You may find the numpy function `argsort` useful.

Finally, your function should make a second call to `scatter` to display each of the cluster centers as a black dot of default size on top of all the other points.

(b) *K means.* (4 points)

Use the Python class `KMeans` in `sklearn.cluster` to find three clusters in the training data. Then use the function `plot_clusters` to display the training data and the clusters. Title the figure, *Question 4(b): K means.* You may use any methods or attributes of the class that you find helpful. In addition, use the `score` method to compute (the negative of) the value of the K-means objective function on the training data and on the test data. Print both scores. If everything is working properly, both scores should be between -3000 and -4000, and the test score should be lower (more negative) than the training score.

Figure 11 at the end of the assignment shows a data set that is similar to, but slightly different from the one you are using. We shall call this the debugging data set. Figure 12 at the end of the assignment shows the result of running K means and `plot_clusters` on this debugging data set.

(c) *Gaussian mixture models.* (3 points)

Repeat part (b) using the Python class `GaussianMixture` in `sklearn.mixture` with `covariance_type='spherical'`. Title the figure, *Question 4(c): Gaussian mixture model (spherical).* Figure 13 at the end of the assignment shows what the figure looks like for the debugging data set. You may use any methods or attributes of the class that you find helpful. Note that in this case, the score method computes the average log-likelihood of the given data. If everything is working properly, both scores should be between -3 and -4, and the test score should be lower (more negative) than the training score.

(d) *Gaussian mixture models.* (3 points)

Repeat part (c) using the Python class `GaussianMixture` in `sklearn.mixture` with `covariance_type='full'`. Title the figure, *Question 4(d): Gaussian mixture model (full).* Figure 14 at the end of the assignment shows what the figure looks like for the debugging data set. You may use any methods or attributes of the class that you find helpful. Again, if everything is working properly, both scores should be between -3 and -4, and the test score should be lower (more negative) than the training score. Also, both scores should be higher (less negative) than the corresponding scores in part (c), implying a better fit to the data. Confirm this by printing the difference between the test scores, preceded by the string, `Q4d-Q4c test scores =`. The difference should be between 0 and 0.1.

(e) *Implementation of K means.* (10 points)

Define a Python function `myKmeans(X,K,I)` that searches for `K` clusters in data matrix `X` by performing `I` iterations of the (hard) K-means algorithm. Choose a reasonable way to initialize the cluster centers, and explain it clearly in the comments in your code. In addition, at the beginning of each iteration, compute and record the value of the K-means objective function, which we shall call the *score*. The function should return three values: (1) the cluster centers, `Mu`, (2) a one-hot encoding of the assignments, `R`, of points to clusters, and (3) a list of the computed scores.

Use `myKmeans` with 100 iterations to find 3 clusters in the training data. Use the returned values to generate two figures: a plot of score v.s. iteration number for the first 20 iterations, and a plot of the clustered training data, using `plot_clusters`. Title the first figure, *Question 4(e): score v.s. iteration (K means)*. Label the horizontal axis, *Iteration*, and the vertical axis, *Score*. Title the second figure, *Question 4(e): Data clustered by K means*, and do not label the axes.

If everything is working properly, the latter figure should look exactly the same as the corresponding figure in part (b). In the former figure, the score should decrease monotonically with iteration and eventually flatten out. If it increases anywhere, even a tiny bit, then there is a bug in your program. However, because the cluster centers are initilized randomly, this graph may look considerably different from one run to another, and may even wiggle widely. However, it should always be monotonic, and always flatten out at the end.

In addition, define a Python function `scoreKmeans(X,Mu)` that computes and returns the score of clusters defined by `Mu` on data matrix `X`. Use this function to compute the score of the clusters found by `myKmeans` on the training data and the test data. Print both scores. If everything is working properly, both scores should be between 3000 and 4000, and the test score should be higher than the training score. Also, both scores should be about the same as the corresponding scores in part (b), but negated. Confirm this by printing the sum of the test scores, preceded by the string, `Q4e+Q4b test scores =`. The sum should be between 2 and -2. (If not, run the programs again until the sum is this small, by random chance.)

For full marks, your code should have at most 1 loop (and no nested loops). Hint: You may find it easier to get everything working with a nested loop, and then use broadcasting to replace the nested loop with a simple loop.

(f) *Expectation Maximization for GMM.* (10 points)

Define a Python function `myGMM(X,K,I)` that searches for `K` clusters in data matrix `X` by performing `I` iterations of the EM algorithm for Gaussian mixture models. You should assume that the covariance matrix is the identity matrix, I, as in the lecture slides. Choose a reasonable way to initialize the cluster centers and the mixing proportions, and explain it clearly in the comments in your code. In addition, at the beginning of each iteration, compute and record the mean log likelihood of the data, which we shall call the *score*. For full marks, do not use any statistical functions (*e.g.*, in `scipy.stats`) to compute the score. Instead, you should use basic numpy operations, such as `sum`, `mean`, `exp` and `log`, to implement the formula you derive in part (g).

Your `myGMM` function should return four values: (1) the cluster centers, `Mu`, (2) the mixing proportions, π, (3) the responsibilities, `R`, and (4) a list of the computed scores.

Use `myGMM` with 100 iterations to find 3 clusters in the training data. Use the returned values to generate two figures: a plot of score v.s. iteration number for the first 20 iterations, and a plot of the clustered training data, using `plot_clusters`. Title the figures and label the axes appropriately.

Figure 15 at the end of the assignment shows what the latter figure looks like for the debugging data set. In the former figure, the score should increase monotonically with iteration and eventually flatten out. If it decreases anywhere, even a tiny bit, then there is a bug in your program. Because the cluster centers are initilized randomly, this graph may look considerably different from one run to another, and may even wiggle widely. However, it should always be monotonic, and always flatten out at the end.

In addition, define a Python function `scoreGMM(X,Mu,Pi)` that computes and returns the score of the clusters defined by `Mu` and `Pi` on data matrix `X`. Use this function to compute the score of the clusters found by `myGMM` on the training data and the test data. Print both scores. If everything is working properly, both scores should be between -3 and -4, and the test score should be lower (more negative) than the training score. In addition, both scores should be less (more negative) than the corresponding scores in part (c). Confirm this by printing the difference in the test scores, preceded by the string, `Q4c-Q4f test scores =`. The difference should be between 0 and 0.1.

For full marks, your code should have at most 1 loop (and no nested loops). Hint: You may find it easier to get everything working with a nested loop, and then use broadcasting to replace the nested loop with a simple loop.

(g) *Log likelihood.* (5 points)

The equation on slide 29 of Lecture 9 gives a formula for the log-likelihood of the data. If there are N points in the data set, then dividing this formula by N gives the mean log-likelihood, *i.e.*, the score used in part (f). Simplify this formula to one that can be programmed with basic numpy operations such as `sum`, `mean`, `exp` and `log`. The formula should have one summation sign for every occurance of `sum` or `mean` in your Python function `scoreGMM` in part (f).

Prove that your simplified formula is correct. It should work for any number of clusters and data of any dimensionality. Your proof should use the following definition, where x is a M-dimensional column vector, and $|\Sigma|$ is the determinant of the covariance matrix Σ:

$$\mathcal{N}(x|\mu,\Sigma) = \frac{e^{-(x-\mu)^T\Sigma^{-1}(x-\mu)/2}}{(2\pi)^{M/2}|\Sigma|^{1/2}} \tag{2}$$

ANSWER: First, recall the following properties of the identity matrix: $|I| = 1$ and $I^{-1} = I$. Thus,

$$\begin{aligned}
\mathcal{N}(x|\mu,I) &= \frac{\exp\left[-(x-\mu)^T I^{-1}(x-\mu)/2\right]}{(2\pi)^{M/2}|I|)^{1/2}} && \text{by Equation (2)} && (3)\\
&= \frac{\exp\left[-(x-\mu)^T(x-\mu)/2\right]}{(2\pi)^{M/2}} && \text{by the properties of } I && (4)\\
&= \frac{\exp\left[-\sum_i(x_i-\mu_i)^2/2\right]}{(2\pi)^{M/2}} && && (5)
\end{aligned}$$

since $y^Ty = \sum_i y_i^2$, for any column vector, y. Second, let $\mathcal{D}$ be a data set of N elements. Then the log-likelihood of the data set is given by

$$\begin{aligned}
\log p(\mathcal{D}) &= \sum_n \log\left[\sum_k \pi_k \mathcal{N}(x^{(n)}|\mu_k, I)\right] && \text{by slide 29 of Lecture 9}\\
&= \sum_n \log\left[\sum_k \pi_k \exp\left(-\sum_i [x_i^{(n)} - \mu_{ki}]^2/2\right)/(2\pi)^{M/2}\right] && \text{by Equation (5)}
\end{aligned}$$

The three summation signs in this formula correspond to the single use of `np.mean` and the two uses of `np.sum` in the Python function `scoreGMM` in the solution to part (f).

(h) (5 points) Give a mathematical proof that in K means, the boundary between two adjacent clusters must be linear.

ANSWER: Recall that if a and b are column vectors, then $\|a\|$ is the magnitude of a and $\|a\|^2 = a^Ta$. It easily follows that $\|a-b\|^2 = \|a\|^2 - 2a^Tb + \|b\|^2$. With this in mind, let c_1 and c_2 be the centers of two adjacent clusters, and let x be a point on the boundary between them. This means that x is equally far from c_1 and c_2, that is, $\|x - c_1\| = \|x - c_2\|$. This gives rise to the following series of equivalent statements:

$$\begin{aligned}
\|x-c_1\|^2 &= \|x-c_2\|^2\\
\|x\|^2 - 2x^Tc_1 + \|c_1\|^2 &= \|x\|^2 - 2x^Tc_2 + \|c_2\|^2\\
-2x^Tc_1 + \|c_1\|^2 &= -2x^Tc_2 + \|c_2\|^2\\
2x^T(c_2-c_1) + \|c_1\|^2 - \|c_2\|^2 &= 0\\
x^Tw + b &= 0
\end{aligned}$$

where $w = 2(c_2 - c_1)$ and $b = \|c_1\|^2 - \|c_2\|^2$. Thus, x is a boundary point if and only if it lies on the linear surface defined by w and b. Thus the boundary is linear.

(i) (5 points) Why are the scores in part (d) higher than those in part (c)? In particular, what is it about the data that causes this to happen? For what kind of data could the scores in part (c) be higher? For what kind of data would the scores be equal? Justify each of your answers.

Note that in part (d), each class, k, can have a different and arbitrary covariance matrix, Σ_k. This is what `sklearn` calls a *full* covariance matrix, because it is not limited in any way. In part (c), each class can also have a different covariance matrix, but of a special form. In particular, the covariance matrix for class k has the form $\sigma_k^2 I$, where I is the identify matrix, and each feature in the class has the same variance, σ_k^2. This is what `sklearn` calls a *spherical* covariance matrix.

ANSWER: The arguments below apply to scores on the training data. However, if the data sets are large enough to prevent overfitting, then they also apply to the test scores. This is certainly the case for the data set in Figure 1.

Expectation Maximization (EM) is an optimization algorithm that maximizes the log likelihood, and the main points in the explanations below rely on a simple fact about optimization: if we expand the set of possible parameters, then the maximum can only go up or stay the same. Formally, suppose we are trying to find the value of a parameter, λ, that maximizes a function, $f(\lambda)$. If S_1 and S_2 are sets of parameter values, and $S_1 \subseteq S_2$, then

$$\max_{\lambda \in S_2} f(\lambda) \;\geq\; \max_{\lambda \in S_1} f(\lambda) \tag{6}$$

If the maximum over S_2 happens to occur for a value of λ in S_1, then

$$\max_{\lambda \in S_2} f(\lambda) \;=\; \max_{\lambda \in S_1} f(\lambda) \tag{7}$$

If the maximum over S_2 does not occur for a value of λ in S_1, then the inequality becomes strict:

$$\max_{\lambda \in S_2} f(\lambda) \;>\; \max_{\lambda \in S_1} f(\lambda) \tag{8}$$

In our case, f is the log likelihood of the data set, and λ is not a single parameter, but all the parameters of the Gaussian mixture model (GMM), *i.e.*, the means, covariance matrices and mixing proportions of the three classes. Both S_1 and S_2 would include all possible means and mixing proportions. In addition, S_2 might include all possible covariance matrices (what `sklearn` calls `full` covariances matrices), while S_1 might be limited to the set of spherical covariance matrices. Note that as the log likelihood increases, the score (accuracy) also tends to increase, and we say that the parameters fit the data better.

In part (c) the covariance matrices are spherical, and in part (d) they are full. Since spherical matrices are a special case of full matrices, maximizing over spherical matrices will never give a better fit than maximizing over full matrices [Inequality (6)]. *i.e.*, there is no data set for which the scores in part (c) are higher than in (d).

However, spherical and full covariance matrices can sometimes give an equally good fit, and thus equal scores [Equation (7)]. In particular, for a data set in which the underlying clusters are spherical Gaussians, maximizing over full covariance matrices will simply converge on a spherical matrix, and thus give the same fit.

For the data set in this question, the underlying clusters are elliptical, not spherical (*i.e.*, circular in 2D), as can be seen in Figure 4 of Assignment 2. Thus, spherical covariance matrices will not provide as good a fit to the data as full covariance matrices, which can provide an optimal fit to elliptical clusters [Inequality (8)]. This is why the scores in part (d) are higher than in part (c).

(j) (5 points) Why are the scores in part (f) lower than those in part (c)? In particular, what is it about the data that causes this? For what kind of data could the scores in part (c) be lower? For what kind of data would the scores be equal? Justify each of your answers.

ANSWER: As in part (i), we assume that the data set is large enough to prevent over fitting, so that the arguments below apply to both training scores and test scores. The main arguments also rely on the three equations and inequalities given in the answer to part (i).

In part (c) the covariance matrices are spherical, while in part (f) they are the identity matrix. The identity matrix is a special kind of spherical matrix, so using the identity matrix will never give a better fit than maximizing over spherical matrices [Inequality (6)]. That is, there is no data set for which the scores in part (c) are lower than in part (f).

However, the identity matrix can sometimes give as good a fit as spherical matrices, and thus equal scores [Equation (7)]. The identify matrix describes a spherical Gaussian cluster with a standard deviation of 1 along each dimension. For a data set in which the underlying clusters have this form, maximizing over spherical matrices will simply converge to the identity matrix, and thus give the same fit.

For the data set in this question, the underlying clusters are elliptical, not spherical (*i.e.*, circular in 2D), as can be seen in Figure 4 of Assignment 2. However, maximizing over spherical matrices might still converge to the identiy matrix if the underlying clusters in the data set all had a variance of 1 along each dimension. However, it is apparent from Figure 4 in Assignment 2 that they do not, since the large red cluster is wider than the other two. Thus, the identity matrix will not provide as good a fit to the data as spherical matrices, which can provide a better fit to clusters of different sizes, and in particular, to clusters whose width is not 1 [Inequality (8)]. This is why the scores in part (f) are lower than in part (c).

164 points total

University of Toronto Mississauga
CSC 311 - Introduction to Machine Learning

Cover sheet for Assignment 3

Complete this page and hand it in with your assignment.

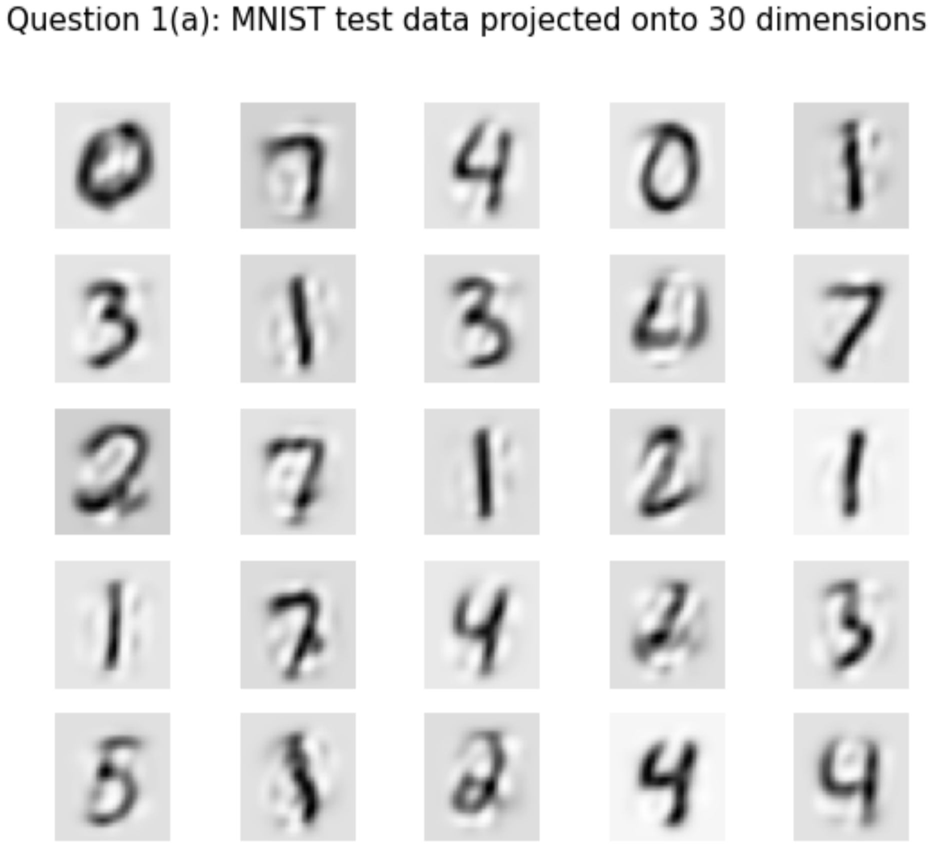
Name: __________________________________
(Underline your last name)

Student number: __________________________________

I declare that the solutions to Assignment 1 that I have handed in are solely my own work, and they are in accordance with the University of Toronto Code of Behavior on Academic Matters.

Signature: __________________________________

Figure 2:



The figures on this and subsequent pages are for debugging purposes. They are for data sets related to but different from the ones you are using. You should be able to duplicate these results, but they are *not* to be handed in. See the relevant questions for more details.

Figure 3:

Question 1(b): MNIST test data projected onto 3 dimensions

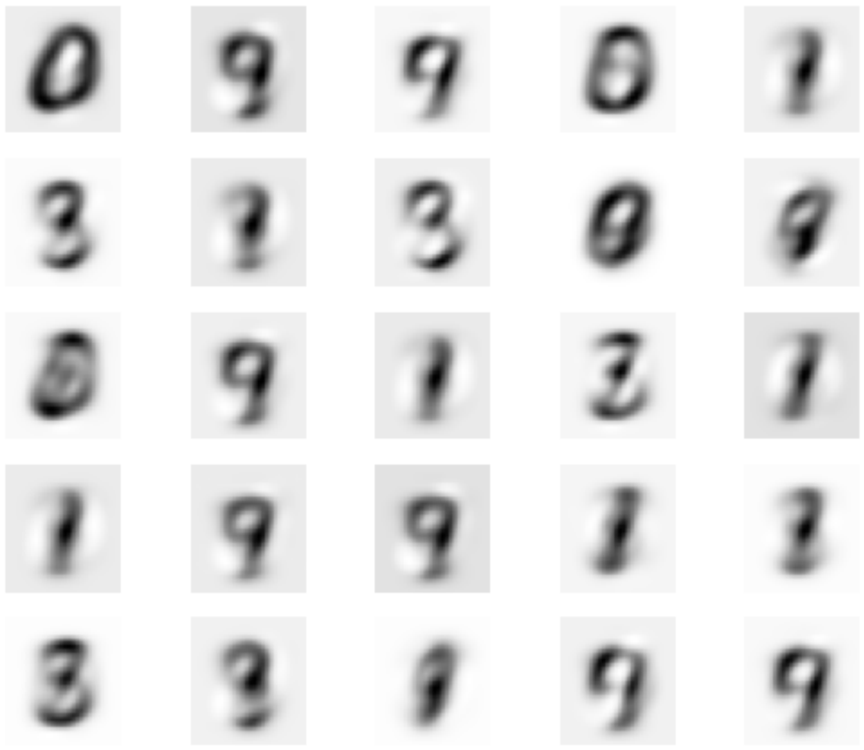


Figure 4:

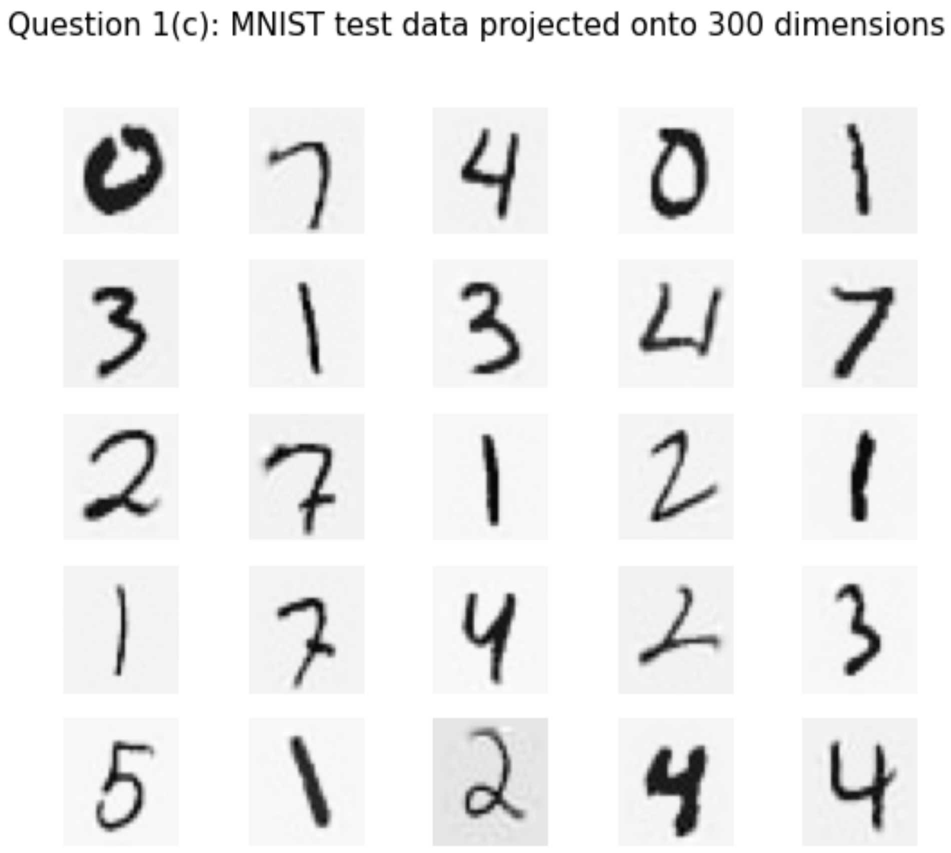

Question 1(c): MNIST test data projected onto 300 dimensions

Figure 5:

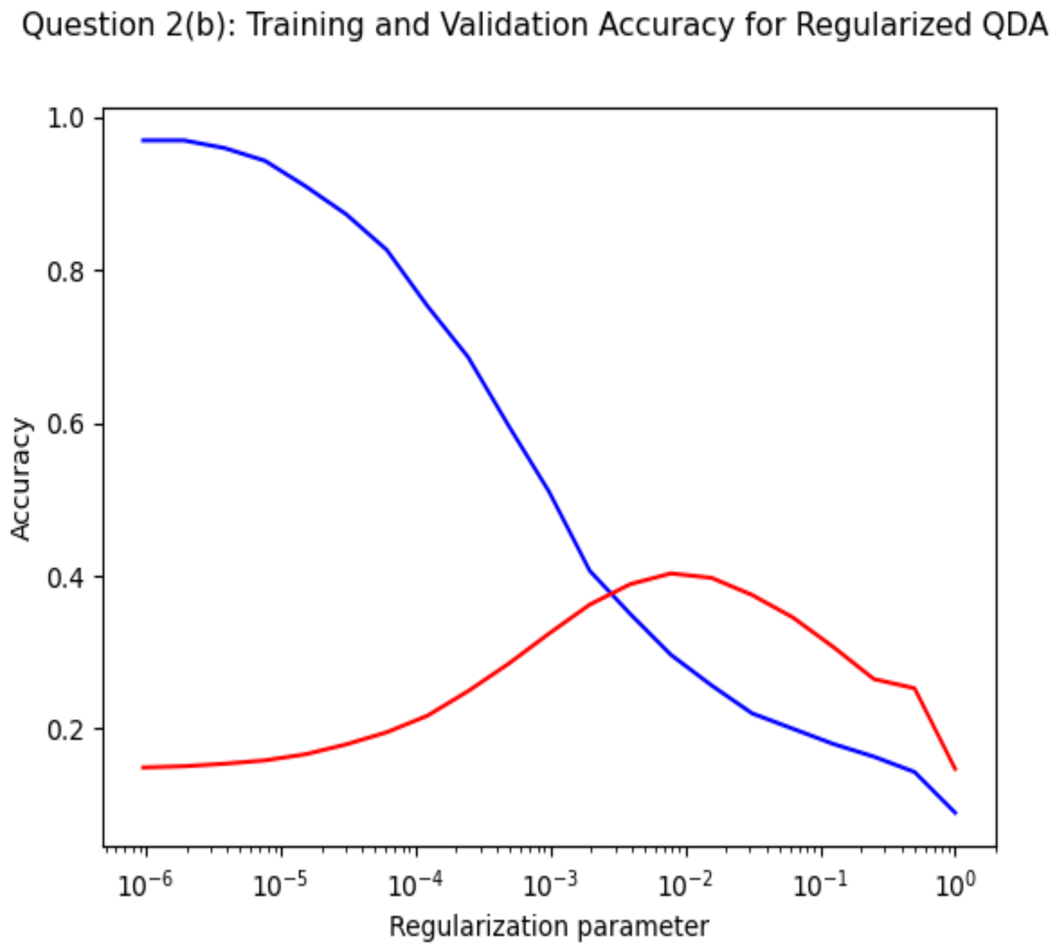

Question 2(b): Training and Validation Accuracy for Regularized QDA
Accuracy
Regularization parameter
1.0
0.8
0.6
0.4
0.2
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}

Figure 6:

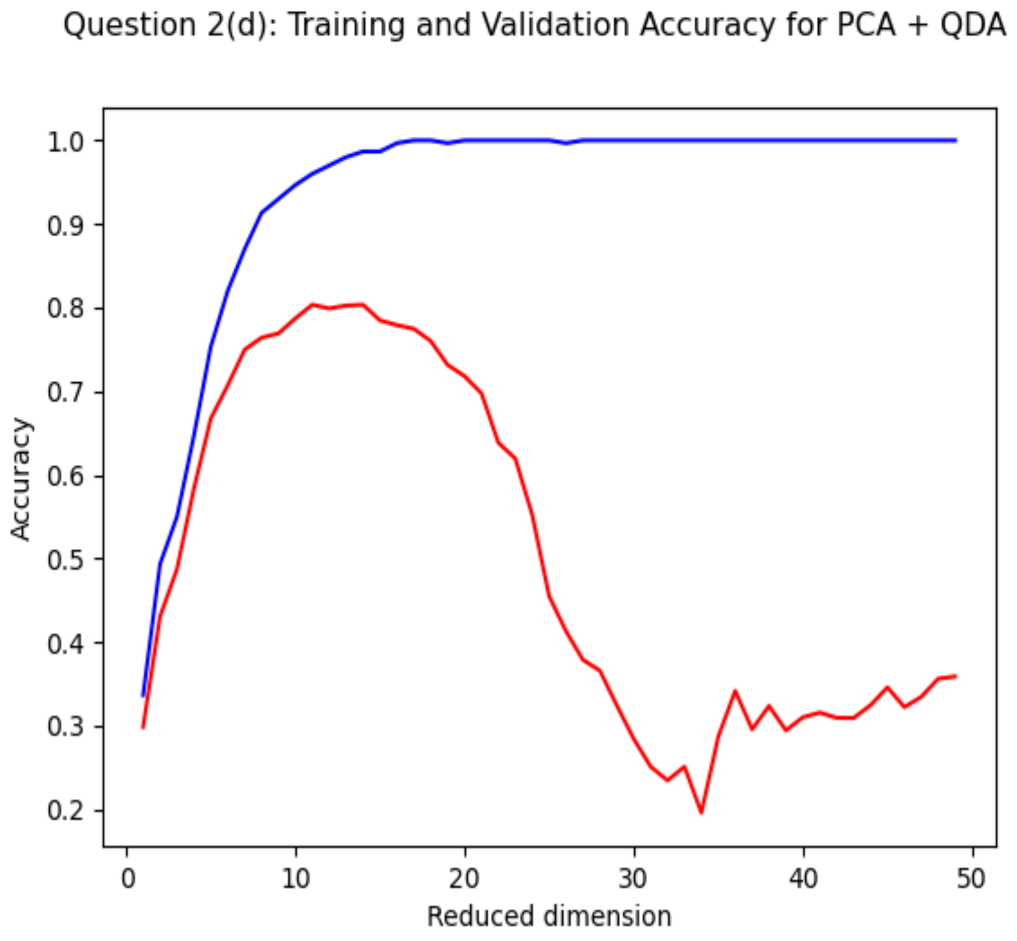

Question 2(d): Training and Validation Accuracy for PCA + QDA
Accuracy
Reduced dimension
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
10
20
30
40
50

Figure 7:

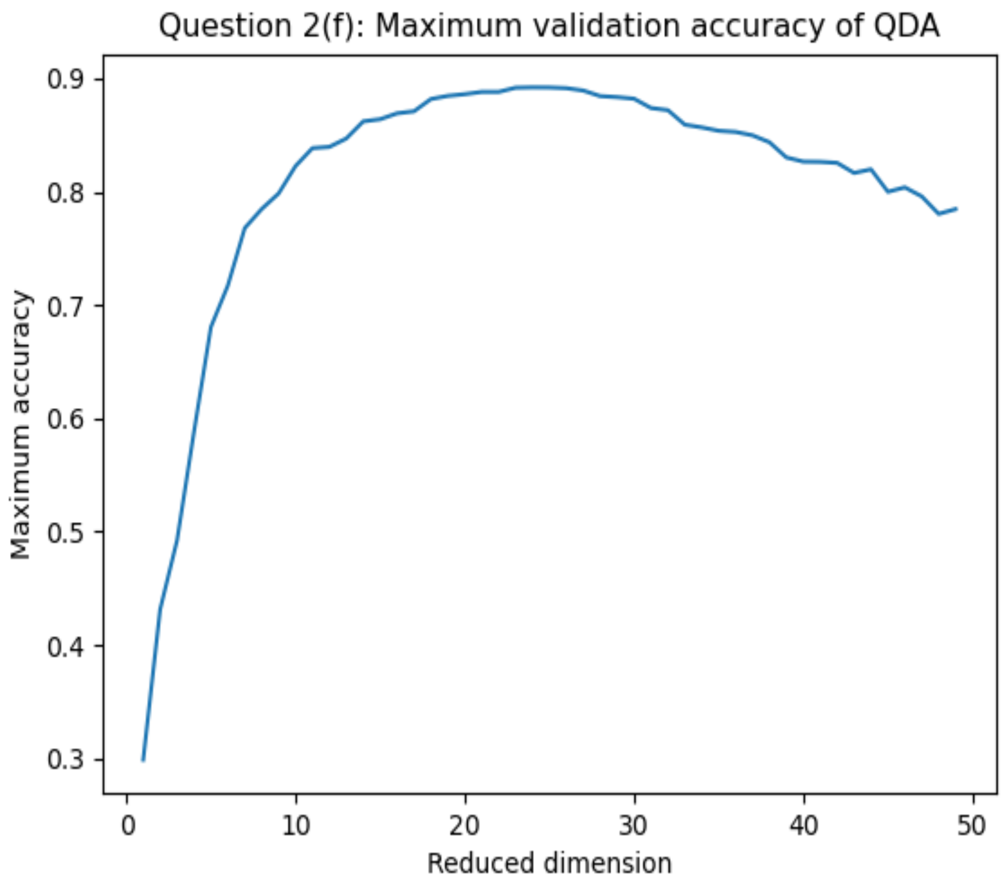

Question 2(f): Maximum validation accuracy of QDA
Maximum accuracy
Reduced dimension
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
10
20
30
40
50

Figure 8:

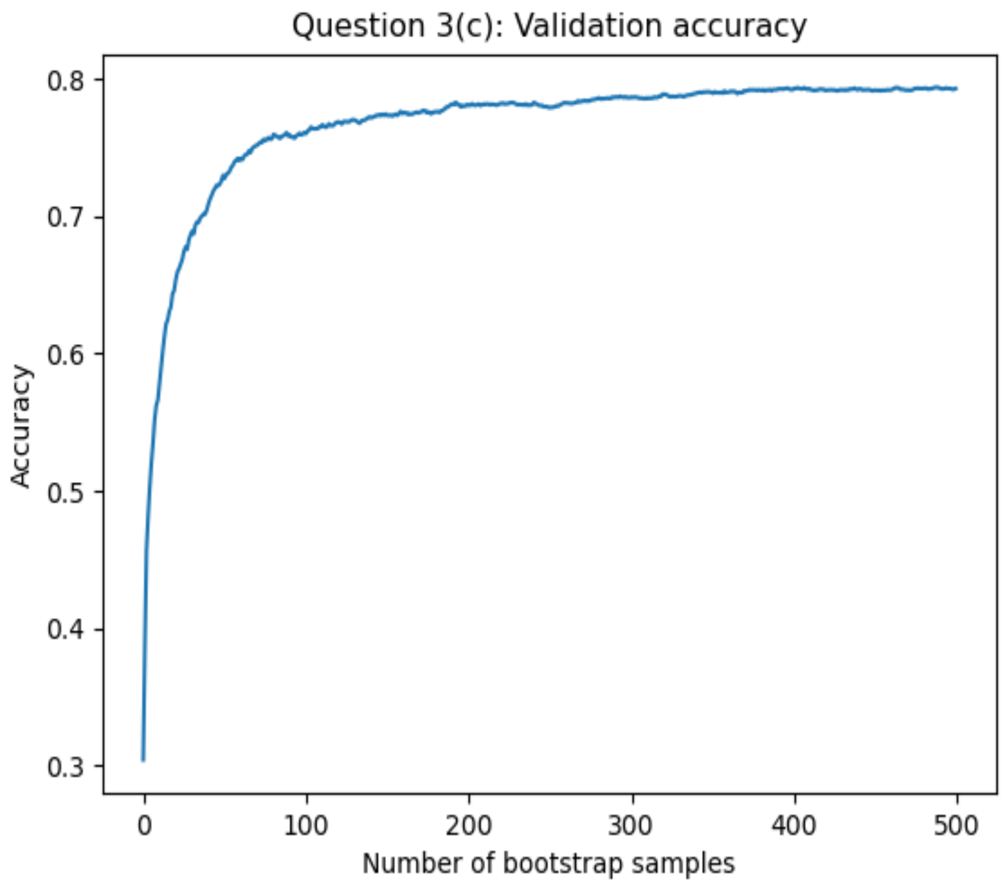

Question 3(c): Validation accuracy
Accuracy
0.3
0.4
0.5
0.6
0.7
0.8
0
100
200
300
400
500
Number of bootstrap samples

Figure 9:

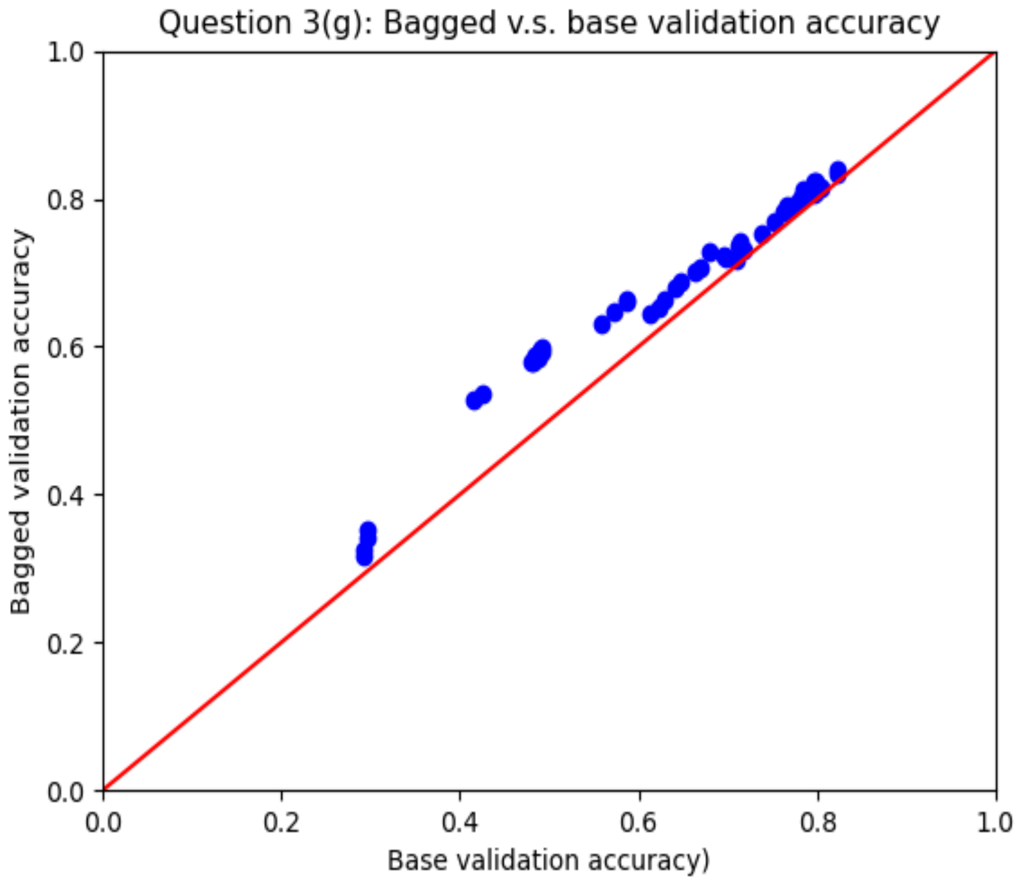

Question 3(g): Bagged v.s. base validation accuracy
Bagged validation accuracy
Base validation accuracy)
0.0
0.2
0.4
0.6
0.8
1.0

Figure 10:

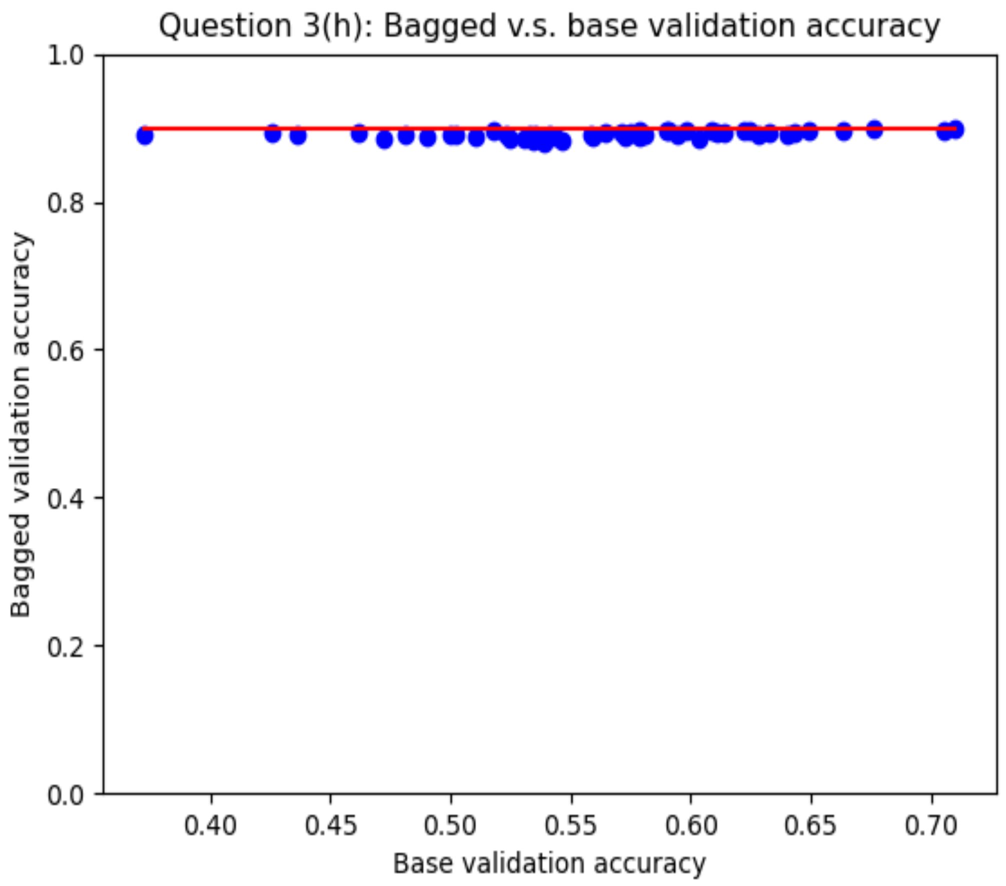

Question 3(h): Bagged v.s. base validation accuracy
1.0
0.8
0.6
0.4
0.2
0.0
Bagged validation accuracy
0.40
0.45
0.50
0.55
0.60
0.65
0.70
Base validation accuracy

Figure 11:

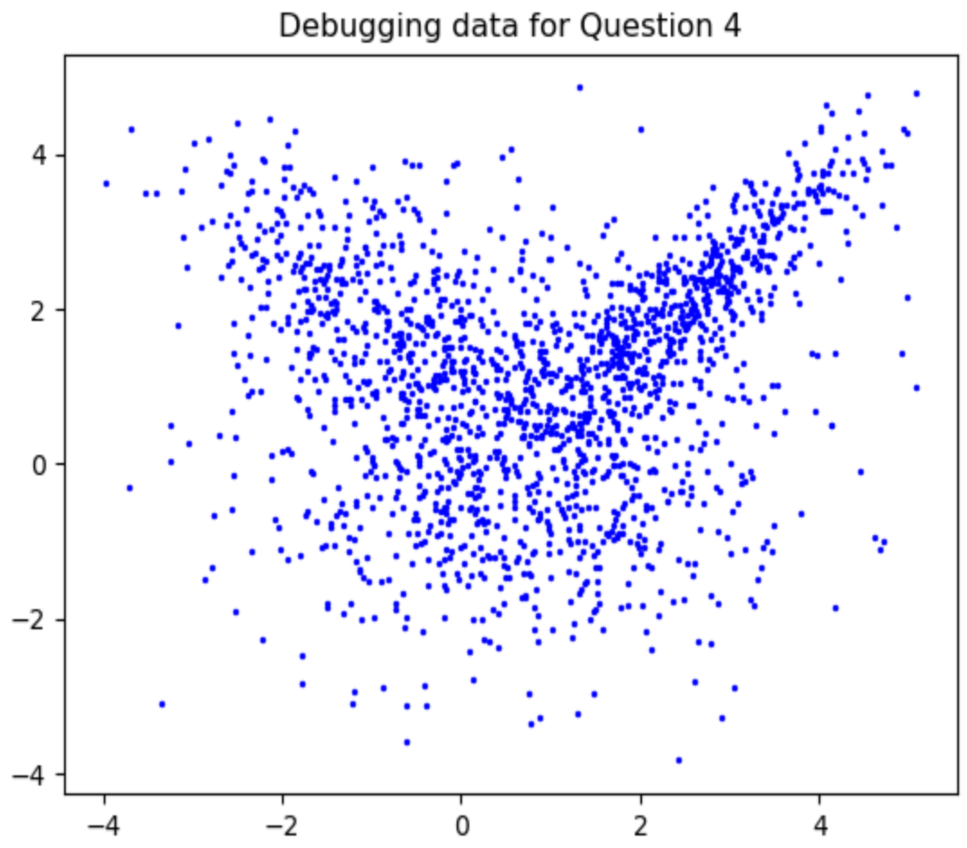


Figure 12:

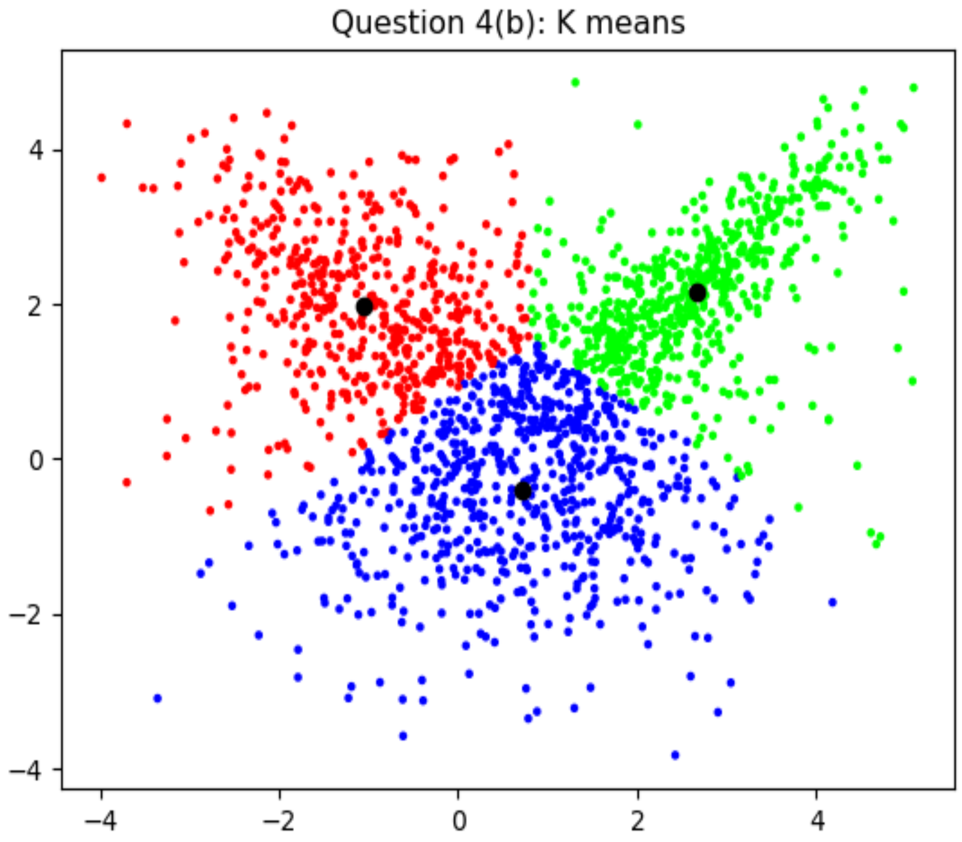


Figure 13:

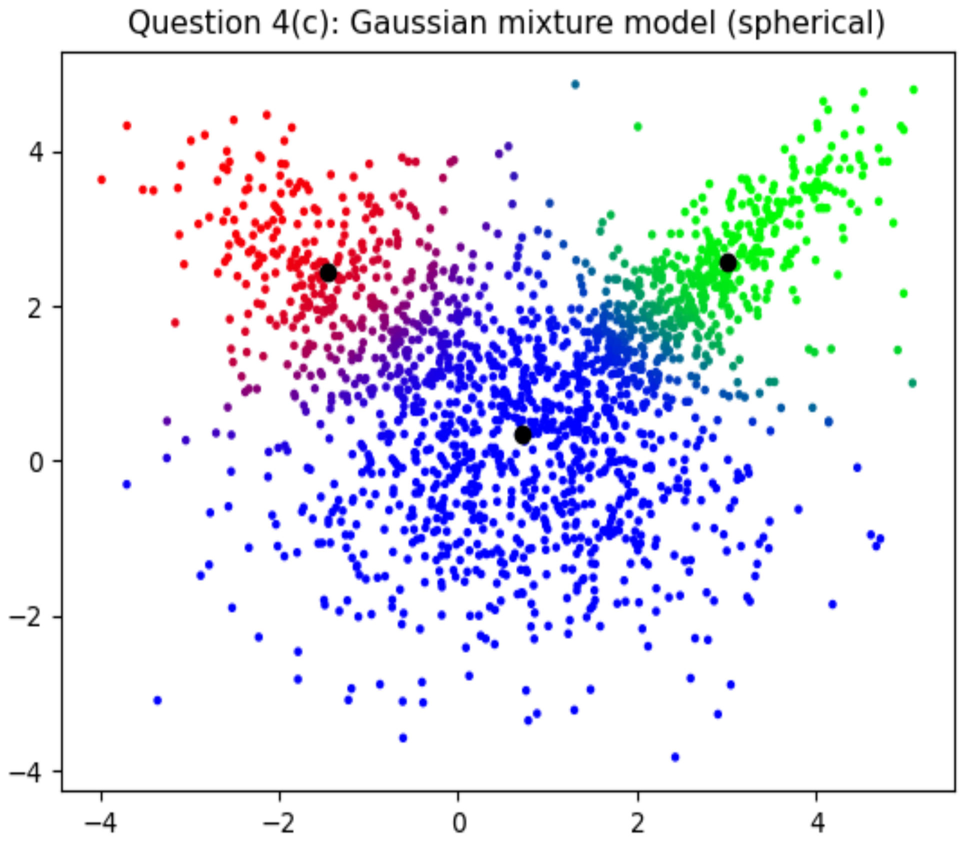


Figure 14:

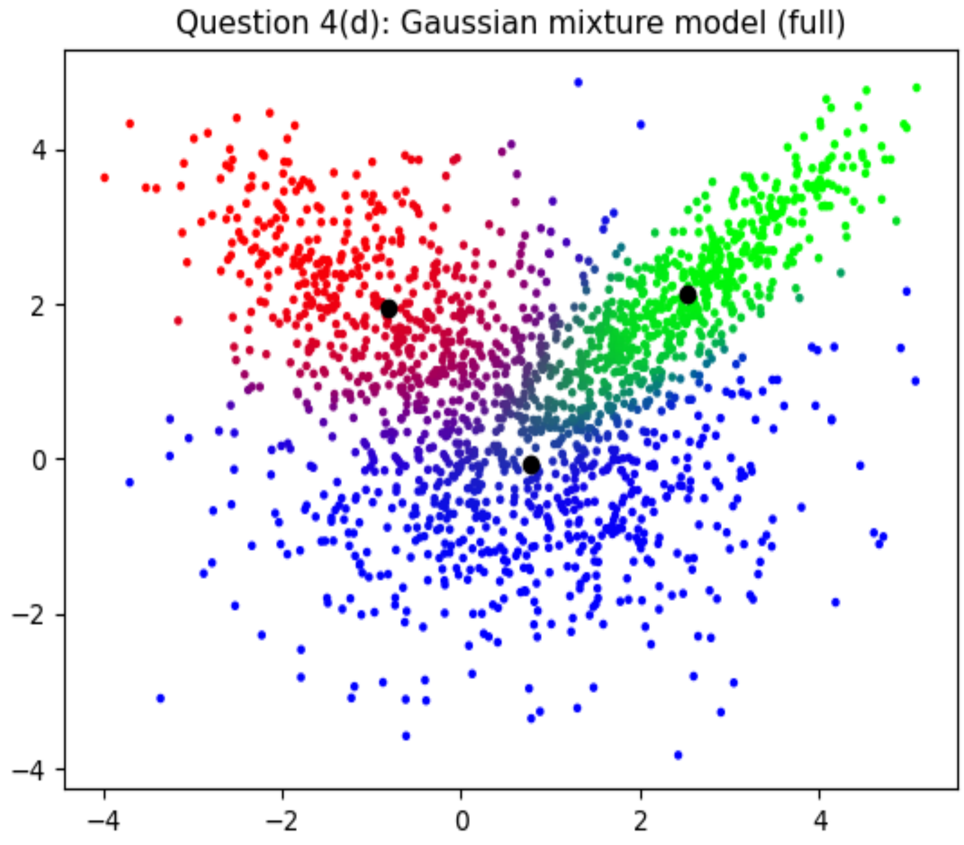
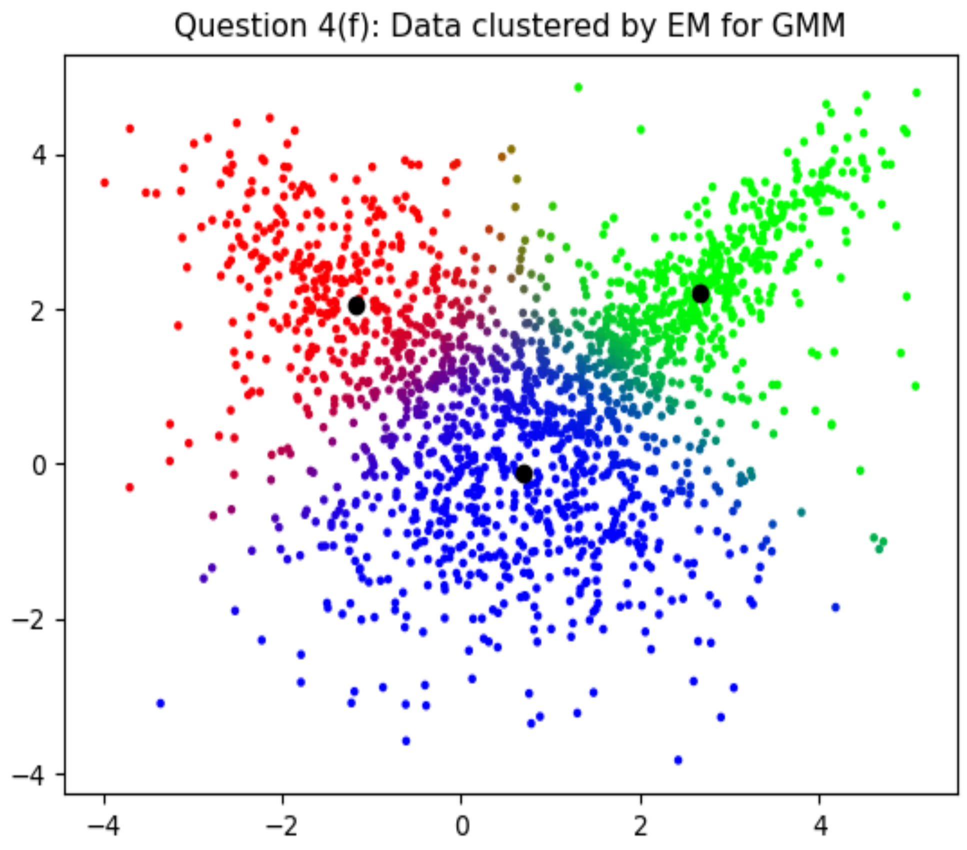


Figure 15:



```
###########  Programming Solutions for Assignment 2  ############

import numpy as np
import numpy.random as rnd
import numpy.linalg as la
import matplotlib.pyplot as plt
import pickle
from sklearn.utils.testing import ignore_warnings
import sklearn.decomposition as decomp
import sklearn.cluster as cluster
import sklearn.mixture as mixture
import sklearn.utils as utils
import sklearn.discriminant_analysis as da



##########  QUESTION 1  ############

print('\n\nQuestion 1.')
print('--------------')

# read MNIST data from a file
# Question 5(a)
# get the mnist data from a file
with open('mnistTVT.pickle','rb') as f:
    Xtrain,Ttrain,Xval,Tval,Xtest,Ttest = pickle.load(f)
Xtrain = Xtrain.astype(np.float64)
Xval = Xval.astype(np.float64)
Xtest = Xtest.astype(np.float64)


# Question 1(a)

# project the MNIST data on K dimensions.
# display projected images and print the RMS error.
def project(K):
    # reduce dimensionality
    pca = decomp.PCA(K)
    pca.fit(Xtrain)
    XtestR = pca.transform(Xtest)    # reduced test data.  shape = [N,K]
    XtestP = pca.inverse_transform(XtestR)    # reconstructed test data.  shape = [N,M]
    # output results
    displayImages(XtestP,5,5)
    err = np.sqrt(np.sum((Xtest-XtestP)**2)/len(Xtest))
    print('RMS error = ',err)
    return XtestP

# display the rows of data as images arranged in a LxW grid
def displayImages(data,L,W):
    plt.figure()
    N,M = np.shape(data)
    # number of rows and columns in each image
    m = int(np.sqrt(M))
    data = np.reshape(data,[N,m,m])
    for i in range(0,L*W):
        # display the ith image
        plt.subplot(L,W,i+1)
        plt.axis('off')
        plt.imshow(data[i], cmap='Greys')


print('\nQuestion 1(a):')
project(30)
plt.suptitle('Question 1(a): MNIST test data projected onto 30 dimensions')


# Question 1(b)
print('\nQuestion 1(b):')
project(3)
plt.suptitle('Question 1(b): MNIST test data projected onto 3 dimensions')


# Question 1(c)
print('\nQuestion 1(c):')
XtestP = project(300)
plt.suptitle('Question 1(c): MNIST test data projected onto 300 dimensions')




# Question 1(d)
# use PCA to project X onto a K-dimensional subspace.
# shape(X) = [N,M]
def myPCA(X,K):
    # compute mean and covariance
    mu = np.mean(X,axis=0)    # shape = [M]
    Xc = X - mu    # centered data.  shape = [N,M]
    cov = np.matmul(Xc.T,Xc)/len(X)    # shape = [M,M]
    # eigenvectors
    U = la.eigh(cov)[1]    # shape(U) = [M,M]
    U = U[:,-K:]    # shape = [M,K]
    # reduce dimensionality
    Xr = np.matmul(Xc,U)    # shape = [N,K]
    # reconstruct
    Xt = np.matmul(Xr,U.T)    # shape = [N,M]
    return Xt + mu


# Question 1(f)
# test myPCA
print('\nQuestion 1(f):')
K = 100
myXtrainP = myPCA(Xtrain,K)
displayImages(myXtrainP,5,5)
plt.suptitle('Question 1(f): MNIST data projected onto 100 dimensions (mine)')

pca = decomp.PCA(K,svd_solver='full')
XtrainR = pca.fit_transform(Xtrain)    # reduced data
XtrainP = pca.inverse_transform(XtrainR)    # reconstructed (projected) data
displayImages(myXtrainP,5,5)
plt.suptitle('Question 1(f): MNIST data projected onto 100 dimensions (sklearn)')

err = np.sqrt(np.sum((myXtrainP-XtrainP)**2)/len(Xtrain))
print('RMS error =',err)




##########  QUESTION 2  ############

print('\nQuestion 2.')
print('-------------')

# small training set
N = 200
XtrainS = Xtrain[:N]
TtrainS = Ttrain[:N]



# Question 2(a)
print('\nQuestion 2(a):')
# train an unregularized QDA classifier
clf = da.QuadraticDiscriminantAnalysis()
ignore_warnings(clf.fit)(XtrainS,TtrainS)
# compute and print accuracies
accTest = clf.score(Xtest,Ttest)
accTrain = clf.score(XtrainS,TtrainS)
print('Training accuracy =',accTrain)
print('Test accuracy =',accTest)



# Question 2(b)
# regularized QDA.
# find the best value of the regularization parameter for QDA
print('\nQuestion 2(b):')
accListVal = []
accListTrain = []
Rlist = 2.0**np.arange(-20,1)    # list of regularization parameters to test
accMax = 0
for R in Rlist:
    # train a QDA classifier with regularization parameter R
    clf = da.QuadraticDiscriminantAnalysis(reg_param=R)
    ignore_warnings(clf.fit)(XtrainS,TtrainS)
    # compute and record accuracies
    accVal = clf.score(Xval,Tval)
    accTrain = clf.score(XtrainS,TtrainS)
    accListVal.append(accVal)
    accListTrain.append(accTrain)
    # record the best results so far
    if accVal > accMax:
        accMax = accVal
        accTrainBest = accTrain
        Rbest = R
# print best results
print('Maximum validation accuracy =',accMax)
print('Corresponding training accuracy =',accTrainBest)
print('Best regularization parameter =',Rbest)
# plot accuracies
plt.figure()
plt.semilogx(Rlist,accListTrain,'b')
plt.semilogx(Rlist,accListVal,'r')
plt.suptitle('Question 2(b): Training and Validation Accuracy for Regularized QDA')
plt.xlabel('Regularization parameter')
plt.ylabel('Accuracy')




# Question 2(d)
# find the best amount of dimensionality reduction for unregularized QDA

print('\nQuestion 2(c):')
# train a PCA+QDA classifer on data X,T.
# use PCA to reduce X to K dimensions.  then use QDA
def train2d(K,X,T):
    pca = decomp.PCA(K,svd_solver='full')
    Xr = pca.fit_transform(X)    # reduced data
    qda = da.QuadraticDiscriminantAnalysis()
    ignore_warnings(qda.fit)(Xr,T)
    acc = qda.score(Xr,T)    # training accuracy
    return pca,qda,acc

# compute the accuracy a PCA+QDA classifier on data X,T.
def test2d(pca,qda,X,T):
    Xr = pca.transform(X)
    acc = qda.score(Xr,T)
    return acc


accListVal = []    # initial list of validation accuracy
accListTrain = []    # initial list of training accuracy
Klist = range(1,50)    # list of K values to test
accMax = 0
for K in Klist:
    # train and evaluate a PCA+QDA classifier
    pca,qda,accTrain = train2d(K,XtrainS,TtrainS)
    accVal = test2d(pca,qda,Xval,Tval)
    # record the accuracies
    accListVal.append(accVal)
    accListTrain.append(accTrain)
    # record the best result so far
    if accVal > accMax:
        accMax = accVal
        accTrainBest = accTrain
        Kbest = K
# print  best results
print('Maximum validation accuracy =',accMax)
print('Corresponding training accuracy =',accTrainBest)
print('Best number of dimensions =',Kbest)
# plot accuracies
plt.figure()
plt.plot(Klist,accListTrain,'b')
plt.plot(Klist,accListVal,'r')
plt.suptitle('Question 2(d): Training and Validation Accuracy for PCA + QDA')
plt.xlabel('Reduced dimension')
plt.ylabel('Accuracy')



# Question 2(f)
# regularized PCA+QDA.
# find best combination of dimensionality reduction and regularization
print('\nQuestion 2(f):')
Klist = range(1,50)
Rlist = 2.0**np.arange(-20,1)
accMax = 0    # maximum accureacy so far
accList = []
for K in Klist:
    # dimensionality reduction via PCA
    pca = decomp.PCA(K)
    pca.fit(XtrainS)
    XtrainR = pca.transform(XtrainS)    # reduced training data
    XvalR = pca.transform(Xval)    # reduced validation data
    accMaxK = 0   # maximum accuracy for current value of K
    for R in Rlist:
        # regularized classification via QDA
        qda = da.QuadraticDiscriminantAnalysis(reg_param=R)
        ignore_warnings(qda.fit)(XtrainR,TtrainS)
        accVal = qda.score(XvalR,Tval)    # validation accuracy
        # record the best result so far
        if accVal > accMaxK:
            accMaxK = accVal
        if accVal > accMax:
            accMax = accVal
            accTrain = qda.score(XtrainR,TtrainS)
            Kbest = K
            Rbest = R
    accList.append(accMaxK)

# print results
print('Maximum validation accuracy =',accMax)
print('Corresponding training  accuracy =',accTrain)
print('Best number of dimensions =',Kbest)
print('Best regularization parameter =',Rbest)
# plot results
plt.figure()
plt.plot(Klist,accList)
plt.title('Question 2(f): Maximum validation accuracy of QDA')
plt.xlabel('Reduced dimension')
plt.ylabel('Maximum accuracy')




##########  QUESTION 3  ############

print('\n\nQuestion 3.')
print('-------------')


# Question 3(a)
# generate a bootstrap sample with at least 3 points from each class.
def myBootstrap(X,T):
    done = False
    while not(done):
        Xb,Tb = utils.resample(X,T)    # create a bootstrap sample
        counts = np.unique(Tb,return_counts=True)[1]
        done = all(counts > 2)
    return Xb,Tb



# Question 3(b)
# train a bagged classifier based on QDA on the small training set.
# print validation accuracy before and after bagging.

# train and evaluate a base classifier
R = 0.004
clf = da.QuadraticDiscriminantAnalysis(reg_param=R)
ignore_warnings(clf.fit)(XtrainS,TtrainS)
acc = clf.score(Xval,Tval)    # base accuracy
# train and evaluate a bagged classifier
N = len(Tval)
M = np.max(Tval)+1
Psum = np.zeros([N,M])
for j in range(50):
    XtrainB,TtrainB = myBootstrap(XtrainS,TtrainS)    # create a bootstrap sample
    clf = da.QuadraticDiscriminantAnalysis(reg_param=R)
    ignore_warnings(clf.fit)(XtrainB,TtrainB)
    Psum += clf.predict_proba(Xval)
prediction = np.argmax(Psum,axis=1)
accBag = np.mean(prediction==Tval)    # bagged accuracy
print('\nQuestion 3(b):')
print('Base validation accuracy =',acc)
print('Bagged validation accuracy =',accBag)



# Question 3(c)
# train a bagged classifier based on QDA on the small training set.
# plot validation accuracy vs number of bootstrap samples.
N = len(Tval)
M = np.max(Tval)+1
```

```
Psum = np.zeros([N,M])
accList = []
for j in range(500):
    XtrainB,TtrainB = myBootstrap(XtrainS,TtrainS)    # create a bootstrap sample
    clf = da.QuadraticDiscriminantAnalysis(reg_param=R)
    ignore_warnings(clf.fit)(XtrainB,TtrainB)
    Psum += clf.predict_proba(Xval)
    prediction = np.argmax(Psum,axis=1)
    accuracy = np.mean(prediction==Tval)
    accList.append(accuracy)

plt.figure()
plt.plot(accList)
plt.title('Question 3(c): Validation accuracy')
plt.xlabel('Number of bootstrap samples')
plt.ylabel('Accuracy')

plt.figure()
plt.semilogx(accList)
plt.title('Question 3(c): Validation accuracy (log scale)')
plt.xlabel('Number of bootstrap samples')
plt.ylabel('Accuracy')



# Question 3(d)

# create a classifier based on PCA+QDA
# and fit it to data X,T.
# K = reduced dimension for PCA
# R = regularization parameter for QDA
def train3d(K,R,X,T):
    pca = decomp.PCA(K)
    Xr = pca.fit_transform(X)    # reduced data
    qda = da.QuadraticDiscriminantAnalysis(reg_param=R)
    ignore_warnings(qda.fit)(Xr,T)
    return pca,qda


# given a PCA+QDA classifier, predict probabilities for data matrix X
def proba3d(pca,qda,X):
    Xr = pca.transform(X)    # reduced data
    return qda.predict_proba(Xr)



# Question 3(e)

# train a bagged classifier based on PCA+QDA on the small data set.
# return its validation accuracy before and after bagging.
def myBag(K,R):
    # train and evaluate a base classifier
    pca,qda = train3d(K,R,XtrainS,TtrainS)
    XvalR = pca.transform(Xval)
    acc = qda.score(XvalR,Tval)    # base accuracy
    # train and evaluate a bagged classifier
    N = len(Tval)    # number of data points
    M = np.max(Tval)+1    # number of classes
    Psum = np.zeros([N,M])    # initialize summed probabilities to 0
    J = 200    # number of bootstrap samples
    for j in range(J):
        XtrainB,TtrainB = myBootstrap(XtrainS,TtrainS)    # create a bootstrap sample
        pca,qda = train3d(K,R,XtrainB,TtrainB)
        P = proba3d(pca,qda,Xval)    # predicted probabilities
        Psum += P
    prediction = np.argmax(Psum,axis=1)    # bagged predictions
    accBag = np.mean(prediction==Tval)    # bagged accuracy
    return acc,accBag



# Question 3(f)
# train a bagged classifier based on PCA+QDA on the small data set.
# print validation accuracy before and after bagging.
acc,accBag = myBag(100,0.01)
print("\nQuestion 3(f):")
print("Base validation accuracy =",acc)
print("Bagged validation accuracy =",accBag)



# Question 3(g).
# Create and test N bagged classifiers based on PCA+QDA.
# train them on the small data set.
# Use random combinations of reduced dimension and regularization parameter.
# Return lists of validation accuracy before and after bagging.
# RandR is a function tnat generates a random value for the regularization parameter, R.
# RandK is a function that generates a random value for the reduced dimension, K.
def testBag(randR,randK,N):
    accList = []
    accBagList = []
    for i in range(N):
        R = randR()
        K = randK()
        acc,accBag = myBag(K,R)
        accList.append(acc)
        accBagList.append(accBag)
    plt.figure()
    plt.scatter(accList,accBagList,c='b')
    plt.ylim(0,1)
    plt.xlabel('Base validation accuracy')
    plt.ylabel('Bagged validation accuracy')
    return accList,accBagList

# train 50 bagged classifiers based on PCA+QDA on the small data set
# where the base classifiers overfit the data.
# plot their validation accuracies.
randR = lambda: rnd.rand(1)*0.8 + 0.2    # generate random reals betwen 0.2 and 1
randK = lambda: rnd.randint(1,11)    # generate random integers between 1 and 10, inclusive
testBag(randR,randK,50)
plt.xlim(0,1)
plt.title('Question 3(g): Bagged v.s. base validation accuracy')
# add a red diagonal line to the plot
plt.plot((0,1),(0,1),'r')



# Question 3(h).
# train 50 bagged classifiers based on PCA+QDA on the small data set
# where the base classifiers underfit the data.
# plot their validation accuracies.
randR = lambda: rnd.rand(1)*0.05    # generate random reals between 0 and 0.05
randK = lambda: rnd.randint(50,201)    # generate random integers between 50 and 200, inclusive
accList,accBagList = testBag(randR,randK,50)
plt.title('Question 3(h): Bagged v.s. base validation accuracy')

# add a red horizontal line to the plot
accBagMax = np.max(accBagList)
accMin = np.min(accList)
accMax = np.max(accList)
plt.plot((accMin,accMax),(accBagMax,accBagMax),'r')

print("\nQuestion 3(g):")
print('Maximum bagged validation accuracy =',accBagMax)





##########  QUESTION 4  ############

print('\n\nQuestion 4.')
print('---------------')


# read the training and testing data
with open('dataA2Q2.pickle','rb') as file:
    dataTrain,dataTest = pickle.load(file)
Xtrain,Ttrain = dataTrain
Xtest,Ttest = dataTest



# Question 4(a)
# plot the data points in X and the clusters defined by R and Mu
def plot_clusters(X,R,Mu):
    # sort columns by total responsibility
    Rsum = np.sum(R,axis=0)
    R = R[:,np.argsort(Rsum)]
    plt.scatter(X[:, 0], X[:, 1], color=R,s=5)
    plt.scatter(Mu[:,0],Mu[:,1],color='k')
```

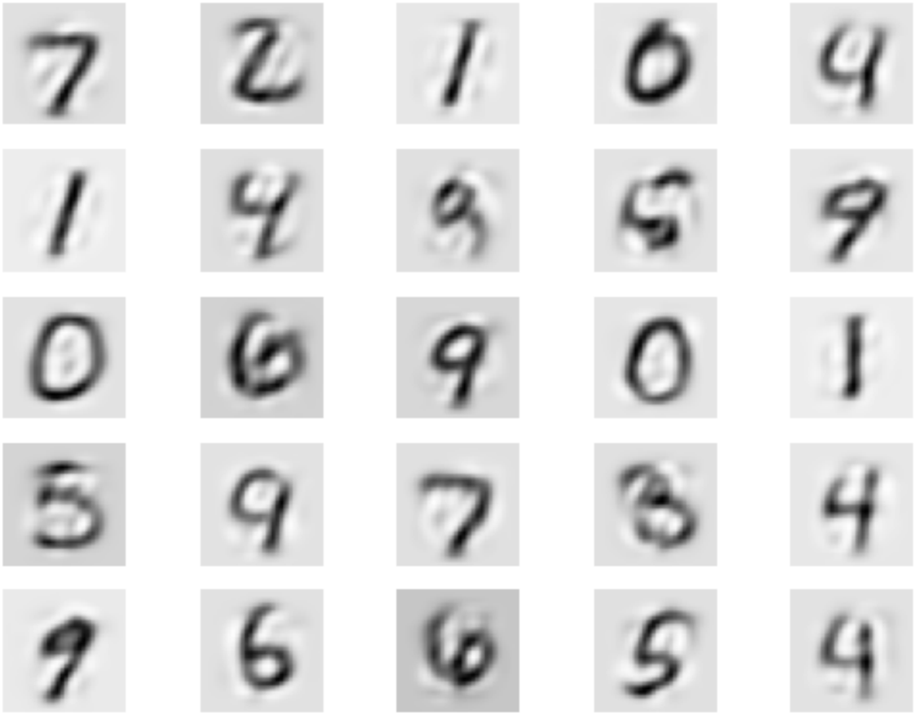
```python
# Question 4(b)

# convert T from an integer to a one-hot encoding
def onehot(T):
    T = np.reshape(T,[-1,1])
    K = np.max(T)+1    # number of classes
    idx = np.reshape(range(K),[1,K])
    return (T==idx).astype(np.int32)

# Use K means to find 3 clusters in the training data
km = cluster.KMeans(n_clusters=3)
km.fit(Xtrain)
# plot the clusters
predictions = km.predict(Xtrain)
R = onehot(predictions)
Mu = km.cluster_centers_
plt.figure()
plot_clusters(Xtrain,R,Mu)
plt.title('Question 4(b): K means')
# print the training and test scores
scoreTrainB = km.score(Xtrain)
scoreTestB = km.score(Xtest)
print('\nQuestion 4(b):')
print('Training score =',scoreTrainB)
print('Test score =',scoreTestB)




# Question 4(c)
# Train a GMM classifier with spherical covariance matrices,
# and print the value of the objective function.
gmm = mixture.GaussianMixture(n_components=3,covariance_type='spherical')
gmm.fit(Xtrain)
# plot the clusters
R = gmm.predict_proba(Xtrain)
Mu = gmm.means_
plt.figure()
plot_clusters(Xtrain,R,Mu)
plt.title('Question 4(c): Gaussian mixture model (spherical)')
# print the mean log-likelihood of the training and test data given the GMM
scoreTrainC = gmm.score(Xtrain)
scoreTestC = gmm.score(Xtest)
print('\nQuestion 4(c):')
print('Training score =',scoreTrainC)
print('Test score =',scoreTestC)



# Question 4(d)
# Train a GMM classifier with 3 clusters and unrestricted covariance matrices
gmm = mixture.GaussianMixture(n_components=3,covariance_type='full')
gmm.fit(Xtrain)
# plot the clusters
R = gmm.predict_proba(Xtrain)
Mu = gmm.means_
plt.figure()
plot_clusters(Xtrain,R,Mu)
plt.title('Question 4(d): Gaussian mixture model (full)')
# print the mean log-likelihood of the training and test data given the GMM
scoreTrainD = gmm.score(Xtrain)
scoreTestD = gmm.score(Xtest)
print('\nQuestion 4(d):')
print('Training score =',scoreTrainD)
print('Test score =',scoreTestD)
print('Q4d-Q4c test scores =',scoreTestD-scoreTestC)



# Question 4(e)

# randomly choose K distinct points from the training data.
def initMu1(X,K):
    Mu = utils.resample(X,n_samples=K,replace=False)
    return Mu

# Randomly generate K 2-dimensional points from a uniform distribution
# over the smallest rectangle containing the training data.
def initMu2(X,K):
    N,M = np.shape(X)
    Xmin = np.min(X,axis=0)
    Xmax = np.max(X,axis=0)
    Mu = Xmin + (Xmax-Xmin)*rnd.rand(K,M)
    return Mu


# shape(Mu) = [K,M]
# Run K-means for I iterations on data X
def myKmeans(X,K,I):
    N,M = np.shape(X)
    # initialize the means
    Mu = initMu2(X,K)
    # alternating minimization
    Mu = np.reshape(Mu,[1,K,M])
    X = np.reshape(X,[N,1,M])
    scoreList = []    # initialize list of scores
    for i in range(I):
        # compute scores (objective function)
        Dsq = np.sum((X-Mu)**2,axis=2,keepdims=True)   # squared distances from points to cluster centers.
shape = [N,K,1]
        Dmin = np.min(Dsq,axis=1,keepdims=True)    # minimum squared distances.  shape = [N,1,1]
        score = np.sum(Dmin)
        scoreList.append(score)
        # update assignnments (assignment step)
        R = (Dsq==Dmin)    # one-hot encoding of Assignments.  shape = [N,K,1]
        R = R.astype(np.int32)
        # update cluster centers (refitting step)
        Rsum = np.sum(R,axis=0,keepdims=True)    # number of points in each cluster.  shape = [1,K,1]
        Xsum = np.sum(X*R,axis=0,keepdims=True)    # sum of points in each cluster.  shape = [1,K,M]
        Mu = Xsum/Rsum    # new cluster centers.  shape=[1,K,M]
    Mu = np.reshape(Mu,[K,M])
    R = np.reshape(R,[N,K])
    return Mu,R,scoreList

# return the K-means score of data set X on the clusters defined by Mu
def scoreKmeans(X,Mu):
    N,M = np.shape(X)
    K,M = np.shape(Mu)
    X = np.reshape(X,[N,1,M])
    Mu = np.reshape(Mu,[1,K,M])
    Dsq = np.sum((X-Mu)**2,axis=2)   # squared distances from points to cluster centers.  shape = [N,K]
    Dmin = np.min(Dsq,axis=1)    # minimum squared distances.  shape = [N]
    return np.sum(Dmin)

# Use 100 iterations of K means to find 3 clusters in the training data
Mu,R,scoreList = myKmeans(Xtrain,3,100)
# plot the progress of K means
plt.figure()
plt.plot(scoreList[:20])
plt.title('Question 4(e): score vs iteration (K means)')
plt.xlabel('Iteration')
plt.ylabel('Score')
# plot the clusters
plt.figure()
plot_clusters(Xtrain,R,Mu)
plt.title('Question 4(e): Data clustered by K means')
# print the K-means score on the training and test data
scoreTrainE = scoreKmeans(Xtrain,Mu)
scoreTestE = scoreKmeans(Xtest,Mu)
print('\nQuestion 4(e):')
print('Training score =',scoreTrainE)
print('Test score =',scoreTestE)
print('Q4e+Q4b =',scoreTestE+scoreTestB)




# Question 4(f)

# shape(Mu) = [K,M]
# Fit a mixture of K Gaussians to data X using I iterations of EM
def myGMM(X,K,I):
    N,M = np.shape(X)
    # initialize the means and mixing coefficients
    Mu = initMu2(X,K)
    Pi = np.ones([K])/K    # each mixing coefficient gets the same value, 1/K
    # # alternate method of initializing Pi,
    # # where each mixing coefficient gets a uniform random value.
    # Pi = rnd.rand(K)
    # Pi = Pi/np.sum(Pi)

    # alternating minimization
    X = np.reshape(X,[N,1,M])
    Mu = np.reshape(Mu,[1,K,M])
    Pi = np.reshape(Pi,[1,K,1])
    scoreList = []    # initialize list of scores
    C = (2*np.pi)**(M/2.0)    # normalizing constant for Gaussian probabilities
    for i in range(I):
        # compute scores (mean log-likelihood)
        Dsq = np.sum((X-Mu)**2,axis=2,keepdims=True)   # squared distances from points to cluster centers.
shape = [N,K,1]
        P = Pi*np.exp(-Dsq/2)/C    # joint probabilities.  shape = [N,K,1]
        Psum = np.sum(P,axis=1,keepdims=True)    # Likelihood.  shape = [N,1,1]
        score = np.mean(np.log(Psum))    # mean log-likelihood
        scoreList.append(score)
        # update responsibilities (E step)
        R = P/Psum    # responsibilities.  shape = [N,K,1]
        # update Mu and Pi (M step)
        Rsum = np.sum(R,axis=0,keepdims=True)    # number of points in each cluster.  shape = (1,K,1)
        Xsum = np.sum(X*R,axis=0,keepdims=True)    # sum of points in each cluster.  shape = [1,K,M]
        Mu = Xsum/Rsum    # new cluster centers.  shape=[1,K,M]
        Pi = Rsum/N    # shape = [1,K,1]
    Mu = np.reshape(Mu,[K,M])
    Pi = np.reshape(Pi,[K])
    R = np.reshape(R,[N,K])
    return Mu,Pi,R,scoreList


# compute the mean log-likelihood of data set X on the clusters defined by Mu and Pi
def scoreGMM(X,Mu,Pi):
    N,M = np.shape(X)
    K,M = np.shape(Mu)
    X = np.reshape(X,[N,1,M])
    Mu = np.reshape(Mu,[1,K,M])
    Pi = np.reshape(Pi,[1,K])

    Dsq = np.sum((X-Mu)**2,axis=2)   # squared distances from points to cluster centers.  shape = [N,K]
    P = Pi*np.exp(-Dsq/2)/(2*np.pi)    # joint probabilities.  shape = [N,K]
    L = np.sum(P,axis=1)    # likelihoods.  shape = [N]
    score = np.mean(np.log(L))    # mean log-likelihood
    return score


# Use 100 iterations of EM to fit a GMM with 3 clusters to the training data
Mu,Pi,R,scoreList = myGMM(Xtrain,3,100)
# plot the progress of EM
plt.figure()
plt.plot(scoreList[:20])
plt.title('Question 4(f): score vs iteration (EM for GMM)')
plt.xlabel('Iteration')
plt.ylabel('Score')
# plot the clusters
plt.figure()
plot_clusters(Xtrain,R,Mu)
plt.title('Question 4(f): Data clustered by EM for GMM')
# print the mean log-likelihood of the GMM on the training and test data
scoreTrainF = scoreGMM(Xtrain,Mu,Pi)
scoreTestF = scoreGMM(Xtest,Mu,Pi)
print('\nQuestion 4(f):')
print('Training score =',scoreTrainF)
print('Test score =',scoreTestF)
print('Q4c-Q4f test scores =',scoreTestC-scoreTestF)
```

PRINTED OUTPUT

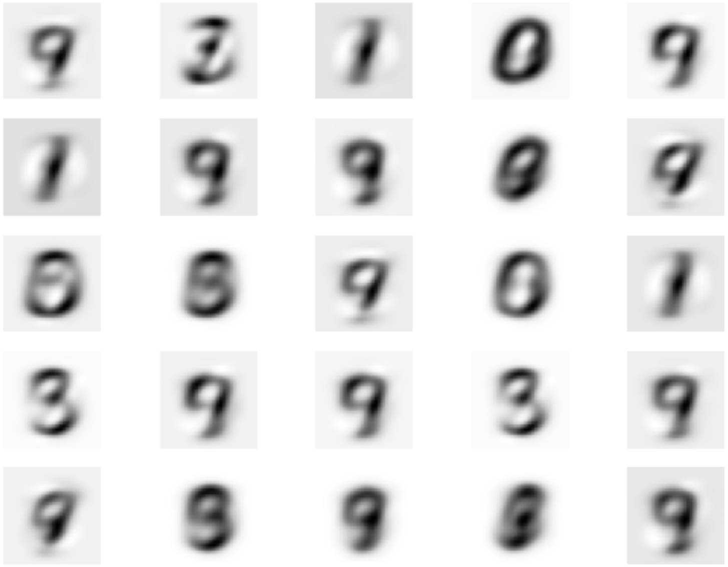
```
Question 1.
—————

Question 1(a):
RMS error =  3.7242454544793344

Question 1(b):
RMS error =  6.361345615754785

Question 1(c):
RMS error =  0.8532700041480263

Question 1(f):
RMS error = 1.1139609490888703e-13


Question 2.
--------------

Question 2(a):
Training accuracy = 1.0
Test accuracy = 0.1415

Question 2(b):
Maximum validation accuracy = 0.2452
Corresponding training accuracy = 0.265
Best regularization parameter = 0.00390625

Question 2(d):
Maximum validation accuracy = 0.6938
Corresponding training accuracy = 0.93
Best number of dimensions = 8

Question 2(f):
Maximum validation accuracy = 0.8529
Corresponding training  accuracy = 0.985
Best number of dimensions = 18
Best regularization parameter = 0.25


Question 3.
```

```
--------------

Question 3(b):
Base validation accuracy = 0.2444
Bagged validation accuracy = 0.6013

Question 3(f):
Base validation accuracy = 0.4099
Bagged validation accuracy = 0.8447

Question 3(g):
Maximum bagged validation accuracy = 0.8575


Question 4.
--------------

Question 4(b):
Training score = -3536.4532603972275
Test score = -3733.380113243647

Question 4(c):
Training score = -3.6947424147740775
Test score = -3.7610707678460384

Question 4(d):
Training score = -3.594658073857925
Test score = -3.729211552517624
Q4d-Q4c test scores = 0.031859215328414514

Question 4(e):
Training score = 3536.424192018344
Test score = 3732.2514702478575
Q4e+Q4b = -1.1286429957895052

Question 4(f):
Training score = -3.7466090254900184
Test score = -3.8131878042469407
Q4c-Q4f test scores = 0.05211703640090226
```

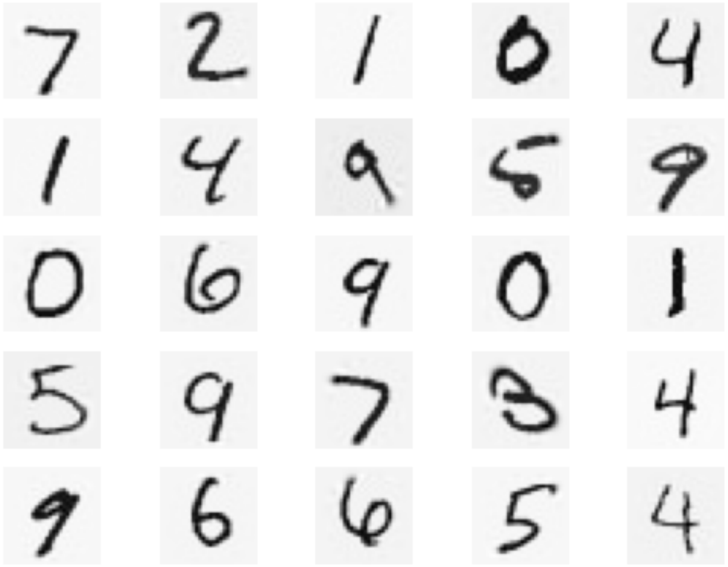
Question 1(a): MNIST test data projected onto 30 dimensions



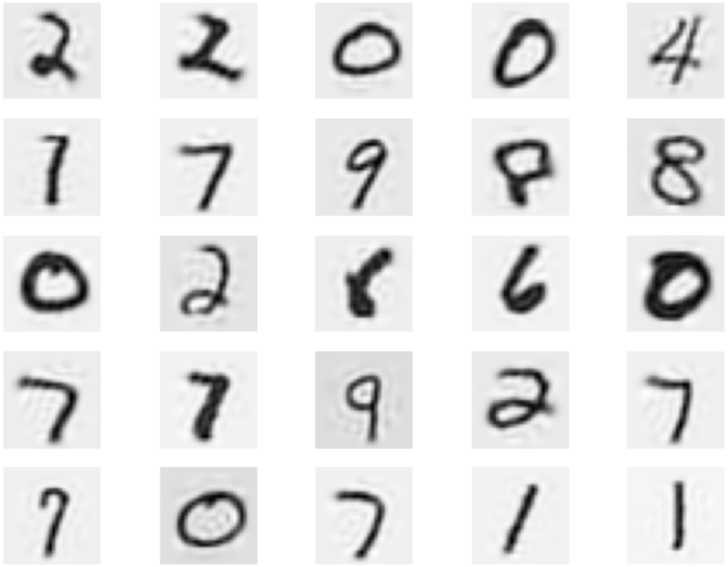
Question 1(b): MNIST test data projected onto 3 dimensions



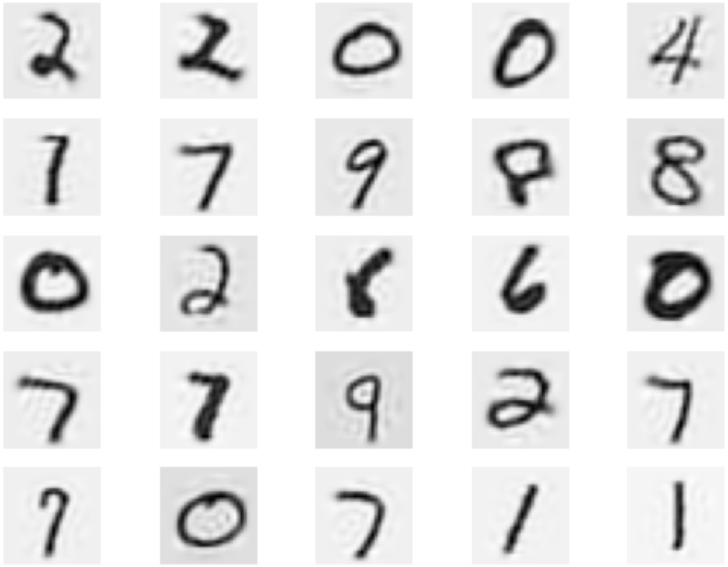
Question 1(c): MNIST test data projected onto 300 dimensions

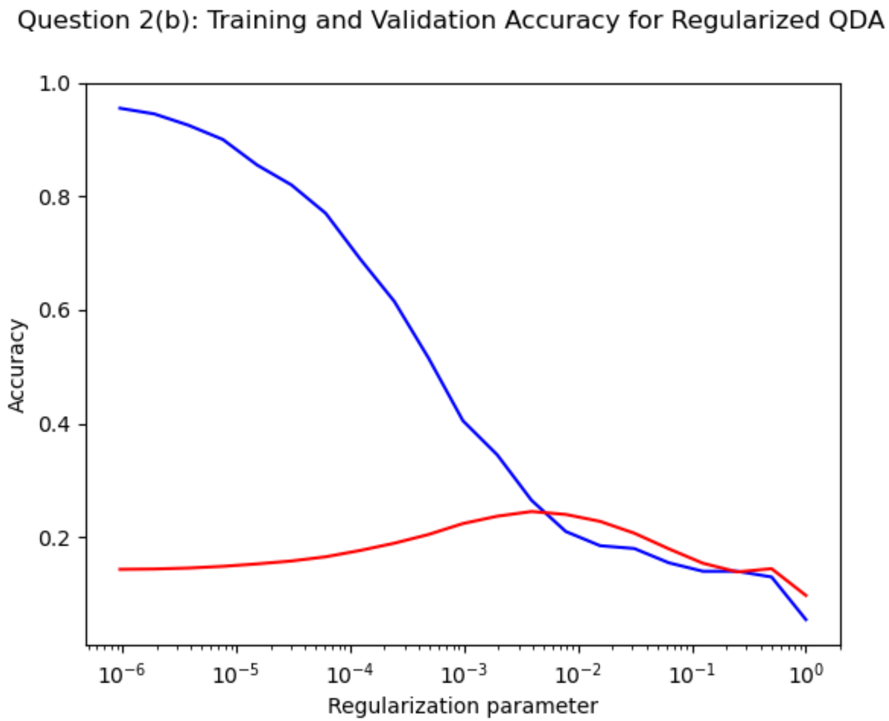


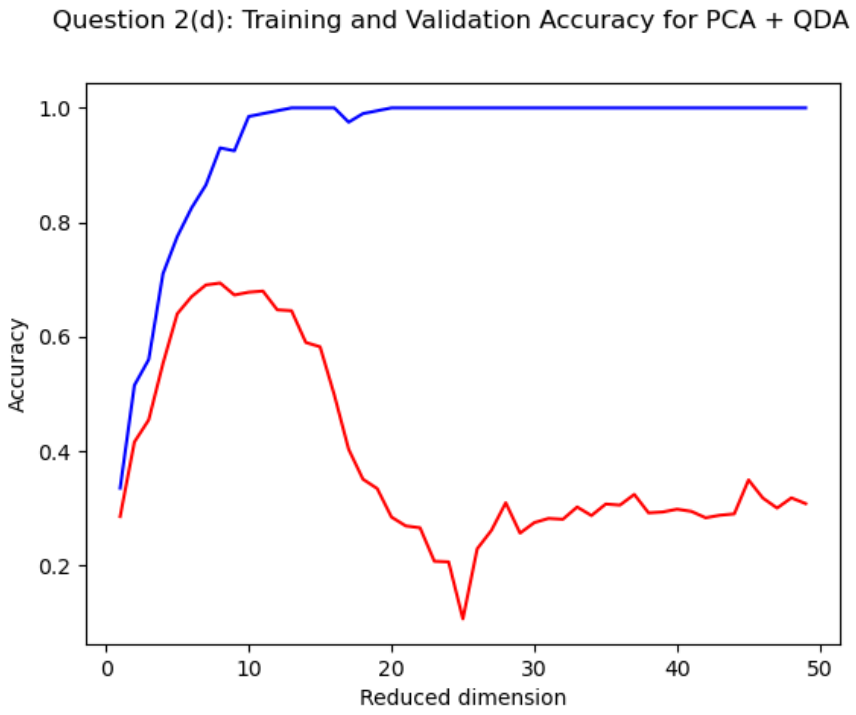
Question 1(f): MNIST data projected onto 100 dimensions (mine)

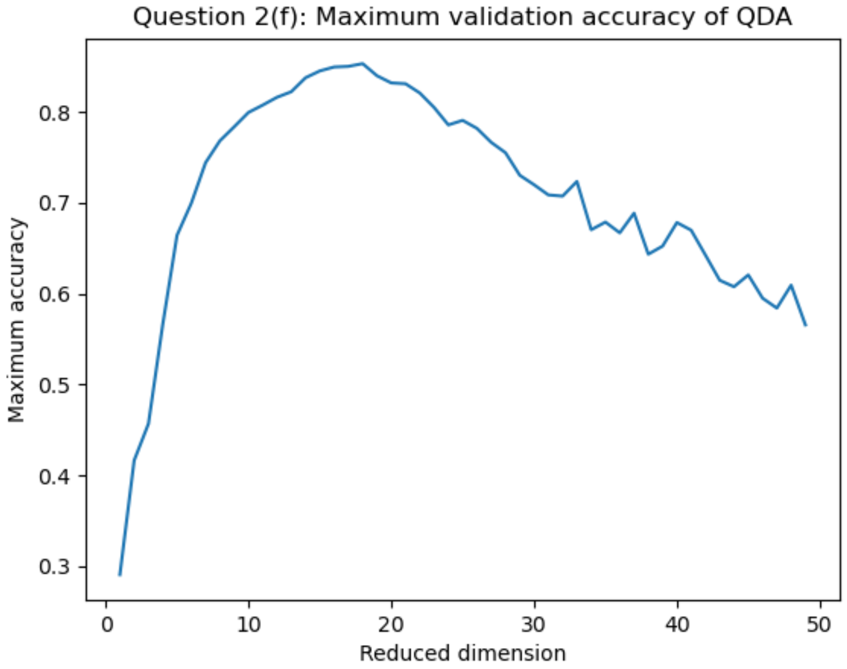


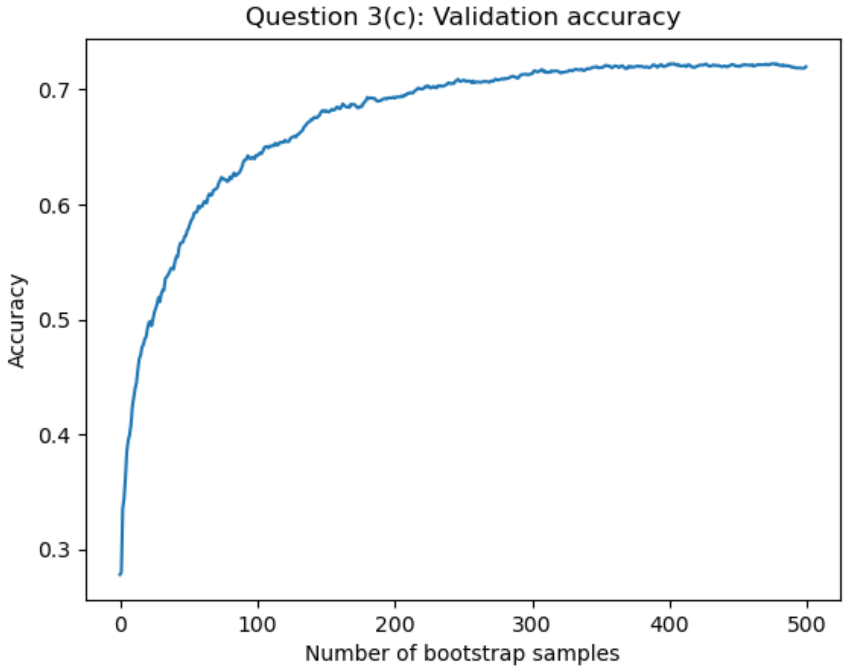
Question 1(f): MNIST data projected onto 100 dimensions (sklearn)

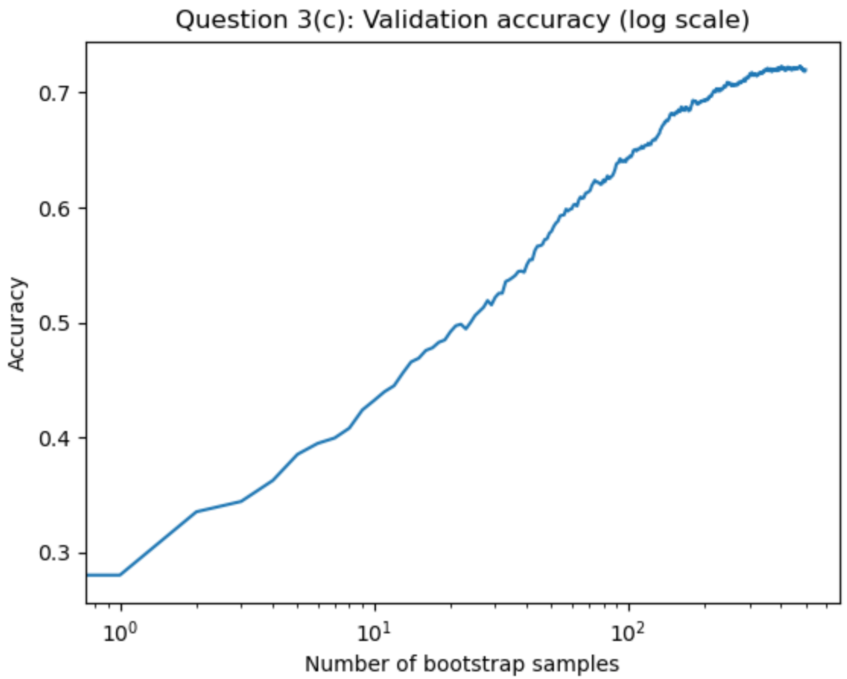


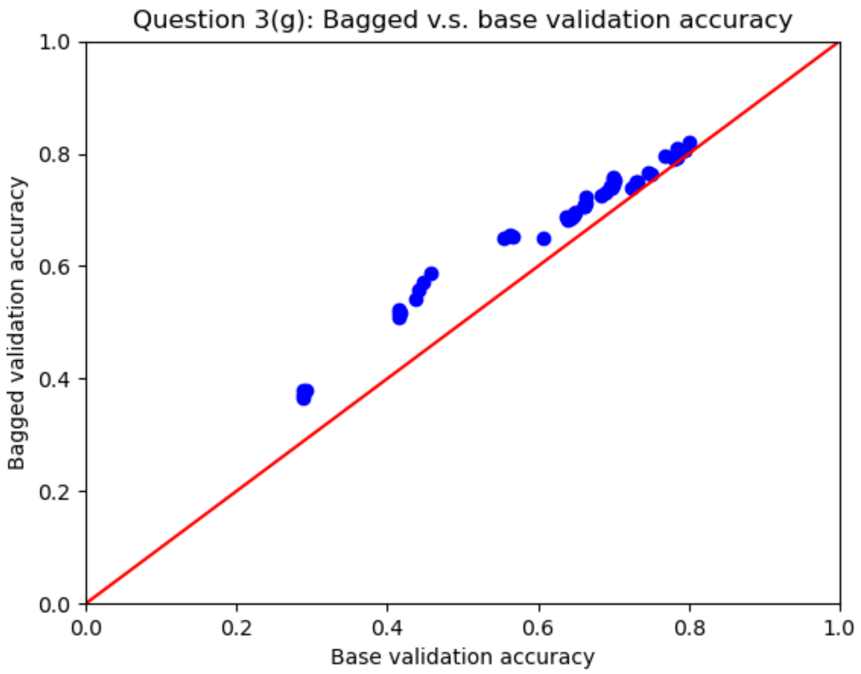

Question 2(b): Training and Validation Accuracy for Regularized QDA
1.0
0.8
0.6
0.4
0.2
Accuracy
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
Regularization parameter

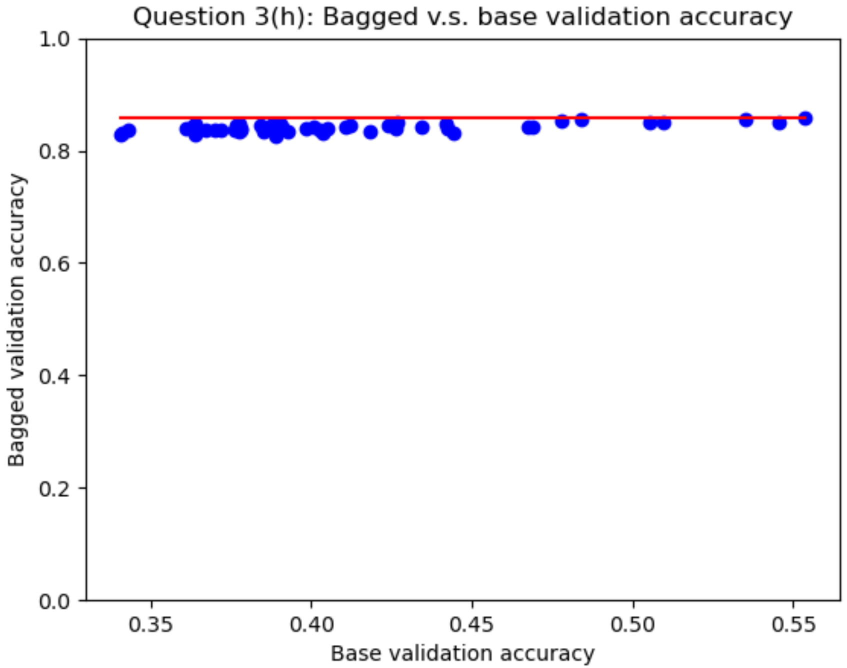

Question 2(d): Training and Validation Accuracy for PCA + QDA
Accuracy
1.0
0.8
0.6
0.4
0.2
0
10
20
30
40
50
Reduced dimension

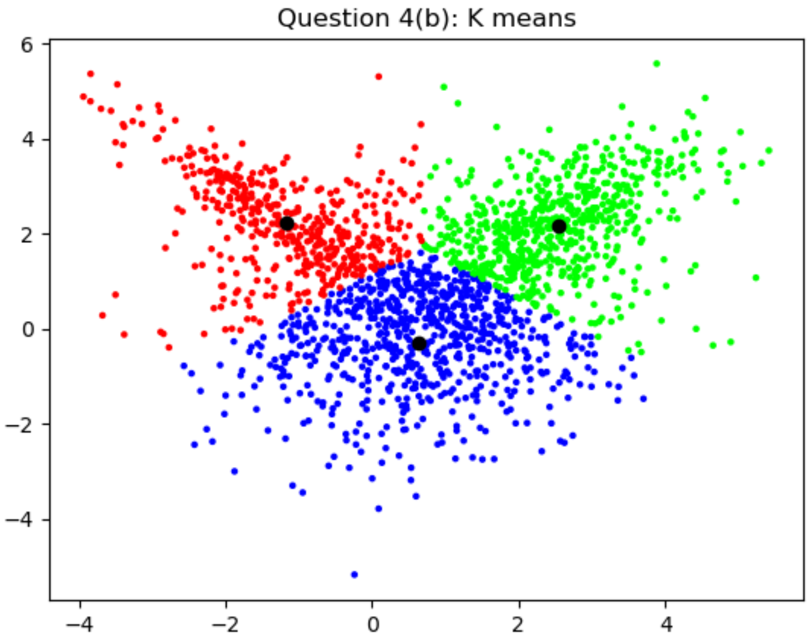

Question 2(f): Maximum validation accuracy of QDA
Maximum accuracy
0.3
0.4
0.5
0.6
0.7
0.8
0
10
20
30
40
50
Reduced dimension

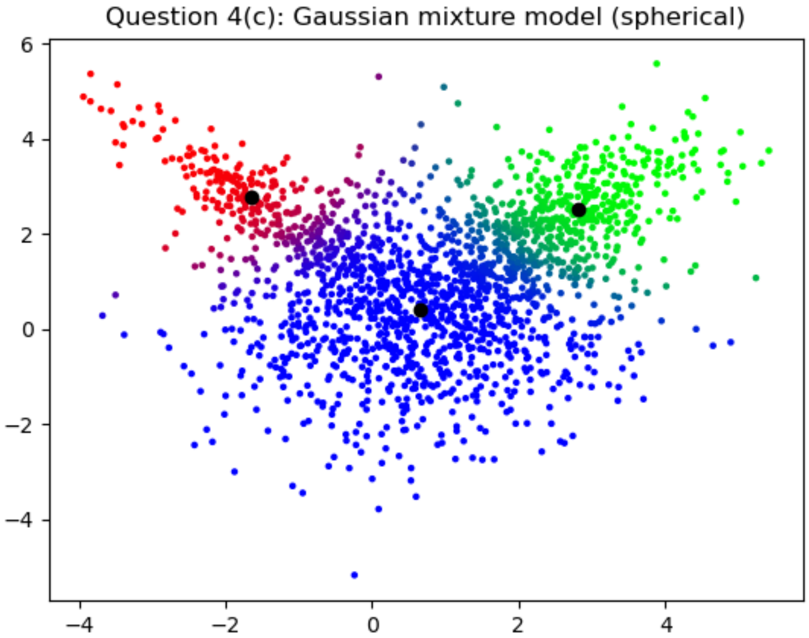

Question 3(c): Validation accuracy
Accuracy
0.3
0.4
0.5
0.6
0.7
0
100
200
300
400
500
Number of bootstrap samples

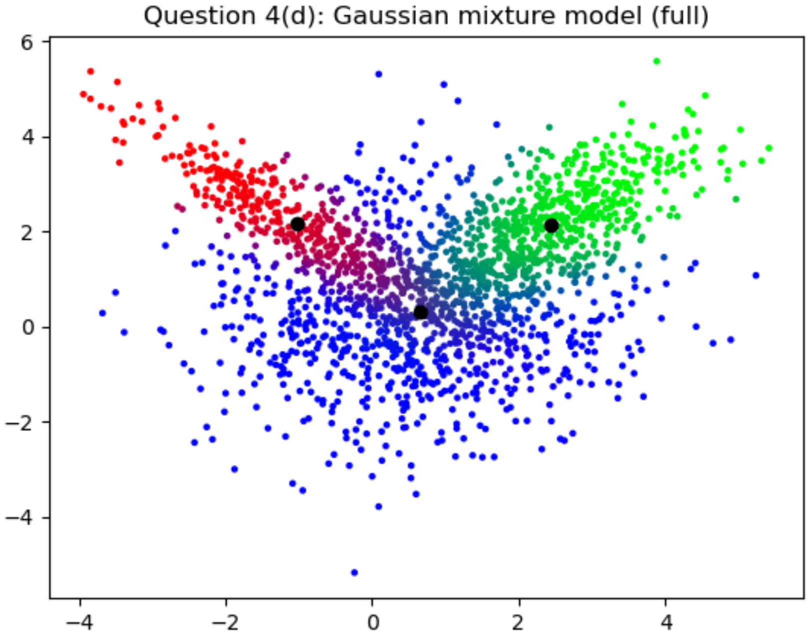

Question 3(c): Validation accuracy (log scale)
Accuracy
0.3
0.4
0.5
0.6
0.7
10^0
10^1
10^2
Number of bootstrap samples

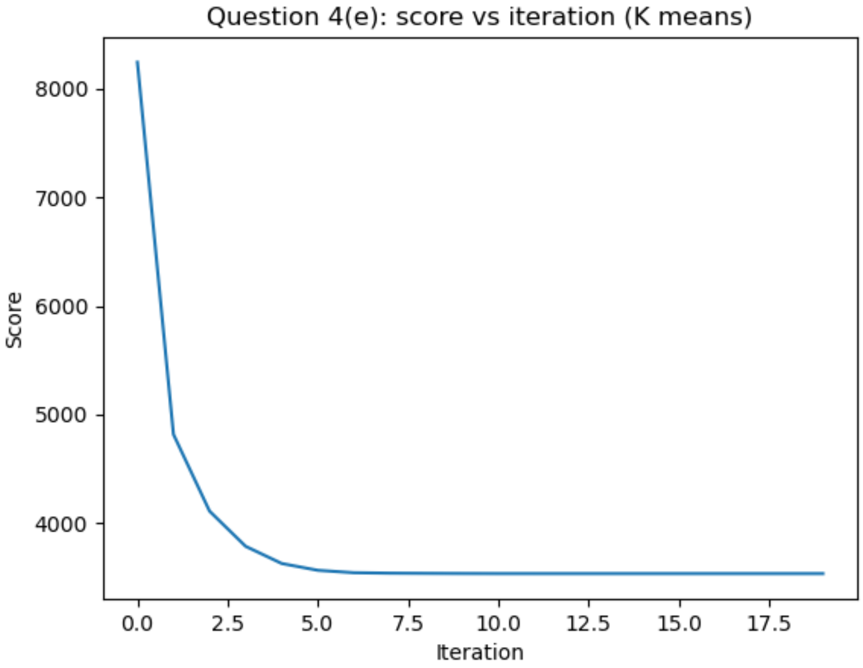

Question 3(g): Bagged v.s. base validation accuracy
Bagged validation accuracy
Base validation accuracy
0.0
0.2
0.4
0.6
0.8
1.0

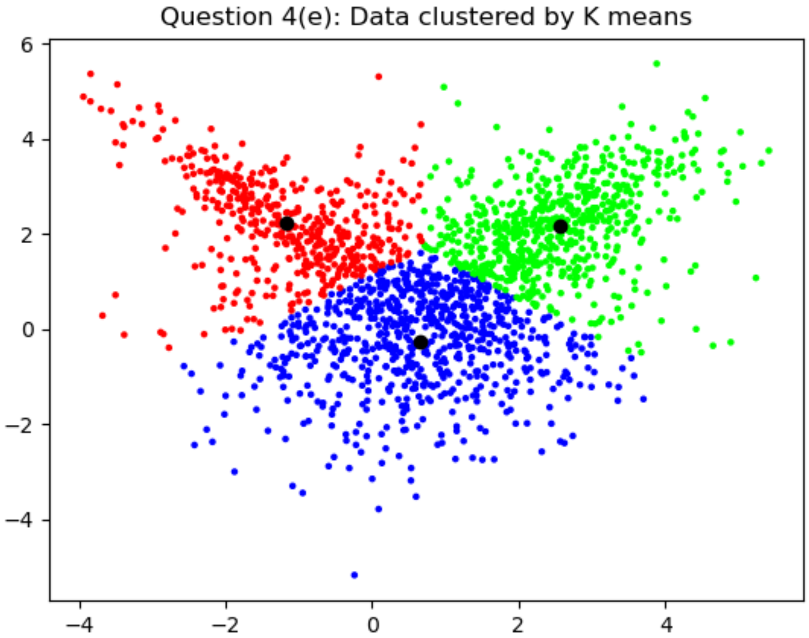

Question 3(h): Bagged v.s. base validation accuracy
Bagged validation accuracy
Base validation accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0.35
0.40
0.45
0.50
0.55

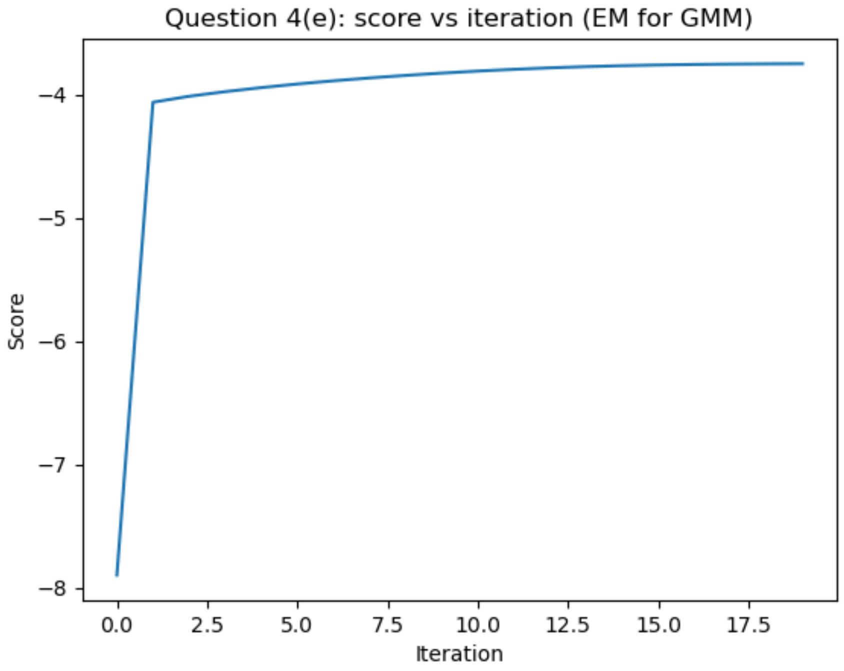

Question 4(b): K means
6
4
2
0
−2
−4
−4
−2
0
2
4

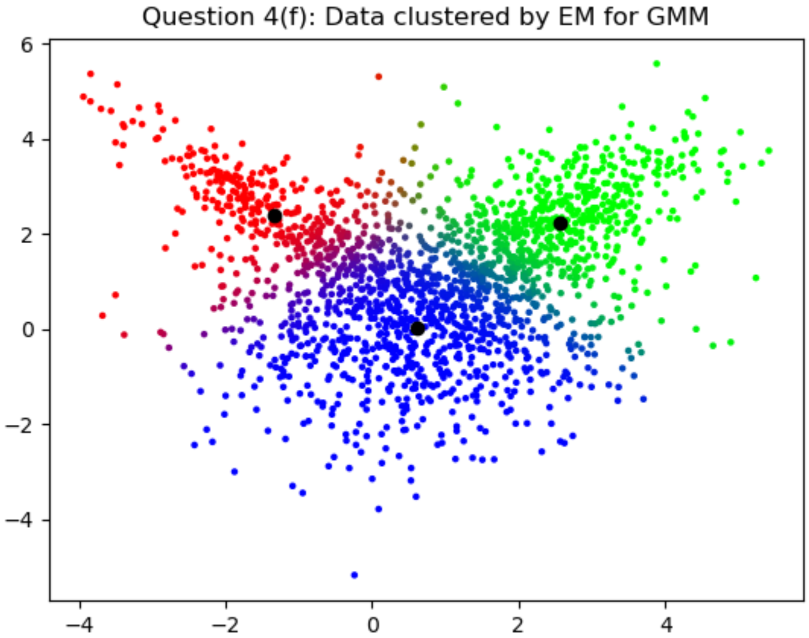

Question 4(c): Gaussian mixture model (spherical)
6
4
2
0
−2
−4
−4
−2
0
2
4


Question 4(d): Gaussian mixture model (full)
6
4
2
0
−2
−4
−4
−2
0
2
4


Question 4(e): score vs iteration (K means)
Score
8000
7000
6000
5000
4000
Iteration
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Question 4(e): Data clustered by K means
6
4
2
0
−2
−4
−4
−2
0
2
4


Question 4(e): score vs iteration (EM for GMM)
Score
Iteration
−4
−5
−6
−7
−8
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Question 4(f): Data clustered by EM for GMM
6
4
2
0
−2
−4
−4
−2
0
2
4