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

Assignment 2: Questions and Solutions

Due date: Sunday November 15, 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.

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 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.

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 cardinal 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. Whenever possible, *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.

You should also avoid the use of recursion or higher-order functions in numerical computations. This includes the python `map` function or any numpy function listed under "functional programming", such as `apply-along-axis`, unless otherwise specified. You should also avoid using Python functions that operate on lists, such as `zip`. These are often loops in disguise and are 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. 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
```

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. *Linear Regression with Feature Mapping.* (30 points total)

   In this question, you will write a Python program to fit a non-linear function to data with linear least-squares regression. As in Question 3 of Assignment 1, the data consists of a set of pairs, (x, t), where the input, x, and the target value, t, are both real numbers. However, in this question, you will use trigonometric functions to define a non-linear feature mapping, ψ, that transforms x into a feature vector, z, before doing linear regression. In particular,

   $$z = \psi(x) = [1, \sin(x), \sin(2x), ... \sin(kx), \cos(x), \cos(2x), ..., \cos(kx)] \quad (1)$$

   for some positive integer, k. Note that z is a vector with $2k + 1$ elements. Likewise, the function ψ is said to consist of $2k + 1$ *basis functions*. For more information on linear regression and basis functions, see Chapter 3.1 in Bishop.

   The function you will fit to data has the form

   $$y(x) = w^T z = w^T \psi(x) \quad (2)$$

   where w is a vector of weights. Thus, y is a linear combination of basis functions.

   Your program should find the weight vector, w, for which the function $y(x)$ best fits the data. In particular, it should find w to minimize the loss function,

   $$l(w) = \sum_{n=1}^{N} [t^{(n)} - y(x^{(n)})]^2 = \sum_{n=1}^{N} [t^{(n)} - w^T z^{(n)}]^2 \quad (3)$$

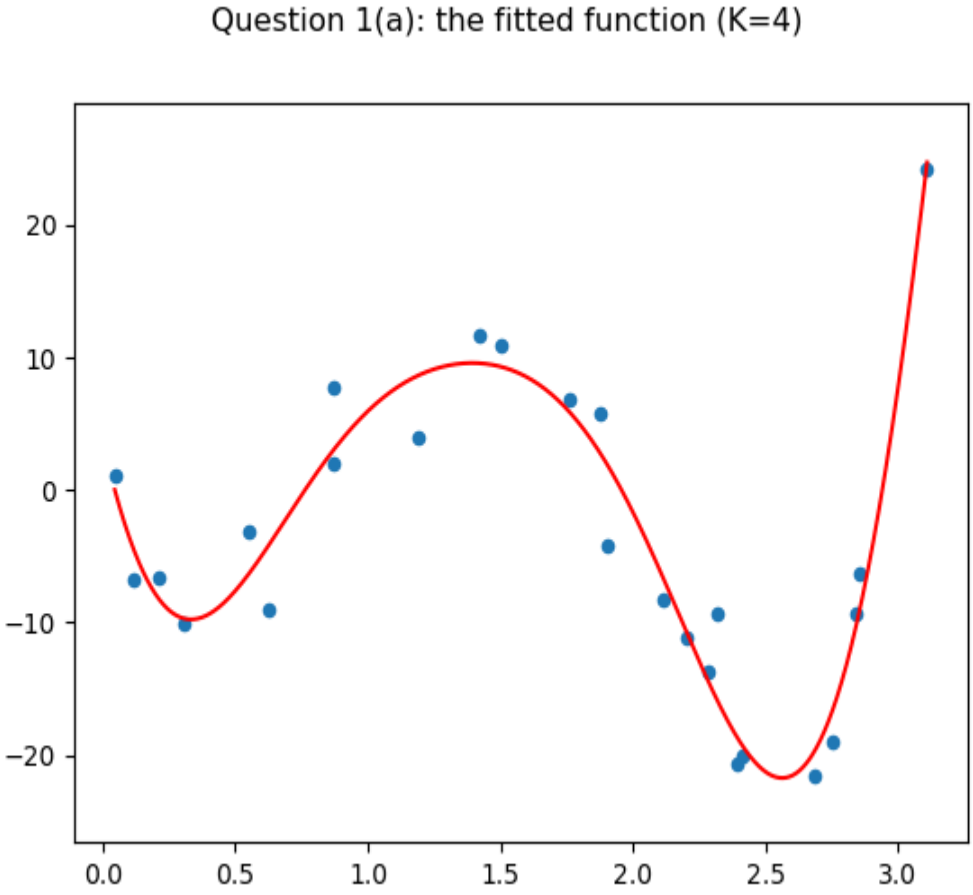
   where the sum is over all training points, $(x^{(n)}, t^{(n)})$. Your program should also compute the mean squared training and test errors, given by:

   $$err_{train} = \sum_{n=1}^{N_{train}} [t^{(n)} - y(x^{(n)})]^2 / N_{train}$$
   $$err_{test} = \sum_{n=1}^{N_{test}} [t^{(n)} - y(x^{(n)})]^2 / N_{test}$$

   where the two sums are over the training data and test data, respectively, and N_{train} and N_{test} are the number of training and test points, respectively.

   The data you will use is in the file `dataA2Q1.pickle.zip` on the course web site. Download and uncompress this file. The file contains training and test data. Next, start the Python interpreter and import the `pickle` module. You can then read the file with the following Python command:

Figure 1:




```
with open('dataA2Q1.pickle','rb') as file:
    dataTrain,dataTest = pickle.load(file)
```

The variable `dataTrain` will now contain the training data, and `dataTest` will contain the test data. Specifically, `dataTrain` is a 2×25 Numpy array, where each column is an (x, t) pair. (Equivalently, the first row gives all the input values, and the second row gives all the target values.) Likewise, `dataTest` is a 2×1000 array of test data. The training data is illustrated by the blue dots in Figure 1.

In answering the questions below, do not use any Python loops unless specified otherwise and do not use any functions from `sklearn`. The point is to implement everything yourself from scratch. All code should be vectorized. To receive any marks for a question below, it must produce the correct answer, or something approximately correct. To receive full marks, the output must be exactly correct and you must meet all constraints on the number of loops, lines of code, vectorization and readibility. To receive any marks for part (a), it must produce correct answers for the other parts.

(a) (6 points) Write a Python function `fit_plot(dataTrain,dataTest,K)` that uses

linear least squares regression to fit a function to the data in `dataTrain`. You should use the feature mapping in Equation (1) with $k =$ `K` to construct a feature vector for each data point. You should use the method `lstsq(Z,T)` in `numpy.linag` to solve the linear least-squares problem itself.[1] Here `Z` is the feature matrix of the training data, and `T` is a vector of the target values. That is, row `n` of `Z` is the feature vector for training point `n`, and `T[n]` is its target value. The function `lstsq` returns a number of values. The first value is the weight vector, w. Your function should use w to compute the training and test error. (Do not use any of the other values returned by `lstsq` in this question.) The function should return the weight vector, the training error and the test error.

In addition, your function should plot the fitted curve on top of the training data. Use the function `scatter` to plot the training data as blue dots, and specify a dot size of 20. Use the function `plot` to draw a fitted curve in red using 1,000 values of x equally spaced between the smallest x-value in the training set and the largest. Use the function `ylim` in `pyplot` to put upper and lower limits on the vertical axis. For the upper limit, use the maximum target value in the training data + 5. For the lower limit, use the minimum target value in the training data - 5. If you have done everything correctly you should get exactly the result shown in Figure 1 when the function is called on the training data in the file `dataA2Q1.pickle` with `K=4`.

You should implement the function so that it can help you later in this question. The function can easily be written in at most 39 lines of modular and highly-readible code (not counting comments and blank lines). Do not use any loops. Hint 1: to avoid loops, you may find the following general fact useful: if u and v are column vectors, then uv^T is a matrix and $[uv^T]_{ij} = u_i v_j$. Hint 2: as an exercise, first try to create the vector $[\sin x, \sin 2x, \sin 3x, \ldots \sin kx]$ for a single value of x without using loops.

(b) (3 points) Run your function `fit_plot` on the training and test data in the file `dataA2Q1.pickle` with `K=3`. Print the value of `K`, the training error, the test error and the weight vector, in that order. Title the figure, *Question 1(b): the fitted function (K=3)*. Label the horizontal axis x, and the vertical axis y.

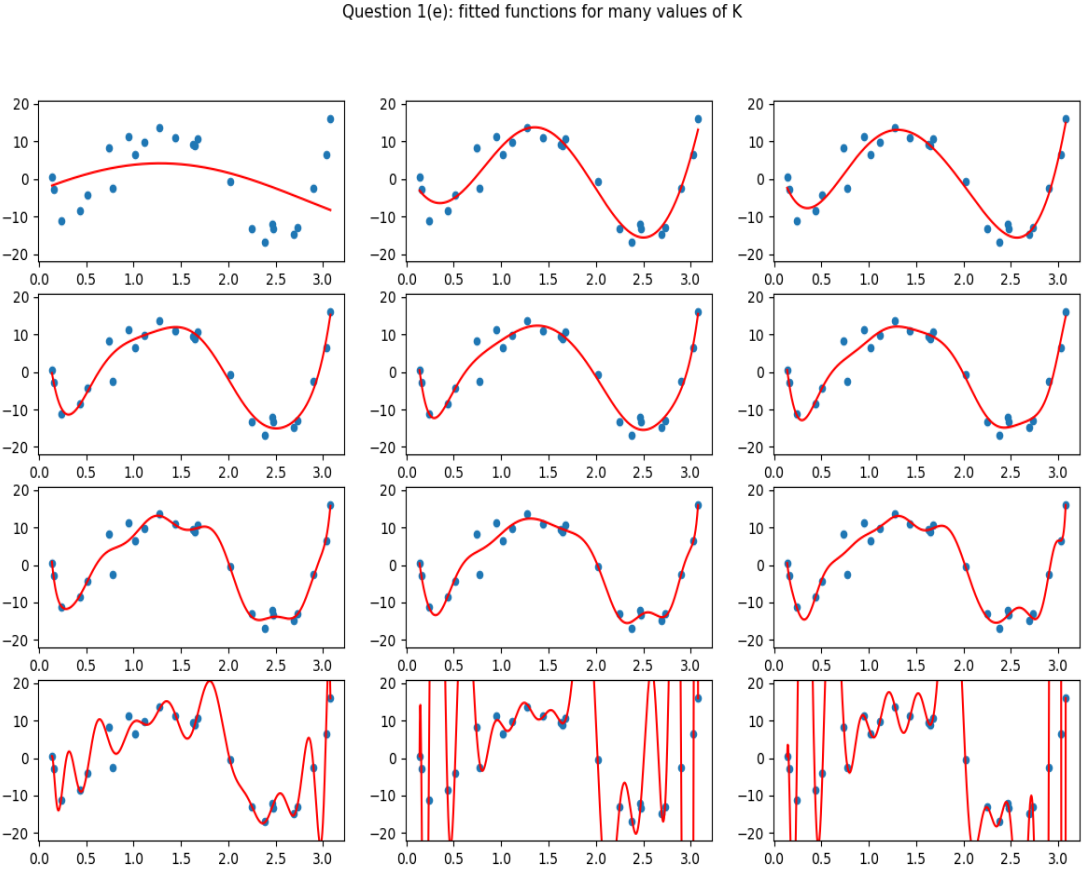
(c) (3 points) Repeat part (b) using `K=9`. Title the figure appropriately. You should be able to do this in at most 2 lines of highy-readible code.

(d) (3 points) Repeat part (b) using `K=12`. Title the figure appropriately. You should be able to do this in at most 2 lines of highy-readible code. If everything is working correctly, the fitted curve should swing wildly up and down, but it should appear to pass through each training point exactly. Likewise, the training error should be almost zero, while the test error is huge. Many of the weights should also be gargantuan. This is extreme over fitting.

(e) (5 points) Run your function `fit_plot` on the training and test data in the file `dataA2Q1.pickle` for values of `K` from 1 to 12 inclusive. Using `subplot`, arrange the 12 plots in a 4×3 grid with `K` increasing from left to right and top to bottom.

[1] You may need to add the keyword argument `rcond=None` to `lstsq` to supress warning messages.

Figure 2:



You may use a single loop, but no nested loops, for this purpose. Title the figure, *Question 1(e): fitted functions for many values of K*.

Manually put the figure into full-screen mode before you save it, to increase resolution and make the figure more readible. If you have done everything correctly the figure should look approximately like that in Figure 2, which is based on a different training set from the one you are using. Notice that the fitted function underfits the data for `K=1`, and overfits for large values of `K`. As `K` increases, the function becomes more-and-more wiggly, then varies wildly and finally soars far above and below the training data and beyond the limits of the axes. You should notice even more extreme behavior with the data in `dataQ2A1.pickle`.

(f) (10 points) To be realistic, we will pretend in this question that we have only a limited amount of data for training and testing (ignoring the 1,000 test points). To make the best use of limited data, you will use 5-fold cross validation to estimate the best value of `K`. Do not use any built-in functions that do (a significant part of) cross validation, such as `cross_validate` in `sklearn`. Instead, you should implement your own cross-validation procedure from scratch using basic Numpy

operations.. You should test values of `K` from 0 to 12 inclusive.

First, divide the training data into five equal-sized folds. For this question, use the first 5 training points as the first fold, the next five training points as the second fold, etc. Use one fold as validation data, and the remaining four folds as training data. For each value of K, use linear least squares to fit a function to the four folds of training data. Use the one fold of validation data to compute the validation error, and the four folds of training data to compute the training error. Do this five times, using each of the five folds in turn as validation data. Thus, for each value of K, you will have five estimates of validation error, and five estimates of training error. Use their averages as overall estimates of validation and training error, respectively. You may use one doubly-nested loop for this. You should reuse code from part (a), but do not cut-and-paste. Instead, part (a) should have well-designed subroutines that you can use here. You can read more about cross validation in Bishop and in Hastie, Tibshirani and Friedman.

On a single pair of axes, plot mean training error v.s. K in blue, and mean validation error v.s. K in red. Label the horizontal axis *K*, and the vertical axis *Mean error*. Title the figure, *Question 1(f): mean training and validation error*. When plotting the errors, use the function `semilogy` in `pyplot` to put a log scale on the vertical axis. Without this, a few extremely large validation errors will dominate the entire graph, making most of the graph flat and uninformative. You should find that for large values of K, the validation error is extremely large and the training error is almost 0. If everything is working correctly, the plotted curves should look approximately like those in Figure 3, which is for a different data set than the one you are using.

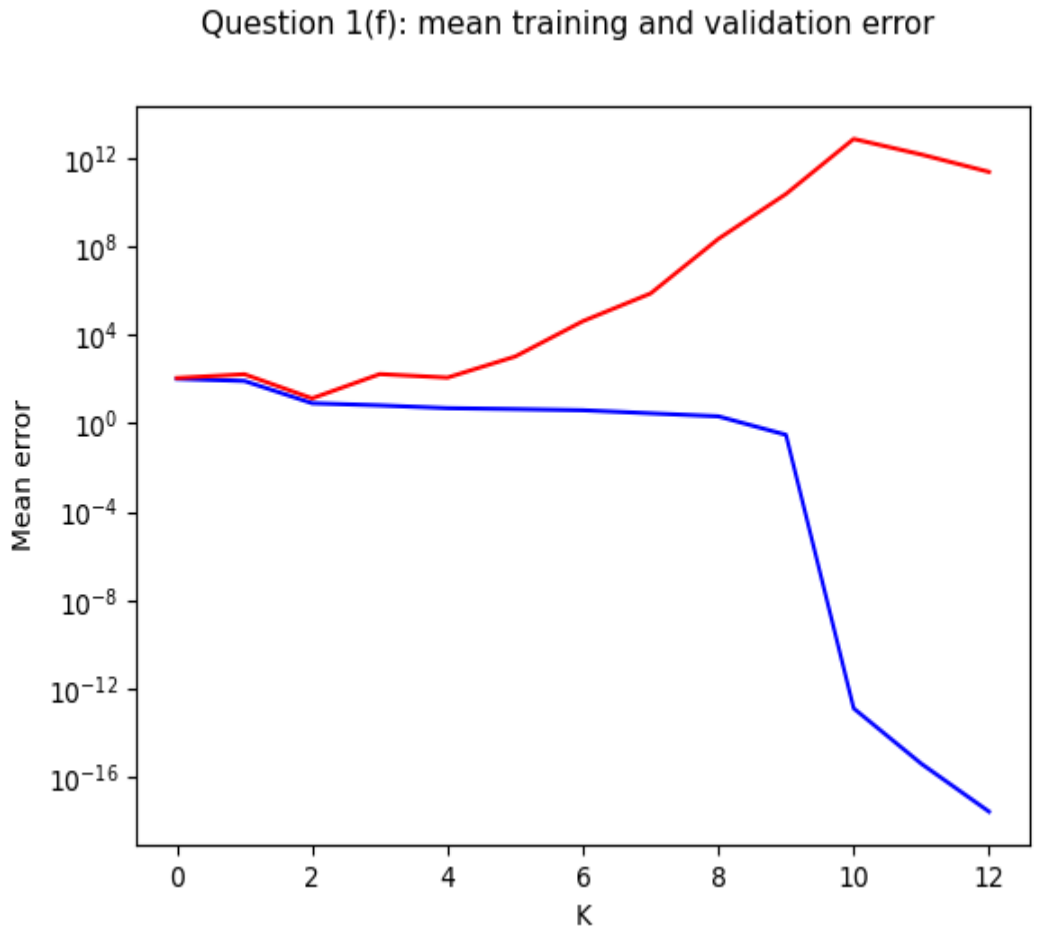
Choose the value of K with the smallest mean validation error. Using this value of K, repeat part (b). Use all the training data (five folds) and all the test data (1,000 points). This maximizes the use of the training data for the final classifier, and gives an unbiased estimate of test error (though in practice we would have far fewer than 1,000 test points). Title the plot, *Question 1(f): the best fitting function*.

2. *Probabilistic Multi-class Classification.* (30 points total)

In this question, you will use three different probabilistic classifiers to do multi-class classification on 2-dimensional (2D) cluster data. You will compute the accuracies of the classifiers, and you will plot the decision boundaries. In the process, you will compare the generative and discriminative approaches to classification. For full marks, you should not do any unnecessary calculations. In particular, you often do not have to fully compute output probabilities into order to make predictions and compute accuracy. Besides increasing execution time, such unnecessary calculations can introduce additional numerical error which can reduce prediction accuracy.

The data you will use is in the file `dataA2Q2.pickle.zip` on the course web site. Download and uncompress this file. The file contains training and test data. Next, start the Python interpreter and import the `pickle` module. You can then read the file and extract the data with the following Python commands:

Figure 3:


Question 1(f): mean training and validation error
Mean error
10^{12}
10^{8}
10^{4}
10^{0}
10^{-4}
10^{-8}
10^{-12}
10^{-16}
0
2
4
6
8
10
12
K

```
with open('dataA2Q2.pickle','rb') as file:
    dataTrain,dataTest = pickle.load(file)
Xtrain,Ttrain = dataTrain
Xtest,Ttest = dataTest
```

The variables `Xtrain` and `Ttrain` will now contain training data, that is, 2D input points and corresponding target values, respectively. Likewise, `Xtest` and `Ttest` contain test data. There are 1,800 training points and 1,800 test points. The training data is illustrated in the scatter plot in Figure 4. The figure shows three overlapping clusters coloured red, blue and green, which are called clusters 0, 1 and 2, respectively, in the target data.

In addition, download the file `bonnerlib2D.py.zip` from the course web site and import it into your program with the statement `import bonnerlib2D as bl2d`. It contains functions for displaying 2-dimensional decision boundaries.

In answering the questions below, do 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 vectorized. To receive any points for a question, it must produce output that is at least approximately correct. To receive full marks, the output must be exactly correct and you must meet all constraints on number of loops, lines of code, vectorization and readibility. To receive any points for part (a), it must produce correct plots, as demonstrated in the other parts.

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

Define a function `plot_data(X,T)` to plot data `X,T`. Here `X` is a data matrix in which each row is a 2-dimensional data point, and `T` is a vector of corresponding class labels. You may assume there are three possible class labels: 0, 1 and 2. You should use the `scatter` function to plot each point in `X` as a dot of size of 2. Colour the dots red, blue or green for classes 0, 1 and 2, respectively. Set the limits of the axes using the functions `xlim` and `ylim` so that the plot holds all the training points with a margin of 0.1, *i.e.*, so that the distance from the edge of the plot to the nearest training point is 0.1. When you plot the training data in the file `dataA2Q2.pickle`, it should look exactly like Figure 4. (Do not hand this in.) The function can be written in at most 7 lines of highly readible code (not counting comments or blank lines).

(b) *Logistic Regression.* (5 points)

As in Question 4 of Assignment 1, use the Python class `LogisticRegression` in `sklearn.linear_model` to define a classifier. This time, use the keyword arguments `multi_class='multinomial'` and `solver='lbfgs'`, to get multi-class classification. Use the `fit` method to train the classifier on the data that you read in above.

Compute the accuracy of your classifier on the test data. Do this in two ways: (1) using the `score` method of the `LogisticRegression` class, and (2) using

Figure 4:

the weight matrix and bias vector of the classifier to make predictions. Call the two estimates of accuracy `accuracy1` and `accuracy2`, respectively. In addition, generate a plot of the training data with the decision boundaries of the classifier superimposed on top.

To compute `accuracy2`, define a Python function `accuracyLR(clf,X,T)` that computes and returns the accuracy of classifier `clf` on data `X,T`, where `clf` does logistic regression. Your function should not use any methods of `LogisticRegression` or any functions in `sklearn`. The point is to implement everything yourself from scratch starting from the fitted linear functions, one linear function for each class. The linear functions are defined by a weight matrix and bias vector, which are stored in the attributes `coef_` and `intercept_`, respectively, of the classifier. The function `accuracyLR` should use them to make predictions on the data. The accuracy is then the average number of correct predictions. The function should work on classifiers with any number of classes, not just 3 classes, and on data of any dimensionality, not just 2 dimensions. It can be written in at most 6 lines of highly-readible code (not counting comment lines or blank lines). Do not use any loops.

After computing `accuracy1` and `accuracy2`, print their values and their difference. If everything is working properly, they should have exactly the same value, and the difference should be 0.

To generate the plot, use the function `plot_data` from part (a) to plot the training data, and use the function `boundaries(clf)` in `bonnerlib2D` to draw the decision boundaries of the classifier. Be sure to call `plot_data` before you call `bounaries`, to set the limits on the axes. If you have done everything properly, the decision boundaries should appear as three black line segments meeting at a common point between the three clusters. Title the figure, *Question 2(b): decision boundaries for logistic regression.*

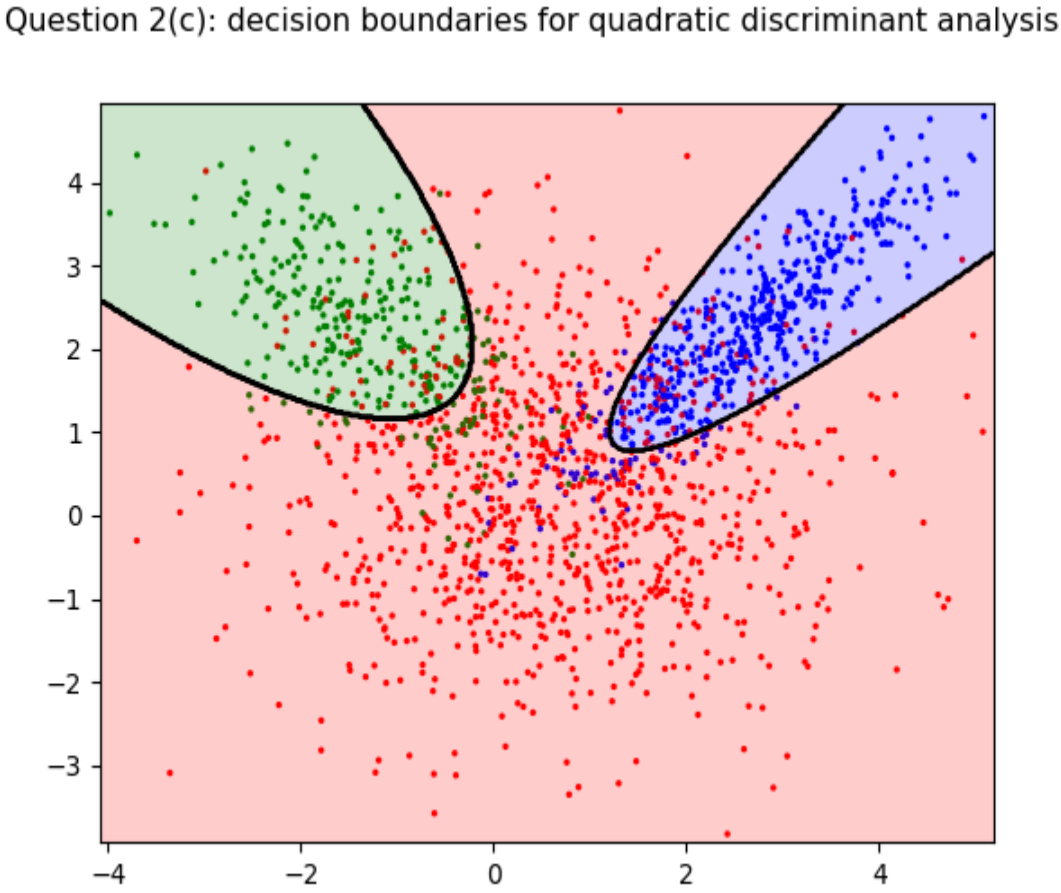
This entire question can be implemented as a Python script with at most 11 lines of highly-readible code, not counting the code for `accuracyLR` and `plot_data`.

(c) *Gaussian (aka Quadratic) Discriminative Analysis.* (10 points)

Repeat part (b) using the Python class `QuadraticDiscriminantAnalysis` in `sklearn.discriminant_analysis` to define the classifier. Use the keyword argument `store_covariance=True`, to store the covariance matrices for each class. Compute the values of `accuracy1` and `accuracy2`, and generate a plot of the training data with the decision boundaries of the classifier superimposed on top.

To compute `accuracy2`, define a function `accuracyQDA(clf,X,T)` that computes and returns the accuracy of classifier `clf` on data `X,T`, where `clf` does quadratic discriminant analysis. Your function should not use any methods of `QuadraticDiscriminantAnalysis` or any functions in `sklearn`. The point is to implement everything yourself from scratch starting from the fitted class models. The class models consist of the mean vector, covariance matrix and prior probability of each class. They are stored in the attributes of the classifier. You should use the class models and Bayes rule to make predictions, from which you

Figure 5:



should compute accuracy. You may use the function `multivariate_normal.pdf` in `scipy.stats` to compute probability densities for each class. Do not use any other functions in `scipy.stats`. Also, the function `accuracyQDA` should work on classifiers with any number of classes, not just 3 classes, and on data of any dimensionality, not just 2 dimensions. It can be written in at most 13 lines of highly-readible code. You may use 1 loop, and no nested loops.

After computing `accuracy1` and `accuracy2`, print their values and their difference. If everything is working properly, they should have exactly the same value, and the difference should be 0.

Finally, plot the training data and decision boundaries. Title the figure, *Question 2(c): decision boundaries for quadratic discriminant analysis.* If everything is working correctly, it should look approximately like Figure 5, which is based on a data set with slightly different properties from the one you are using. This entire question can be implemented as a Python script with at most 11 lines of highly-readible code, not counting the code for `accuracyQDA` and `plot_data`.

(d) *Gaussian Naive Bayes.* (10 points)

Repeat part (b) using the Python class `GaussianNB` in `sklearn.naive_bayes` to define the classifier. Compute the values of `accuracy1` and `accuracy2`, and generate a plot of the training data with the decision boundaries of the classifier superimposed on top.

To compute `accuracy2`, define a function `accuracyNB(clf,X,T)` that computes and returns the accuracy of classifier `clf` on data `X,T`, where `clf` does Gaussian naive Bayes. Your function should not use any methods of `GaussianNB` or any functions in `sklearn`. The point is to implement everything yourself from scratch starting from the fitted class models. The class models consist of the mean and variance of each feature of each class, and of the prior probability of each class. They are stored in the attributes of the classifier.[2] You should use the class models and Bayes rule to make predictions, from which you should compute accuracy. Do not use any functions in `numpy.random` or `scipy.stats`. Also, your function `accuracyNB` should work on classifiers with any number of classes, not just 3 classes, and on data of any dimensionality, not just 2 dimensions. It can be written in at most 16 lines of highly-readible code. Do not use any loops.

Recall from the lecture slides that in Gaussian naive Bayes, if $x = (x_1, ..., x_d)$ is a data point, then its class-conditional densities are given by

$$P(x|t=k) \;=\; \Pi_i P(x_i|t=k) \qquad \text{where} \qquad P(x_i|t=k) \;=\; \frac{e^{-(x_i-\mu_{ik})^2/2\sigma_{ik}^2}}{\sqrt{2\pi}\sigma_{ik}}$$

The first equation says that within class k, the features of x are independent, and the second equation says that in class k, feature x_i has a Gaussian distribution with mean μ_{ik} and variance σ_{ik}^2. You should use these formulas in the function `accuracyNB`. You can use broadcasting to implement them without using loops (See the Tips on Scientific Programming in Python).

After computing `accuracy1` and `accuracy2`, print their values and their difference. If everything is working properly, they should have exactly the same value, and the difference should be 0.

Finally, plot the training data and decision boundaries. Title the figure, *Question 2(d): decision boundaries for Gaussian naive Bayes.* If everything is working correctly, it should look similar to the plot in part (c) but the two decision boundaries should be much more circular (but not perfect circles). This entire question can be implemented as a Python script with at most 11 lines of highly-readible code, not counting the code for `accuracyNB` and `plot_data`.

3. *Neural Networks: intro* (35 points total)

   This question is a warm-up execise to familiarize you with neural networks before you implement them yourself in Questions 4 and 5. Here, you will fit a variety of neural networks to the data from Question 2, explore their properties, and write code that does forward propagation from inputs to outputs. We focus here on neural networks for classification, not regression.

[2]Note: the attribute `clf.sigma_` is variance, not standard deviation.

To define a neural net, use the Python class `MLPClassifier` in `sklearn.neural_network`. In this question, all neural networks will have a single hidden layer that uses the logistic activation function. You should train them using stochastic gradient descent (sgd) with an initial learning rate of 0.01 and an optimization tolerance of 10^{-6}. These are parameters of `MLPClassifier` that you can set. All other parameters should use their default values. When training a neural net, be sure that the training data uses integer encodings for the class labels, *not* 1-of-K encodings, as `MLPclassifier` expects integer labels for multi-class classification.[3]

In answering the questions below, do not use any Python loops, except for generating subplots. 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. During training, you will probably get a warning message about convergence. You can ignore these messages, but please remove them from your printout before handing it in. To receive any points for a question, it must produce output that is at least approximately correct. To receive full marks, the output must be exactly correct and you must meet all constraints on number of loops, lines of code, vectorization and readibility.

(a) (0 points) Reload the data from the file `dataA2Q2.pickle`.

(b) (3 points) Because the weights of a neural network are initialized randomly, set the seed for random number generation to 0 by calling `numpy.random.seed(0)`, so that everyone gets the same initial weights, and thus the same results. Then define a neural network as specified above. Also specify 1 hidden unit and a maximum of 1000 epochs of training. Fit the neural network to the training data using the `fit` method. Compute the accuracy of the trained network on the test data using the `score` method. Print the test accuracy. Plot the training data and the decision boundaries as in Question 2. If everything is working correctly, the decision boundary should be a straight line. Title the figure, *Question 3(b): Neural net with 1 hidden unit.* Design the code so that it will help you with later questions.

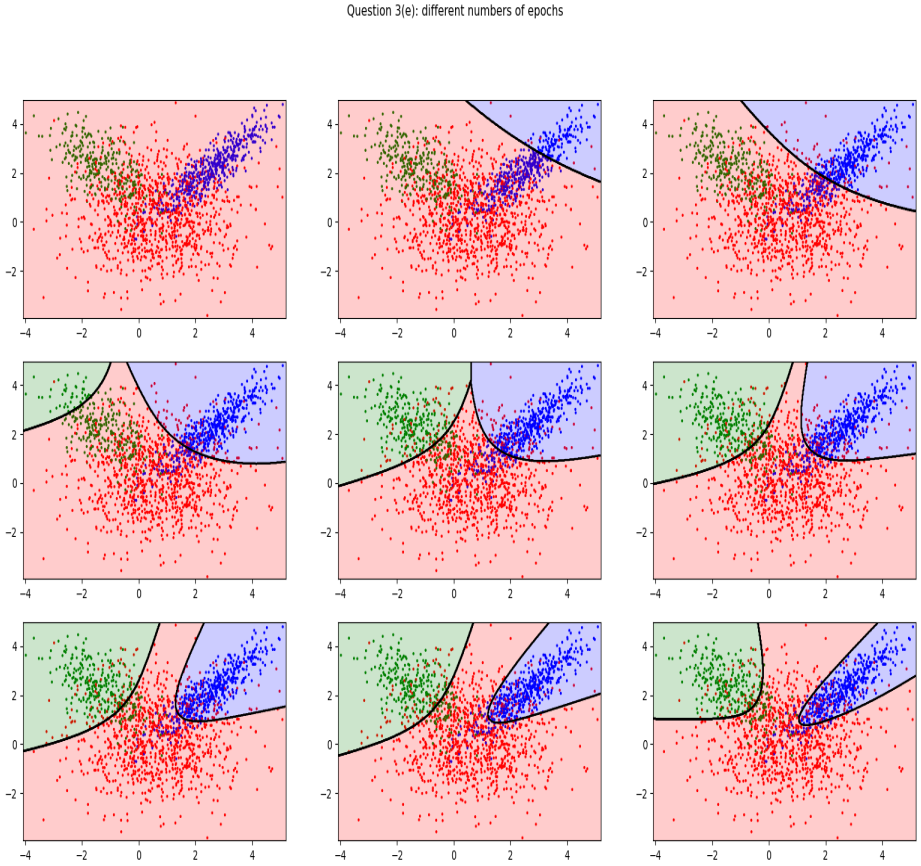
(c) (3 points) Repeat part (b) for a neural net with 2 hidden units. Title the figure appropriately. This can be done with at most 3 lines of highly-readible code. If everything is working correctly, you should see two curved decision boundaries that divide the input space into three decision regions that roughly correspond to the three clusters.

(d) (3 points) Repeat part (b) for a neural net with 9 hidden units. Title the figure appropriately. This can be done with at most 3 lines of highly-readible code. If everything is working correctly, the figure shold look similar (but not identical) to that of Question 2(c).

(e) (5 points) To see how the decision boundary evolves during learning, repeat part (b) nine times using a neural net with 7 hidden units, but with increasing amounts

[3] `MLPclassifier` assumes that class labels encoded as binary arrays are meant for *multi-label* classification, something we have not studied, and which is different from *multi-class* classification.

Figure 6:



of training. Specifically, use 2^2, 2^3, 2^4, ... 2^{10} training iterations. Use `subplot` to arrange the nine plots in a 3×3 grid in a single figure, with the number of iterations increasing from left to right and top to bottom. Title the figure, *Question 3(e): different numbers of epochs*. This can be done with at most 5 lines of highly-readible code.

Manually put the figure into full-screen mode before saving it, to increase resolution and make the figure more readible. If everything is working correctly, the results should look similar to those in Figure 6, which is based on a dataset with slightly different properties from the one you are using. In your case, the decision boundaries for 2^{10} iterations (in the bottom right) should look much more similar to those in part (d). Nine test accuracies should also be printed out, generally increasing with the number of iterations (but not always).

(f) (5 points) To see the effect of the initial weights, repeat part (b) nine times using a neural net with 5 hidden units, but using nine different sets of initial weights. To do this, use a different seed for random number generation each time. Specifically, use the numbers 1,2,...,9 as seeds, in that order. Use `subplot` to arrange the nine

plots in a 3×3 grid in a single figure, with the seed value increasing from left to right and top to bottom. Title the figure, *Question 3(f): different initial weights*. This can be done with at most 5 lines of highly-readible code.

Manually put the figure into full-screen mode before saving it, to increase resolution and make the figure more readible. If everything is working correctly, you should get slightly different decision boundaries each time (or most times), all somewhat similar to those in part (d). Nine test accuracies should also be printed out, not all the same.

(g) (6 points) Following the approach in Question 2, compute the test accuracy of a neural network in two ways: (1) by using the `score` method of the `MLPClassifer` class, and (2) by implementing forward propagation within the neural net to make predictions. Call the two estimates of accuracy `accuracy1` and `accuracy2`, respectively.

To compute `accuracy2`, define a Python function `accuracyNN(clf,X,T)` that computes and returns the accuracy of classifier `clf` on data `X,T`, where `clf` is a neural network with one hidden layer. Your function should not use any methods of `MLPClassifier` or any functions in `sklearn`. The point is to implement everything yourself from scratch using the fitted weight matrices and bias vectors of the neural net, which are stored in the attributes of `clf`. The function `accuracyNN` should use them to propagate the inputs of the neural net to the outputs, and to make predictions. The function should work for neural nets with any number of output classes, not just 3 classes, and for data of any dimensionality, not just 2 dimensions. It can be written in at most 8 lines of highly-readible code (not counting comment lines or blank lines). Do not use any loops.

As in part (d), define a neural network with 9 hidden units and at most 1,000 epochs of training. Fit the network to the training data. Be sure to set the seed for random number generation to 0 before you start. After computing `accuracy1` and `accuracy2`, print their values and their difference. If everything is working properly, they should have exactly the same value, and the difference should be 0. This question can be implemented as a Python script with at most 8 lines of highly-readible code, not counting the code for `accuracyNN`. The line that uses `MLPClassifier` to define the neural network counts as 1 line of code, even if it is spread over several lines to improve readibility.

(h) (10 points) Repeat part (g), but compute the mean cross entropy of the neural net on the test data, instead of the accuracy, and do so in two different ways. Call the resulting cross entropies `CE1` and `CE2`.

Specifically, define a function `ceNN(clf,X,T)` that computes `CE1` and `CE2` for classifier `clf` on data `X,T`, where `clf` is a neural net with one hidden layer. For each input vector in `X`, first compute the log probability of each output class, and do this in two ways. For `CE1`, use the `predict_log_proba_` method of `clf`. For `CE2`, do not use any methods of `clf` or any functions in `sklearn`. Instead, compute the log probabilities yourself after first propagating the inputs of the neural net to the output, as in part (g). From the two log probabilities, compute and return the two cross entropies. Your function should work for neural nets with any number

of output classes, not just 3 classes, and for data of any dimensionality, not just 2 dimensions. It can be written in at most 22 lines of highly-readible code (not counting comment lines or blank lines). Do not call any subroutines from part (g), but feel free to cut-and-paste as much code as you like. You may use any of the attributes of `clf`. Do not use any loops. Hint: use broadcasting and indexing to compute a one-hot encoding of `T`.

As in part (g), define a neural network with 9 hidden units and at most 1,000 epochs of training. Fit the network to the training data. Be sure to set the seed for random number generation to 0 before you start. After using `ceNN` to compute `CE1` and `CE2`, print their values and their difference. Print the cross entropies out in full (about 17 significant digits). If everything is working properly, they should have exactly the same value (approximately 0.43), and the difference should be 0 (or almost 0, depending on your machine). This question can be implemented as a Python script with at most 7 lines of highly-readible code, not counting the code for `ceNN`. The line that uses `MLPClassifier` to define the neural network counts as 1 line of code, even if it is spread over several lines to improve readibility.

4. *Neural Networks: theory* (52 points total)

In this question, you will develop matrix equations needed to implement a neural net in Question 5.

The neural net: The net has two hidden layers and a single output unit, where the hidden units use a tanh activation function. We will use the network for binary classification, so the activation function at the output is a sigmoid. The operation of the neural net can be specified as follows:

$$o = \sigma(\tilde{g}) \qquad \tilde{g} = gU + u_0 \tag{4}$$

$$g = \tanh(\tilde{h}) \qquad \tilde{h} = hV + v_0 \tag{5}$$

$$h = \tanh(\tilde{x}) \qquad \tilde{x} = xW + w_0 \tag{6}$$

Here x, h, g and o are row vectors representing the input, the first hidden layer, the second hidden layer, and the output, respectively; $\tilde{g}$, $\tilde{h}$ and $\tilde{x}$ are row vectors representing linear transformations of g, h and x, respectively; U, V and W are weight matrices; u_0, v_0 and w_0 are row vectors representing bias terms; and σ and tanh are the sigmoid function and the hyperbolic tangent function, respectively. Because there is only a single output, the output vector, o, has only a single element, so we will treat o as a real number. Likewise for $\tilde{g}$. Similarly, the weight matrix U has only a single column, so we will treat U as a column vector.

Recall that the tanh function is given by

$$\tanh(y) \;=\; \frac{e^y - e^{-y}}{e^y + e^{-y}}$$

and the sigmoid function is given by $\sigma(y) = 1/(1 + e^{-y})$, for any real number, y. In this question, you may use the following properties of the tanh and sigmoid functions without proving them:

$$\frac{\partial \tanh(y)}{\partial y} = 1 - [\tanh(y)]^2 \tag{7}$$

$$\frac{\partial \sigma(y)}{\partial y} = \sigma(y)[1 - \sigma(y)] \tag{8}$$

You may not use any other results from the lecture slides, although you may copy their proofs if you find them useful.

Since we are using the neural network for classification, the loss function used during training is cross entropy:

$$C = \sum_n c(t^{(n)}, o^{(n)}) \qquad \text{where} \qquad c(t, o) = -t \log o - (1 - t) \log(1 - o) \tag{9}$$

where the sum is over training points. Here, $o^{(n)}$ is the output of the neural net on input $x^{(n)}$, and $t^{(n)}$ is a binary value representing the target class of $x^{(n)}$.

The chain rule: Below, you will derive gradient equations for back propagation in the neural net. Back propagation is based on the chain rule, and since each layer has multiple inputs and outputs, we will need a generalization of the chain rule for multi-variate functions. Abstractly, suppose that $y = g(x)$ and $z = h(x)$, where x, y and z are all real numbers. Then, the multi-variate chain rule says that

$$\frac{\partial f[g(x), h(x)]}{\partial x} = \frac{\partial f(y, z)}{\partial y}\frac{\partial y}{\partial x} + \frac{\partial f(y, z)}{\partial z}\frac{\partial z}{\partial x}$$

for any differentiable function, $f(y, z)$. Intuitively, this says that x affects f through y and through z, and that both these effects must be added up to determine the total effect on f due to a change in x.

To put this in the context of our neural net, let $\tilde{h}_j$ and h_i be components of the vectors $\tilde{h}$ and h, respectively. Then Equation (5) is equivalent to the following:

$$g_j = \tanh \tilde{h}_j \qquad\qquad \tilde{h}_j = \sum_i h_i V_{ij} + v_{0j} \tag{10}$$

for all j. The right-hand equality shows that changes in the value of a given h_i affect *all* the $\tilde{h}_j$. All these separate effects must be added up to determine the total effect on the output of the neural net due to changes in h_i. Formally,

$$\frac{\partial c}{\partial h_i} = \sum_j \frac{\partial c}{\partial \tilde{h}_j}\frac{\partial \tilde{h}_j}{\partial h_i} \tag{11}$$

where c is an abreviation for the cross entropy, $c(t, o)$.

Note that we do not always need this multivariate chain rule. For example, the left-hand equality in Equation (10) shows that $\tilde{h}_j$ affects g_j, but no other g_i. So, the univariate chain rule is all we need here:

$$\frac{\partial c}{\partial \tilde{h}_j} = \frac{\partial c}{\partial g_j} \frac{\partial g_j}{\partial \tilde{h}_j} \tag{12}$$

Intuitively, this says that $\tilde{h}_j$ affects c only through g_j. In this question, you will have to think carefuly about the dependencies within a neural net in order to determine which chain rule to use. Be sure to make it clear in your proofs which chain rule you are using, and why.

What to do: Prove each of the five matrix equations below from scratch, using only the results stated above, along with basic results from calculus and linear algebra. Use the proofs in the solutions to Assignment 1 and the midterm as a model and guide for how your proofs should be done. Please also see the Tips on Proving Theorems on page 4 of this assignment. To make your proofs easier to mark, use the indices m, n to range over training instances, and use i, j, k to range over hidden and input units, as in the equations above. In the matrix equations below, X, H, $\tilde{H}$, and G are data matrices whose n^{th} rows are $x^{(n)}$, $h^{(n)}$, $\tilde{h}^{(n)}$ and $g^{(n)}$, respectively. Likewise, $\tilde{G}$, O and T are column vectors whose n^{th} elements are $\tilde{g}^{(n)}$, $o^{(n)}$ and $t^{(n)}$, respectively. $\vec{1}$ is a row vector of 1's. Except for part (c), all multiplications are matrix multiplications.

Note that your proofs must be rooted in Equations (4) to (6), which describe the behaviour of the neural net. These equations are vector-based, since a neural net operates on one vector at a time. The neural net in this question takes a single vector, x, as input, generates a vector of hidden values, h, from which it generates another vector of hidden values, g, from which it generates a single real number, o, as output. From this vector-based description, you must derive matrix equations that describe the behaviour of the neural net on a set of vectors. These equations are used for efficient batch (or mini-batch) training of a neural net.

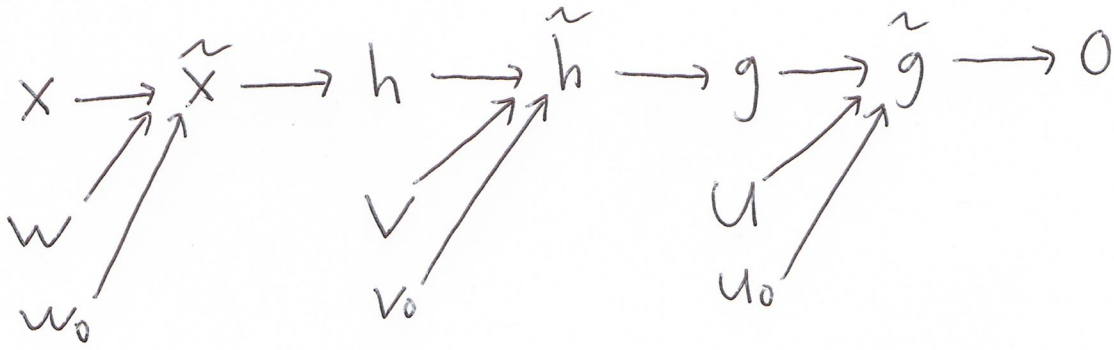
The derivations can be broken down into several steps. The first step is to express Equations (4) to (6) at a finer level of detail, as in Equation (10). Such equations describe the neural net in terms of individual neurons, such as h_i and g_j, and individual weights and bias terms, such as V_{ij} and v_{0j}. There are no vectors or matrices in these equations, only real numbers. For this reason, you can infer derivatives using the familiar rules of calculus for real variables.

The next step is to derive equations for the derivatives needed by back propagation, such as $\partial c/\partial V_{ij}$ and $\partial c/\partial h_i$. These equations should be expressed entirely in terms of real numbers, not vectors or matrices. This step typically involves calculus, including the chain rule, but no linear algebra.

The final step is to vectorize the equations, that is, to translate them into the vector and matrix-based equations given below. This step typically involves linear algebra, but little or no calculus. It will probably be the least familiar step for you, and therefore

the hardest. It will also have the most marks. You can find examples in the solutions to Assignment 1 and the midterm.

(a) (5 points) Draw the computation graph of the neural net. (See slide 36 of Lecture 5). Each node of the graph is a vector or matrix. You may draw the graph by hand and scan it in.



(b) (8 points)

$$\frac{\partial C}{\partial \tilde{G}} = O - T \tag{13}$$

PROOF: First note that

$$\begin{aligned}
\frac{\partial c(t,o)}{\partial o} &= \frac{\partial[-t\log o - (1-t)\log(1-o)]}{\partial o} && \text{by the definition of cross entropy} \\
&= -t\frac{\partial \log o}{\partial o} - (1-t)\frac{\partial \log(1-o)}{\partial o} && \text{by the linearity of differentiation} \\
&= \frac{-t}{o} + \frac{1-t}{1-o} && \text{since } \partial \log x/\partial x = 1/x \\
&= \frac{-t(1-o)+(1-t)o}{o(1-o)} && \text{by basic algebra} \\
&= \frac{o-t}{o(1-o)} && \text{by basic algebra}
\end{aligned}$$

Moreover, $\partial o/\partial \tilde{g} = o(1-o)$, by Equation (8), since $o = \sigma(\tilde{g})$. Thus,

$$\begin{aligned}
\frac{\partial c(t,o)}{\partial \tilde{g}} &= \frac{\partial c(t,o)}{\partial o}\frac{\partial o}{\partial \tilde{g}} && \text{by the chain rule} \\
&= \frac{o-t}{o(1-o)}\,o(1-o) && \text{by the results above} \\
&= o - t && \text{trivially}
\end{aligned}$$

This is almost the result we want. All that remains is to expand it from a single input, x, to a batch of inputs, X, and to vectorize it by replacing real numbers (o, t and $\tilde{g}$) with vectors and matrices (O, T and $\tilde{G}$). The first step is to note that for input $x^{(n)}$, this result becomes

$$\frac{\partial c^{(n)}}{\partial \tilde{g}^{(n)}} = o^{(n)} - t^{(n)} \tag{14}$$

where $c^{(n)}$ is short hand for $c(t^{(n)}, o^{(n)})$. In addition, $\partial c^{(n)}/\partial \tilde{g}^{(m)} = 0$ for $m \neq n$, since $c^{(n)}$ is independent of $\tilde{g^{(m)}}$, since they correspond to different inputs. Thus,

$$\frac{\partial C}{\partial \tilde{g}^{(m)}} = \frac{\partial \sum_n c^{(n)}}{\partial \tilde{g}^{(m)}} = \sum_n \frac{\partial c^{(n)}}{\partial \tilde{g}^{(m)}} = \frac{\partial c^{(m)}}{\partial \tilde{g}^{(m)}} \tag{15}$$

In the sequence of statements below, we transform Equation (14) into Equation (13). Each equation in this sequence follows from the one above, and the

first one follows from Equation (14):

$$\begin{aligned}
\frac{\partial C}{\partial \tilde{g}^{(n)}} &= o^{(n)} - t^{(n)} && \text{by Equation (15)} \\
\frac{\partial C}{\partial \tilde{G}_n} &= O_n - T_n && \text{since } \tilde{g}^{(n)} = \tilde{G}_n, \quad o^{(n)} = O_n, \quad t^{(n)} = T_n \\
\left[\frac{\partial C}{\partial \tilde{G}}\right]_n &= [O - T]_n && \text{by the definitions of gradient and vector addition} \\
\frac{\partial C}{\partial \tilde{G}} &= O - T && \text{since all their components are equal}
\end{aligned}$$

(c) (10 points)

$$\left[\frac{\partial C}{\partial \tilde{H}}\right]_{nk} = (1 - G_{nk}^2)\left[\frac{\partial C}{\partial G}\right]_{nk} \tag{16}$$

PROOF: First, recall that $g_k = \tanh \tilde{h}_k$ by Equation (10), so $\partial g_k / \partial \tilde{h}_k = 1 - g_k^2$ by Equation (7). Combining this with the univariate chain rule in Equation (12), we get

$$\frac{\partial c}{\partial \tilde{h}_k} = \frac{\partial c}{\partial g_k}\frac{\partial g_k}{\tilde{h}_k} = \frac{\partial c}{\partial g_k}(1 - g_k^2)$$

where c is an abreviation for the cross entropy, $c(t, o)$. This is almost the result we want. All that remains is to expand it from a single input, x, to a batch of inputs, X, and to vectorize it by replacing real numbers (h_k, $\tilde{h}_k$ and g_k) with matrices (H, $\tilde{H}$ and G). The first step is to note that for input $x^{(n)}$, this result becomes

$$\frac{\partial c^{(n)}}{\partial \tilde{h}_k^{(n)}} = \frac{\partial c^{(n)}}{\partial g_k^{(n)}}\left(1 - [g_k^{(n)}]^2\right) \tag{17}$$

Moreover, as in Equation (15),

$$\frac{\partial C}{\partial \tilde{h}^{(n)}} = \frac{\partial c^{(n)}}{\partial \tilde{h}^{(n)}} \tag{18}$$

$$\frac{\partial C}{\partial g^{(n)}} = \frac{\partial c^{(n)}}{\partial g^{(n)}} \tag{19}$$

In the sequence of statements below, we transform Equation (17) into Equation (16). Each equation in this sequence follows from the one above, and the first one follows from Equation (17):

$$\frac{\partial C}{\partial \tilde{h}_k^{(n)}} = \frac{\partial C}{\partial g_k^{(n)}}\left(1 - [g_k^{(n)}]^2\right) \quad \text{by Equations (18) and (19)}$$

$$\frac{\partial C}{\partial \tilde{H}_{nk}} = \frac{\partial C}{\partial G_{nk}}(1 - G_{nk}^2) \quad \text{since } \tilde{h}_k^{(n)} = \tilde{H}_{nk},\ g_k^{(n)} = G_{nk}$$

$$\left[\frac{\partial C}{\partial \tilde{H}}\right]_{nk} = \left[\frac{\partial C}{\partial G}\right]_{nk}(1 - G_{nk}^2) \quad \text{by the definition of gradient}$$

(d) (12 points)

$$\frac{\partial C}{\partial V} \;=\; H^T \frac{\partial C}{\partial \tilde{H}}$$

PROOF: First, recall from Equation (10) that $\tilde{h}_j = \sum_i h_i V_{ij} + v_{0j}$, so $\partial \tilde{h}_j / \partial V_{ij} = h_i$. Moreover, this is the only way in which V_{ij} affects the output of the neural net and the cross entropy, so the univariate chain rule applies. Thus,

$$\frac{\partial c}{\partial V_{ij}} \;=\; \frac{\partial c}{\partial \tilde{h}_j} \frac{\partial \tilde{h}_j}{\partial V_{ij}} \;=\; \frac{\partial c}{\partial \tilde{h}_j} h_i$$

where c is an abreviation for $c(t, o)$. This is almost the result we want. All that remains is to expand it from a single input, x, to a batch of inputs, X, and to vectorize it by replacing real numbers (h_i, $\tilde{h}_j$ and V_{ij}) with matrices (H, $\tilde{H}$ and V). The first step is to note that for input $x^{(n)}$, this result becomes

$$\frac{\partial c^{(n)}}{\partial V_{ij}} \;=\; \frac{\partial c^{(n)}}{\partial \tilde{h}_j^{(n)}} h_i^{(n)} \tag{20}$$

Keeping this in mind, we vectorize as follows:

$$\begin{aligned}
\left[\frac{\partial C}{\partial V}\right]_{ij} &= \frac{\partial C}{\partial V_{ij}} && \text{by the definition of gradient} \\
&= \frac{\partial \sum_n c^{(n)}}{\partial V_{ij}} && \text{by Equation (9)} \\
&= \sum_n \frac{\partial c^{(n)}}{\partial V_{ij}} && \text{trivially} \\
&= \sum_n \frac{\partial c^{(n)}}{\partial \tilde{h}_j^{(n)}} h_i^{(n)} && \text{by Equation (12)} \\
&= \sum_n \frac{\partial C}{\partial \tilde{h}_j^{(n)}} h_i^{(n)} && \text{by Equation (18)} \\
&= \sum_n \frac{\partial C}{\partial \tilde{H}_{nj}} H_{ni} && \text{since } \tilde{h}_j^{(n)} = \tilde{H}_{nj}, \quad h_i^{(n)} = H_{ni} \\
&= \sum_n H_{ni} \frac{\partial C}{\partial \tilde{H}_{nj}} && \text{trivially} \\
&= \sum_n [H^T]_{in} \left[\frac{\partial C}{\partial \tilde{H}}\right]_{nj} && \text{by the definitions of matrix transpose and gradient} \\
&= \left[H^T \frac{\partial C}{\partial \tilde{H}}\right]_{ij} && \text{by the definition of matrix multiplication}
\end{aligned}$$

It follows that $\partial C / \partial V = H^T \, \partial C / \partial \tilde{H}$ since all their components are equal.

(e) (5 points)

$$\frac{\partial C}{\partial v_0} \;=\; \vec{1}\,\frac{\partial C}{\partial \tilde{H}}$$

PROOF: The proof in part (d) could be repeated almost verbatim. Instead, we shall extract an equation from the proof and use it as a kind of lemma. To do this, we need to treat bias terms as weights. So we extend the weight matrix V with a new row containing the bias vector v_0, and we extend the vector of hidden values h with a new component containing a 1. We call these extensions row 0 and component 0, respectively. With them, Equation (5) becomes $\tilde{h} = Vh$, where $h_0 = 1$ and $V_{0j} = v_{0j}$. Note that this means the matrix of hidden values H is extended with a column of 1s. We call this extension column 0, so $H_{n0} = 1$.

Next, we extract the following equation from the last sequence of equalities in the proof of part (d):

$$\frac{\partial C}{\partial V_{ij}} \;=\; \sum_n H_{ni}\,\frac{\partial C}{\partial \tilde{H}_{nj}} \tag{21}$$

We will use this equation with $i = 0$.

Finally, we note that $[\,\vec{1}\,]_n = 1$ since $\vec{1}$ is a vector of 1s.

Keeping all this in mind,

$$\begin{aligned}
\left[\frac{\partial C}{\partial v_0}\right]_j &= \frac{\partial C}{\partial v_{0j}} && \text{by the definition of gradient}\\
&= \frac{\partial C}{\partial V_{0j}} && \text{since } v_{0j} = V_{0j} \text{ as noted above}\\
&= \sum_n H_{n0}\,\frac{\partial C}{\partial \tilde{H}_{nj}} && \text{by Equation (21) with } i = 0\\
&= \sum_n [\,\vec{1}\,]_n\,\frac{\partial C}{\partial \tilde{H}_{nj}} && \text{since } H_{n0} = 1 = [\,\vec{1}\,]_n \text{ as noted above}\\
&= \sum_n [\,\vec{1}\,]_n \left[\frac{\partial C}{\partial \tilde{H}}\right]_{nj} && \text{by the definition gradient}\\
&= \left[\vec{1}\,\frac{\partial C}{\partial \tilde{H}}\right]_j && \text{by the definition of matrix-vector multiplication}
\end{aligned}$$

It follows that $\partial C/\partial v_0 = \vec{1}\,\partial C/\partial \tilde{H}$ since all their components are equal.

(f) (12 points)

$$\frac{\partial C}{\partial H} = \frac{\partial C}{\partial \tilde{H}} V^T \tag{22}$$

PROOF: Recall again from Equation (10) that $\tilde{h}_j = \sum_i h_i V_{ij} + v_{0j}$, so $\partial \tilde{h}_j / \partial h_i = V_{ij}$. Combining this with the multi-variate chain rule in Equation (11), we get

$$\frac{\partial c}{\partial h_i} = \sum_j \frac{\partial c}{\partial \tilde{h}_j} \frac{\partial \tilde{h}_j}{\partial h_i} = \sum_j \frac{\partial c}{\partial \tilde{h}_j} V_{ij}$$

This is almost the result we want. All that remains is to expand it from a single input, x, to a batch of inputs, X, and to vectorize it by replacing real numbers (h_i, $\tilde{h}_j$ and V_{ij}) with matrices (H, $\tilde{H}$ and V). The first step is to note that for input $x^{(n)}$, this result becomes

$$\frac{\partial c^{(n)}}{\partial h_i^{(n)}} = \sum_j \frac{\partial c^{(n)}}{\partial \tilde{h}_j^{(n)}} V_{ij} \tag{23}$$

Moreover, as in Equation (15),

$$\frac{\partial C}{\partial h^{(n)}} = \frac{\partial c^{(n)}}{\partial h^{(n)}} \tag{24}$$

In the sequence of statements below, we transform Equation (23) into Equation (22). Each equation in this sequence follows from the one above, and the first one follows from Equation (23):

$$\frac{\partial C}{\partial h_i^{(n)}} = \sum_j \frac{\partial C}{\partial \tilde{h}_j^{(n)}} V_{ij} \qquad \text{by Equations (24) and (18)}$$

$$\frac{\partial C}{\partial H_{ni}} = \sum_j \frac{\partial C}{\partial \tilde{H}_{nj}} V_{ij} \qquad \text{since } h_i^{(n)} = H_{ni}, \quad \tilde{h}_j^{(n)} = \tilde{H}_{nj}$$

$$\left[\frac{\partial C}{\partial H}\right]_{ni} = \sum_j \left[\frac{\partial C}{\partial \tilde{H}}\right]_{nj} [V^T]_{ji} \qquad \text{by the definitions of gradient and matrix transpose}$$

$$\left[\frac{\partial C}{\partial H}\right]_{ni} = \left[\frac{\partial C}{\partial \tilde{H}} V^T\right]_{ni} \qquad \text{by the definition of matrix multiplication}$$

$$\frac{\partial C}{\partial H} = \frac{\partial C}{\partial \tilde{H}} V^T \qquad \text{since all their components are equal}$$

Here is a complete set of matrix equations for back propagation in the neural net. You do not have to prove them, but you may use them in Question 5. Each of them is similar to one of the five matrix equations above, and has a similar proof.

$$\frac{\partial C}{\partial \tilde{G}} = O - T$$

$$\frac{\partial C}{\partial U} = G^T \frac{\partial C}{\partial \tilde{G}} \qquad \frac{\partial C}{\partial V} = H^T \frac{\partial C}{\partial \tilde{H}} \qquad \frac{\partial C}{\partial W} = X^T \frac{\partial C}{\partial \tilde{X}}$$

$$\frac{\partial C}{\partial u_0} = \vec{1} \frac{\partial C}{\partial \tilde{G}} \qquad \frac{\partial C}{\partial v_0} = \vec{1} \frac{\partial C}{\partial \tilde{H}} \qquad \frac{\partial C}{\partial w_0} = \vec{1} \frac{\partial C}{\partial \tilde{X}}$$

$$\frac{\partial C}{\partial G} = \frac{\partial C}{\partial \tilde{G}} U^T \qquad \frac{\partial C}{\partial H} = \frac{\partial C}{\partial \tilde{H}} V^T$$

$$\left[\frac{\partial C}{\partial \tilde{H}}\right]_{nk} = (1 - G_{nk}^2) \left[\frac{\partial C}{\partial G}\right]_{nk} \qquad \left[\frac{\partial C}{\partial \tilde{X}}\right]_{nk} = (1 - H_{nk}^2) \left[\frac{\partial C}{\partial H}\right]_{nk}$$

5. *Neural Networks: implementation* (60 points total)

   In this question, you will use the theory developed in Question 4 to write Python programs that train neural networks on the MNIST data. As in Question 4, we will only consider neural nets with two hidden layers, a tanh activation function, and a single logistic (sigmoid) output. In addition, the neural nets in this question will have 100 neurons in each hidden layer. You will implement both batch and stochastic gradient descent. The batch implementation is more straighforward, but as you will see, it takes much longer to converge.

   For comparison, and to provide some quick results, you will first train neural networks on MNIST using `MLPclassifier`. Unless otherwise specified, do not use any functions in `sklearn` other than `MLPclassifier` and its methods. When using `MPLClassifier` to define a neural net, always specify stochastic gradient descent (sgd) as the optimization method, with an optimization tolerance of 10^{-6}. You will not be workng with the full MNIST data set, but with a reduced version consisting of only two digits, since you will be doing binary classification.

   In answering the questions below, do not use any Python loops unless specified otherwise. All code should be vectorized. To receive any points for a question, it must produce output that is at least approximately correct. To receive full marks, the output must be exactly correct and you must meet all constraints on number of loops, lines of code, vectorization and readibility. To receive any marks for parts (a) and (b) they must be at least approximately correct as demonstrated in the output of the other parts.

(a) (4 points) Reload the data from the file `mnistTVT.pickle.zip` and create a reduced data set consisting of just the digits 5 and 6, exactly as in Question 6 of Assignment 1 (except you will not need a validation set). Be sure to preserve the relative order of the digits. Also, since we will be doing binary classification, the target labels should be 0s and 1s. Use 1 to labels the 5s, and 0 to label the 6s. There should be 9444 points in the reduced training set, and 1850 points in the reduced test set. You may cut-and-paste and adapt code from Assignment 1 for this purpose.

You will also need a reduced data set of the digits 4 and 5. Use 1 to label the 4s, and 0 to label the 5s. This data set is for debugging your programs, and no code or results related to it should be handed in.

(b) (7 points) Write a function `evaluateNN(clf,X,T)` that evaluates classifier `clf` on data `X,T`, where `clf` is a neural net defined by `MLPClassifier` as specified above. The function should estimate the accuracy and the cross entropy of the neural net on the data in two different ways, as described in Questions 3(g) and (h), but adapted for binary classification. Call these estimates `accuracy1`, `accuracy2`, `CE1`, `CE2`, as before. The function should return all four of these values. To compute `accuracy2` and `CE2`, you will have to propagate the data in `X` from the input of the neural net to the output. You will test this code in part (c), and use it again in part (d). If you design the subroutines properly, you will also be able to reuse most of the code for computing `accuracy2` and `CE2` in parts (f) and (g). Because this code is already tested, it will simplify your debugging job there.

(c) (6 points) Using `MLPclassifier`, define a neural net as specified in the introduction. Specify an initial learning rate of 0.01, a batch size of 100, and a maximum of 100 iterations of training. If you want to monitor the loss function as training proceeds, then set `verbose=True`, but do not hand in the information printed out. Fit the network to the reduced training data. Be sure to set the seed for random number generation to 0 before you start. Call `evaluateNN` on the trained neural net with the reduced test data. Print the values of `accuracy1` and `accuracy2` and their difference. If everything is working correctly, both values should be exactly the same and the difference should be 0. Likewise, print the values of `CE1` and `CE2` and their difference. If everything is working correctly, both values should be exactly the same and the difference should be 0 (or almost 0, depending on your machine).

If you test your programs by running them on the reduced data set of digits 4 and 5, then the accuracy should be exactly 0.9973319103521878, and the cross entropy should be exactly 0.00891227323537442 (or almost exactly, depending on your machine).[4] Do not hand in these results. Instead, hand in results for the reduced data set of digits 5 and 6.

[4] "Almost exactly" means that all but the last few significant digits should be identical.

(d) (6 points) In this question, you will explore the effect of batch size by training and testing neural nets using different batch sizes. In particular, use `MLPClassifier` to define 14 different neural nets. Use the settings specified in the introduction, and also specify an initial learning rate of 0.001, at most one epoch of training, and batch sizes of 2^k, for k from 0 to 13, inclusive. (You may use one loop, but no nested loops, for this purpose.) Note that this gives batch sizes from $2^0 = 1$ to $2^{13} = 8192$. The latter encompasses almost the entire training set of 9444 samples, so it effectively corresponds to batch gradient descent.

Fit the neural nets to the reduced training data. Be sure to set the seed for random number generation to 0 before training each one. Use `evaluateNN` to compute the accuracy and cross entropy of each trained net on the reduced test data. Record the values of `accuracy2` and `CE2`. Finally, generate two plots: one of `accuracy2` v.s. batch size, and one of `CE2` v.s. batch size. Use a log scale on the horizonal axes. Title the plots, *Question 5(d): Accuracy v.s. batch size*, and *Question 5(d): Cross entropy v.s. batch size*, respectively. Label the horizontal axes, `batch size`, and label the vertical axes `accuracy` and `cross entropy`, respectively.

If everything is working correctly, and if you test your programs by running them on the reduced data set of digits 4 and 5, then the two plots should look exactly like those in Figures 7 and 8. Do not hand these in. Instead, hand in plots for the reduced data set of digits 5 and 6.

(e) (12 points total.) The plots in part (d) show how 1 epoch of training is affected by batch size. The plots show that accuracy decreases with batch size, and cross entropy increases. Explain why this happens (5 points). The graphs show in particular that a batch size of 1 gives the geatest accuracy and lowest cross entropy. Why would we *not* use a batch size of 1 (5 points). To answer this, pay attention during the execution of your programs, and explain any significant observations that do not show up in your graphs. If you do not notice anything, then print something out after each net has been trained. What difference would it make if you could execute your programs on a massively parallel machine, such as a gpu (2 points)?

ANSWER:

Progress per epoch. At each iteration, gradient descent performs a weight update, which intuitively is a step in the downhill direction in weight space, bringing it closer to the minimum of the cost function (the bottom of the hill). Small mini-batches result in more steps per epoch than large ones. Thus, if the gradient from a small mini-batch is accurate, i.e., similar to the gradient from a entire batch, then small mini-batches can get much closer to the minimum in a single epoch, resulting in increased accuracy and decreased cross-entropy, which is what we observe with the MNIST data.

Figure 7:

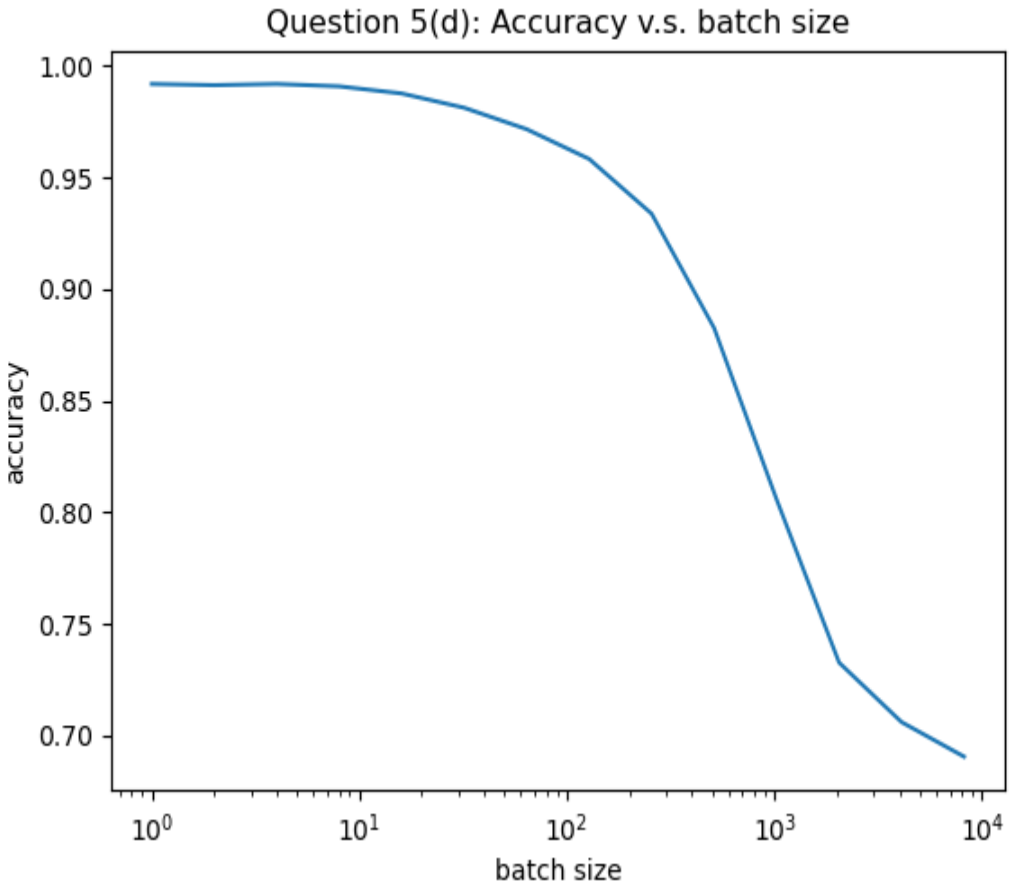
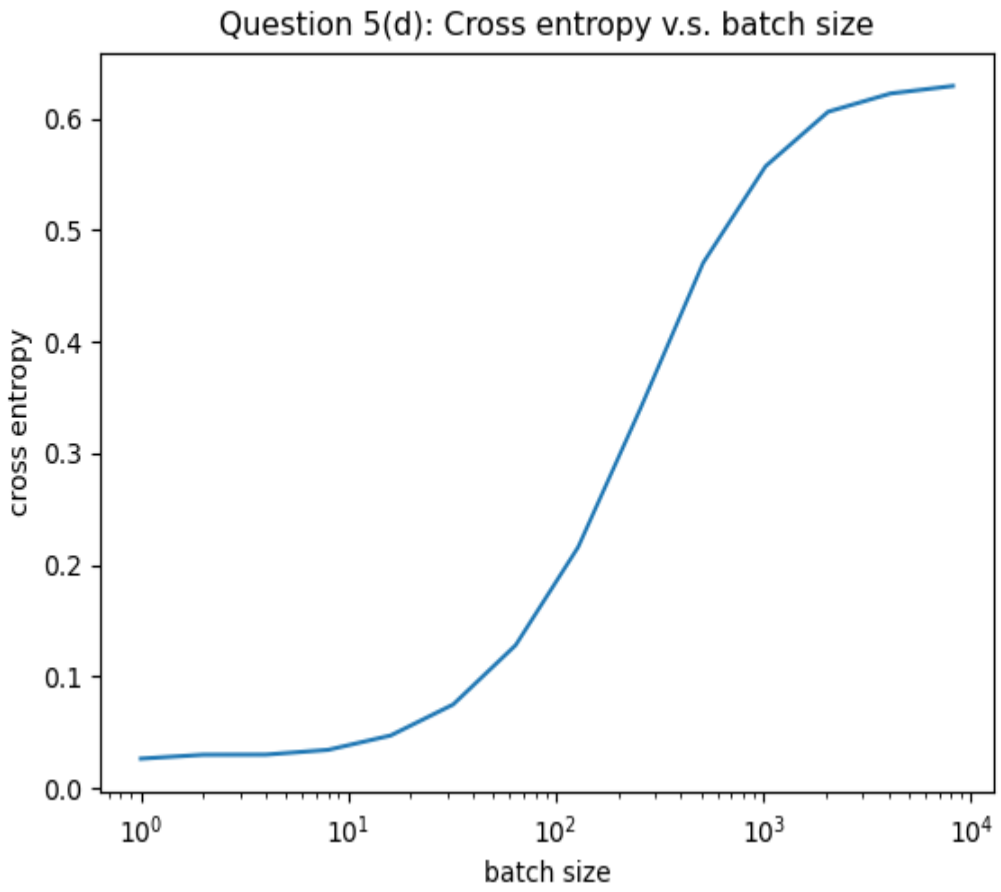

Question 5(d): Accuracy v.s. batch size
accuracy
1.00
0.95
0.90
0.85
0.80
0.75
0.70
10^0
10^1
10^2
10^3
10^4
batch size

Figure 8:


Question 5(d): Cross entropy v.s. batch size
cross entropy
batch size
0.0
0.1
0.2
0.3
0.4
0.5
0.6
10^0
10^1
10^2
10^3
10^4

Note, however, that this may not be the case in general. The gradient may be much less accurate for small mini-batches, because each mini-batch may contain very different data, and thus result in very different gradients. In this case, stochastic gradient descent will tend to wander downhill slowly, zig-zagging back and forth along the way.

However, for the MNIST data (and for many real data sets), things are different. Many of the images of a given digit are very similar, so the gradient from a random sample of images (such as a mini-batch) is not that different from the gradient of a large mini-batch, and is therefore reasonably accurate. Thus, because the gradients are reasonably accurate, and because there are many more downhill steps per epoch with small mini-batches, stochastic gradient descent gets much closer to the minimum in a single epoch with small mini-batches than with large ones.

Execution time. This explains why after one epoch of the MNIST data, a mini-batch size of 1 achieves the highest accuracy and lowest cross entropy: because it does the most weight updates. However, observing the execution, one notices that although the accuracy is highest with a mini-batch size of 1, so is the execution time. This is because the execution time for one iteration of gradient descent is not directly proportional to mini-batch size. There are two components to each iteration: (1) forward and back propagation, and (2) updating the weights. Part (1) is proportional to mini-batch size, since it computes a complete set of hidden values and derivatives for each training sample. However, part (2) has a constant cost, independent of mini-batch size, since the set of weights has a fixed size.

When the mini-batch size is large, part (1) will dominate the cost of an iteration, so it will be roughly proportional to mini-batch size. In this case, the cost (execution time) of an epoch will be roughly proportional to the size of the training set, and independent of mini-batch size. i.e., the cost of an epoch will be equal to the total number of training samples that must be processed by forward and back propagation.

However, when the mini-batch size is very small, part (2) will dominate, so iterations will have constant cost. In this case, the cost (execution time) of an epoch will be proportianl to the number of weight updates, which goes up as mini-batch size goes down. This is why an epoch takes a long time to execute when the mini-batch size is 1: it does the largest number of weight updates per epoch.

To sumarize, on the MNIST data, a mini-batch size of 1 gets much closer to the minimum with each epoch than do larger mini-batches, and will therefore reach the minimum in the fewest number of epochs. However, it may still take a long time to reach the minimum, since the execution time per epoch is the largest for a mini-batch size of 1. For this reason, the actual time required to reach the minimum may be optimized by mini-batches that are small, but not the smallest possible.

Parallelism. When using a massively parallel machine, such as a gpu, we expect the optimal mini-batch size to be even larger. This is partly because gradient descent is an inherently sequential algorithm, i.e., it processes mini-batches sequentially, one after the other. Thus, the only way it can take advantage of parallelism is by processing more data per iteration, that is, by using larger mini-batches. In fact, up to a limit, the cost per iteration is now constant and does not increase as mini-batch size increases, since all the various training samples in a mini-batch can now be processed in parallel. e.g., The time required to do forward and back propagation for 100 training samples is the same as for one training sample. Thus, it makes sense to use larger mini-batches, since by doing so, the gradients will be more accurate. Convergence will therefore come after fewer iterations, but without any increase in cost (execution time) per iteration.

(f) *Batch Gradient Descent: implementation.* (15 points)

Use the theory developed in Question 4 to write a Python program that trains a neural net as specified in the introduction using batch gradient descent.

Initialize the weight matrices randomly using a standard Gaussian distribution (*i.e.*, mean 0 and variance 1). Initialize them bottom up: first W, then V, then U.[5] Initialize the bias terms to 0. Be sure to set the seed for random number generation to 0 at the start of your program. Your program may use one loop but no nested loops, and all code should be vectorized. Using program comments, clearly indicate what portion of your program implements the forward pass of training, what portion implements back propagation and what portion implements the weight updates.

When doing weight updates, use the average gradient, not the gradient itself. This means dividing the gradient, as given by the formulas in Question 4, by N, the number of terms in the sum in Equation (9). In batch gradient descent, N is the number of points in the training set, but would be the batch size in stochastic gradient descent. Using the average gradient means that the learning rate does not have to change much when the size of the training set changes. Notice that using the average gradient is equivalent to dividing the learning rate by N.

At the beginning of each iteration, before the weights and bias terms are updated, compute the accuracy of the neural net on the reduced test data, then print the iteration number and the test accuracy. (The number of the first iteration is 0.) When training has finished, compute and print the accuracy and cross entropy of the neural net on the reduced test data.

Run your program on the reduced training data using a learning rate of 0.1 and 10 iterations of gradient descent.

If everything is working correctly, then if you test your program by running it on the reduced data set of digits 4 and 5, the test accuracy printed at the end should be exactly 0.6905016008537886, and the test cross-entropy should be ex-

[5] Because initialization is random, the order affects the initial state of the neural net, and thus the final state, and thus the output of the trained net.

actly 1.5652003569260415 (or almost exactly, depending on your machine). Do not hand in these results. Instead, hand in results for the reduced data set of digits 5 and 6.)

(g) *Stochastic Gradient Descent: implementation.* (10 points)

Modify your program in part (f) to peform stochastic gradient descent with mini-batches. That is, instead of computing the gradient of the loss function on the entire training set at once, compute the gradient on a small, random subset of the training data (called a mini-batch), perform weight updates, and then move on to the next mini-batch, and so on. As you saw in part (d), this can lead to much faster convergence.

To produce random mini-batches, shuffle the training data randomly, then sweep across the shuffled data from start to finish. For example, if we want mini-batches of size 100, then the first mini-batch is the first 100 points in the training set. The second mini-batch is the second 100 points. The third mini-batch is the third 100 points, etc. (If the number of training points is not a multiple of 100 then the last mini-batch in a sweep will have fewer than 100 points in it.) Each such sweep of the training data is called an epoch. Program comments should clearly indicate where an epoch begins and where mini-batches are created.

During each epoch, your program should do the following. First, compute the accuracy of the neural net on the reduced test data, and print the epoch number and the accuracy. (The number of the first epoch is 0.) Second, use the following statement to randomly shuffle the training data: `X,T = sklearn.utils.shuffle(X,T)`. (Do not do any other shuffling of the data.) Third, sweep through the shuffled data, generating mini-batches and processing each of them in turn as described above. When updating weights, use the average gradient, averaged over the current mini-batch.

Unlike part (f), your program here may use two loops, one nested inside the other.

Like part (f), your program should also do the following. Set the seed for random number generation to 0 at the start of the program. Initialize the weight matrices randomly using a standard Gaussian distribution (*i.e.*, mean 0 and variance 1), and initialize the bias terms to 0. When training has finished, compute and print the accuracy and cross entropy of the neural net on the reduced test data. Use program comments to clearly indicate what portion of your program implements the forward pass of training, what portion implements back propagation, and what portion implements the weight updates. Use vectorized code.

As in part (f), run your program on the reduced training data using a learning rate of 0.1 and 10 epochs of gradient descent. In addition, use a mini-batch size of 10. If everything is working correctly, then if you test your program by running it on the reduced data set of digits 4 and 5, the test accuracy printed at the end should be above 0.95, and the test cross-entropy should be below 0.1. Do not hand in these results. Instead, hand in results for the reduced data set of digits 5 and 6.

Notice that the accuracy for stochastic gradient descent is much higher than for batch gradient descent, and the cross entropy (which we want to minimize) is much lower.

207 points total

```
###########  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 sklearn.linear_model as lin
import sklearn.utils as utils
import pickle
import sklearn.discriminant_analysis as da
import sklearn.naive_bayes as nb
import bonnerlib2D as bl2d
import sklearn.neural_network as nn
from scipy.stats import multivariate_normal
from sklearn.utils.testing import ignore_warnings



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

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


# read the training and testing data
with open('dataA2Q1.pickle','rb') as file:
    dataTrain,dataTest = pickle.load(file)


# plt.scatter(dataTrain[0],dataTrain[1],s=20)
# plt.suptitle('Training data for Question 1')


# Question 1(a).

# construct a feature matrix from the data in X
# using K trigonometric functions.
def feature_matrix(X,K):
    N = len(X)    # number of data points
    Klist = range(1,K+1)
    Klist = np.reshape(Klist,(1,K))
    X = np.reshape(X,(N,1))
    XK = X*Klist    # multiples of X.  shape = [N,K]
    sinXK = np.sin(XK)    # sine basis functions
    cosXK = np.cos(XK)    # cosine basis functions
    ones = np.ones((N,1))    # vector of ones
    Z = np.concatenate([ones,sinXK,cosXK],axis=1)    # feature matrix.  shape = [N,2K+1]
    return Z


# compute the mean squared-error of a function on a data set.
# T = target vector
# Z = feature matrix
def error(T,Z,w):
    Y = np.matmul(Z,w)    # predicted values
    return np.mean((T-Y)**2)


# Fit a function to the training data using K basis functions.
def myfit(dataTrain,dataTest,K):
    Xtrain,Ttrain = dataTrain
    Xtest,Ttest = dataTest
    # fit a function to the training data
    Ztrain = feature_matrix(Xtrain,K)
    w = la.lstsq(Ztrain,Ttrain,rcond=None)[0]
    errTrain = error(Ttrain,Ztrain,w)
    # compute test error
    Ztest = feature_matrix(Xtest,K)
    errTest = error(Ttest,Ztest,w)
    return w,errTrain,errTest
```

```
# plot the function defined by the weight vector w
def myplot(w,dataTrain,K):
    Xtrain,Ttrain = dataTrain
    xMin = np.min(Xtrain)
    xMax = np.max(Xtrain)
    Xplot = np.linspace(xMin,xMax,1000)    # 1000 equally spaced points
    Zplot = feature_matrix(Xplot,K)
    Yplot = np.matmul(Zplot,w)    # function values
    plt.scatter(Xtrain,Ttrain,c='b',s=20)    # plot the training data
    plt.plot(Xplot,Yplot,'r')    # plot the function
    yMin = np.min(Ttrain) - 5
    yMax = np.max(Ttrain) + 5
    plt.ylim(yMin,yMax)


# Fit a function to training data and plot the results.
def fit_plot(dataTrain,dataTest,K):
    w,errTrain,errTest = myfit(dataTrain,dataTest,K)
    myplot(w,dataTrain,K)
    return w,errTrain,errTest


# Question 1(b)
print('\nQuestion 1(b):')

def test_fit(dataTrain,dataTest,K):
    plt.figure()
    w,errTrain,errTest = fit_plot(dataTrain,dataTest,K)
    plt.xlabel('x')
    plt.ylabel('y')
    print('K =',K)
    print('Training error =', errTrain)
    print('Test error =', errTest)
    print('Weight vector =')
    print(w)

test_fit(dataTrain,dataTest,3)
plt.suptitle('Question 1(b): the fitted function (K=3)')



# Question 1(c)
print('\nQuestion 1(c):')
test_fit(dataTrain,dataTest,9)
plt.suptitle('Question 1(c): the fitted function (K=9)')



# Question 1(d)
print('\nQuestion 1(d):')
test_fit(dataTrain,dataTest,12)
plt.suptitle('Question 1(d): the fitted function (K=12)')



# Question 1(e)
plt.figure()
kmax = 12
for k in range(1,kmax+1):
    plt.subplot(4,3,k)
    fit_plot(dataTrain,dataTest,k)
plt.suptitle('Question 1(e): fitted functions for many values of K')



# Question 1(f)
print('\nQuestion 1(f):')
# Cross validation
kmax = 12    # maximum value of k
F = 5    # number of folds
S = 5    # size of each fold
errorsTrain = np.zeros([F,kmax+1])    # space for training errors
```

```
errorsVal = np.zeros([F,kmax+1])    # space for validation errors
for f in range(F):
    # use fold f as validation data
    j1 = f*S    # start of fold f
    j2 = j1+S    # end of fold f
    dataVal = dataTrain[:,j1:j2]    # validation data
    # use the remaining folds as training data
    dataTrain1 = dataTrain[:,:j1]
    dataTrain2 = dataTrain[:,j2:]
    dataTrain3 = np.concatenate([dataTrain1,dataTrain2],axis=1)    # training data
    # compute training and validation error for each value of f and k
    for k in range(kmax+1):
        w,errTrain,errVal = myfit(dataTrain3,dataVal,k)
        errorsTrain[f,k] = errTrain
        errorsVal[f,k] = errVal

# compute mean error for each value of k
errorsTrain = np.mean(errorsTrain,axis=0)
errorsVal = np.mean(errorsVal,axis=0)

# plot mean training and validation errors
plt.figure()
plt.semilogy(errorsTrain,'b')    # plot training errors
plt.semilogy(errorsVal,'r')    # plot validation errors
plt.xlabel('K')
plt.ylabel('Mean error')
plt.suptitle('Question 1(f): mean training and validation error')

# train and test the final classifier
kBest = np.argmin(errorsVal)
print('for the best fit:')
test_fit(dataTrain,dataTest,kBest)
plt.suptitle('Question 1(f): the best fitting function')




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

######  Question 2  ######

# Question 2
print('\n')
print('Question 2.')
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

# plt.figure()
# plot_data(Xtrain,Ttrain)
# plt.title('Training data for Question 2')


# Question 2(a)
# display cluster data
def plot_data(X,T):
    colors = np.array(['r','b','g'])    # red for class 0 , blue for class 1, green for class 2
    plt.scatter(X[:, 0], X[:, 1], color=colors[T],s=2)
    xmin,ymin = np.min(X,axis=0) - 0.1
    xmax,ymax = np.max(X,axis=0) + 0.1
    plt.xlim(xmin,xmax)
    plt.ylim(ymin,ymax)



# Question 2(b)
```

```
print('')
print('Question 2(b):')

# compute accuracy of classifier clf on data X,T.
# use the weight matrix and bias vector to make predictions.
def accuracyLR(clf,X,T):
    W = clf.coef_    # weight matrix
    w0 = clf.intercept_    # bias vector
    Z = np.matmul(X,W.T) + w0    # values of the linear function for each training point
    Predictions = np.argmax(Z,axis=1)    # predictions for each training point
    return np.mean(Predictions==T)    # average number of correct predictions


# fit a logistic-regression classifier to data and display the results
clf = lin.LogisticRegression(multi_class='multinomial',solver='lbfgs')
clf.fit(Xtrain,Ttrain)
# Compute test accuracy in two different ways.
accuracy1 = clf.score(Xtest,Ttest)
accuracy2 = accuracyLR(clf,Xtest,Ttest)
# print and plot results
print('accuracy1 =', accuracy1)
print('accuracy2 =', accuracy2)
print('accuracy1 - accuracy2 =', accuracy1-accuracy2)
plt.figure()
plot_data(Xtrain,Ttrain)
bl2d.boundaries(clf)
plt.title('Question 2(b): decision boundaries for logistic regression')



# Question 2(c)
print('')
print('Question 2(c):')

# compute accuracy of classifier clf on data X,T.
# use the class models and Bayes rule to make predictions.
def accuracyQDA(clf,X,T):
    N,M = np.shape(X)    # number of test points
    C = len(clf.classes_)    # number of classes
    cov = clf.covariance_    # estimated covariance matrices for each class.  shape = [C,M,M]
    mean = clf.means_    # estimated mean vectors for each class.  shape = [C,M]
    Prior = clf.priors_    # estimated prior probabilities for each class.  shape = [C]
    # Compute class conditional probabilities
    Pclass = np.zeros([C,N])    # reserve space for the probabilities
    for c in range(C):
        Pclass[c] = multivariate_normal.pdf(X,mean[c],cov[c])
    # Use Bayes rule to compute posteriors
    Prior = np.reshape(Prior,[C,1])
    Posterior = Pclass*Prior    # shape = [C,N]
    # make predictions and compute accuracy
    Prediction = np.argmax(Posterior,axis=0)
    return np.mean(Prediction==T)    # average number of correct predictions


# Fit a QDA classifier to data and display the results
clf = da.QuadraticDiscriminantAnalysis(store_covariance=True)
clf.fit(Xtrain,Ttrain)
# compute test accuracy in two different ways
accuracy1 = clf.score(Xtest,Ttest)
accuracy2 = accuracyQDA(clf,Xtest,Ttest)
# print and plot results
print('Accuracy1 =', accuracy1)
print('Accuracy2 =', accuracy2)
print('Accuracy1 - Accuracy2 = ', accuracy1-accuracy2)
plt.figure()
plot_data(Xtrain,Ttrain)
bl2d.boundaries(clf)
plt.suptitle('Question 2(c): decision boundaries for quadratic discriminant analysis')



# Question 2(d)
print('')
```

```
print('Question 2(d):')

# compute accuracy of classifier clf on data X,T.
# use the class models and Bayes rule to make predictions.
def accuracyNB(clf,X,T):
    N,M = np.shape(X)    # data shape
    C = len(clf.classes_)    # number of classes
    var = clf.sigma_    # estimated variance for each class and feature.  shape = [C,M]
    mean = clf.theta_    # estimated mean for each class and feature.  shape = [C,M]
    Prior = clf.class_prior_    # estimated prior probabilities for each class.  shape = [C]
    # Compute class conditional probabilities
    X = np.reshape(X,[1,N,M])
    mean = np.reshape(mean,[C,1,M])
    var = np.reshape(var,[C,1,M])
    P = np.exp(-(X-mean)**2/(2*var))/np.sqrt(2*np.pi*var)    # shape = [C,N,M]
    Pclass = np.prod(P,axis=2)   # shape = [C,N]
    # Use Bayes rule to compute posteriors
    Prior = np.reshape(Prior,[C,1])
    Posterior = Pclass*Prior    # shape = [C,N]
    # make predictions and compute accuracy
    Prediction = np.argmax(Posterior,axis=0)
    return np.mean(Prediction==T)    # average number of correct predictions


# fit a Gaussian naive Bayes classifier to data and display the results
clf = nb.GaussianNB()
clf.fit(Xtrain,Ttrain)
# compute test accuracy in two different ways
accuracy1 = clf.score(Xtest,Ttest)
accuracy2 = accuracyNB(clf,Xtest,Ttest)
# print and plot results
print('Accuracy1 =', accuracy1)
print('Accuracy2 =', accuracy2)
print('Accuracy1 - Accuracy2 = ', accuracy1-accuracy2)
plt.figure()
plot_data(Xtrain,Ttrain)
bl2d.boundaries(clf)
plt.suptitle('Question 2(d): decision boundary for Naive Bayes')




######  Question 3  ######

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

# Question 3(a)
# 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 3(b)
print('\nQuestion 3(b):')
# Fit a neural net with K hidden units to the training data.
# Use I iterations of stochastic gradient descent.
def myfitNN(K,I,seed=0):
    rnd.seed(seed)
    clf = nn.MLPClassifier(hidden_layer_sizes=[K],
                           activation='logistic',
                           max_iter=I,
                           solver='sgd',
                           tol=10.0**(-6),
                           learning_rate_init=0.01)
    ignore_warnings(clf.fit)(Xtrain,Ttrain)    # train the neural net
    # compute and print the test accuracy
    accuracy = clf.score(Xtest,Ttest)
```

```
    print('Accuracy =', accuracy)
    # plot the test data and decision boundaries
    plot_data(Xtrain,Ttrain)
    bl2d.boundaries(clf)

plt.figure()
myfitNN(1,1000)
plt.suptitle('Question 3(b): Neural net with 1 hidden unit')


# Question 3(c)
print('\nQuestion 3(c):')
plt.figure()
myfitNN(2,1000)
plt.suptitle('Question 3(c): Neural net with 2 hidden units')


# Question 3(d)
print('\nQuestion 3(d):')
plt.figure()
myfitNN(9,1000)
plt.suptitle('Question 3(d): Neural net with 9 hidden units')


# Question 3(e)
# different numbers of iterations
print('\nQuestion 3(e):')
plt.figure()
for logI in range(1,10):
    plt.subplot(3,3,logI)
    myfitNN(7,2**(logI+1))
plt.suptitle('Question 3(e): different numbers of epochs')


# Question 3(f)
# different initial weights
print('\nQuestion 3(f):')
plt.figure()
for i in range(1,10):
    plt.subplot(3,3,i)
    myfitNN(5,1000,i)
plt.suptitle('Question 3(f): different initial weights')


# Question 3(g)
print('\nQuestion 3(g):')

# compute the accuracy of a neural-net classifier on a data set in two ways.
# clf is the classifier
# X,T is the data set
def accuracyNN(clf,X,T):
    W1,W2 = clf.coefs_
    b1,b2 = clf.intercepts_
    Z1 = np.matmul(X,W1) + b1
    H1 = 1/(1+np.exp(-Z1))
    Z2 = np.matmul(H1,W2) + b2
    Prediction = np.argmax(Z2,axis=1)
    return np.mean(Prediction==T)

rnd.seed(0)
clf = nn.MLPClassifier(hidden_layer_sizes=[9],
                    activation='logistic',
                    max_iter=1000,
                    solver='sgd',
                    tol=10.0**(-6),
                    learning_rate_init=0.01)
ignore_warnings(clf.fit)(Xtrain,Ttrain)    # train the neural net
# compute test accuracy in two different ways
accuracy1 = clf.score(Xtest,Ttest)
accuracy2 = accuracyNN(clf,Xtest,Ttest)
# print and plot results
print('Accuracy1 =', accuracy1)
print('Accuracy2 =', accuracy2)
```

```
print('Accuracy1 - Accuracy2 = ', accuracy1-accuracy2)


# Question 3(h)
print('\nQuestion 3(h):')

# compute the log of the softmax of Z
def log_softmax(Z):
    expZ = np.exp(Z)
    logsumexpZ = np.log(np.sum(expZ,axis=1,keepdims=True))
    return Z - logsumexpZ


# compute the cross entropy of a neural-net classifier on a data set in two ways.
# clf is the classifier
# X,T is the data set
def ceNN(clf,X,T):
    # compute a one-hot encoding of T
    N = len(T)
    C = clf.n_outputs_    # number of output classes
    cList = range(C)
    cList = np.reshape(cList,[1,C])
    T = np.reshape(T,[N,1])
    onehot = (T==cList)    # shape = [N,C]
    # compute CE1
    logP1 = clf.predict_log_proba(X)    # shape = [N,C]
    CE1 = -np.sum(logP1*onehot)/N
    # forward propagation
    W1,W2 = clf.coefs_
    b1,b2 = clf.intercepts_
    Z1 = np.matmul(X,W1) + b1
    H1 = 1/(1+np.exp(-Z1))
    Z2 = np.matmul(H1,W2) + b2
    # compute CE2
    logP2 = log_softmax(Z2)
    CE2 = -np.sum(logP2*onehot)/N
    return CE1, CE2


rnd.seed(0)
clf = nn.MLPClassifier(hidden_layer_sizes=[9],
                       activation='logistic',
                       max_iter=1000,
                       solver='sgd',
                       tol=10.0**(-6),
                       learning_rate_init=0.01)
ignore_warnings(clf.fit)(Xtrain,Ttrain)    # train the neural net
# compute and print cross entropies
CE1,CE2 = ceNN(clf,Xtest,Ttest)
print('CE1 =', CE1)
print('CE2 =', CE2)
print('CE1-CE2 = ', CE1 - CE2)




######  Question 5  ######

print('\n')
print('Question 5.')
print('-----------')

# 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)


# construct reduced data consisting of digits d1 and d2.
# all data sets are global variables.
def reduce(d1,d2):
```

```
    global Xtrain2, Ttrain2, Xtest2, Ttest2
    # reduced training data
    idx = (Ttrain==d1) | (Ttrain==d2)    # index to digits d1 and d2
    Xtrain2 = Xtrain[idx]
    Ttrain2 = Ttrain[idx]
    # reduced test data
    idx = (Ttest==d1) | (Ttest==d2)
    Xtest2 = Xtest[idx]
    Ttest2 = Ttest[idx]
    # binarize the labels
    Ttrain2 = (Ttrain2==d1)+0
    Ttest2 = (Ttest2==d1)+0
    # convert to column vectors
    Ttrain2 = np.reshape(Ttrain2,[-1,1])
    Ttest2 = np.reshape(Ttest2,[-1,1])

reduce(5,6)



# Question 5(b)

# mean cross entropy of P and T.
# P = probabilities.  T = target values.
def cross_entropy(P,T):
    # compute cross entropy
    CE = -T*np.log(P) - (1-T)*np.log(1-P)   # cross entropies
    return np.mean(CE)   # mean cross entropy

# accuracy of predictions
# given probabilities P and targets T.
def accuracy(P,T):
    # compute accuracy
    Predictions = (P > 0.5)    # predictions
    correct = (Predictions==T)    # correct predictions
    return np.mean(correct)

def sigmoid(Z):
    return 1/(1+np.exp(-Z))

# do a forward pass through a neural net from input to output.
# return the output and hidden values.
# shape(X) = [N,I]
# shape(W) = [I,J]
# shape(V) = [J,K]
# shape(U) = [K,1]
def forward(X,U,V,W,u0,v0,w0):
    Xt = np.matmul(X,W)+w0    # shape = [N,J]
    H = np.tanh(Xt)    # shape = [N,J]
    Ht = np.matmul(H,V)+v0    # shape = [N,K]
    G = np.tanh(Ht)    # shape = [N,K]
    Gt = np.matmul(G,U) + u0    # shape = [N,1]
    O = sigmoid(Gt)    # shape = [N,1]
    return O,G,H


# compute the accuracy and cross entropy of neural-net clf on data X,T
def evaluateNN(clf,X,T):
    # compute accuracy1 and CE1
    accuracy1 = clf.score(X,T)
    P = clf.predict_proba(X)
    P = P[:,1]    # probability of class 1
    P = np.reshape(P,[-1,1])    # convert to a column vector
    CE1 = cross_entropy(P,T)
    # compute accuracy2 and CE2
    W,V,U = clf.coefs_
    w0,v0,u0 = clf.intercepts_
    P = forward(X,U,V,W,u0,v0,w0)[0]
    accuracy2 = accuracy(P,T)
    CE2 = cross_entropy(P,T)
    return accuracy1,accuracy2,CE1,CE2
```

```
# Question 5(c).
# stochastic gradient descent
print('\nQuestion 5(c):')
rnd.seed(0)
N = 100    # batch size
clf = nn.MLPClassifier(hidden_layer_sizes=[100,100],
                    activation='tanh',
                    max_iter=100,
                    solver='sgd',
                    tol=10.0**(-6),
                    learning_rate_init=0.01,
                    batch_size=N)
ignore_warnings(clf.fit)(Xtrain2,Ttrain2)    # train the neural net
# compute and print cross entropies
accuracy1,accuracy2,CE1,CE2 = evaluateNN(clf,Xtest2,Ttest2)
print('accuracy1 =', accuracy1)
print('accuracy2 =', accuracy2)
print('accuracy1 - accuracy2 =', accuracy1-accuracy2)
print('CE1 =', CE1)
print('CE2 =', CE2)
print('CE1-CE2 = ', CE1-CE2)



# Question 5(d)
# effects of mini-batch size
accList = []
ceList = []
logN = np.array(range(14))
Nlist = 2**logN    # batch sizes
for N in Nlist:
    rnd.seed(0)
    clf = nn.MLPClassifier(hidden_layer_sizes=[100,100],
                        activation='tanh',
                        max_iter=1,
                        solver='sgd',
                        tol=10.0**(-6),
                        learning_rate_init=0.001,
                        batch_size=N)
    ignore_warnings(clf.fit)(Xtrain2,Ttrain2)    # train the neural net
    # compute and print cross entropies
    accuracy1,accuracy2,CE1,CE2 = evaluateNN(clf,Xtest2,Ttest2)
    accList.append(accuracy1)
    ceList.append(CE1)

plt.figure()
plt.semilogx(Nlist,accList)
plt.title('Question 5(d): Accuracy v.s. batch size')
plt.xlabel('batch size')
plt.ylabel('accuracy')

plt.figure()
plt.semilogx(Nlist,ceList)
plt.title('Question 5(d): Cross entropy v.s. batch size')
plt.xlabel('batch size')
plt.ylabel('cross entropy')




# Question 5(f)
print('\nQuestion 5(f):')

# do N epochs of batch gradient descent for a neural net
# with J neurons in hidden layer 1
#  and K neurons in hidden layer 2.
# lrate is the learning rate.
# return the weights and bias terms
def batch_gd(J,K,N,lrate):
    rnd.seed(0)
    Ntrain,I = np.shape(Xtrain2)    # input shape
```

```
    # initialise the weights and biases
    W = rnd.randn(I,J)    # random initial weights for hidden layer 1
    V = rnd.randn(J,K)    # random initial weights for hidden layer 2
    U = rnd.randn(K,1)    # random initial weights for output layer
    w0 = np.zeros([1,J])    # initial bias term for hidden layer 1
    v0 = np.zeros([1,K])    # initial bias terms for hidden layer 2
    u0 = 0    # initial bias term for output layer

    X= Xtrain2
    T = Ttrain2
    lrate = lrate/float(Ntrain)
    for n in range(N):
        # compute and print test accuracy
        O = forward(Xtest2,U,V,W,u0,v0,w0)[0]
        accTest = accuracy(O,Ttest2)
        print('Epoch', n,'     Test accuracy =',accTest)

        # forward pass using training data (compute hidden values and output)
        O,G,H = forward(X,U,V,W,u0,v0,w0)

        # backward pass using training data (compute gradients of C)
        # output layer
        gGt = O - T    # gradient wrt Gt
        gU = np.matmul(G.T,gGt)    # gradient wrt U
        gu0 = np.sum(gGt)    # gradient wrt u0
        # hidden layer 2
        gG = np.matmul(gGt,U.T)    # gradient wrt G
        gHt = (1-G**2)*gG    # gradient wrt Ht
        gV = np.matmul(H.T,gHt)    # gradient wrt V
        gv0 = np.sum(gHt,axis=0)    # gradient wrt v0
        # hidden layer 1
        gH = np.matmul(gHt,V.T)    # gradient wrt H
        gXt = (1-H**2)*gH    # gradient wrt Xt
        gW = np.matmul(X.T,gXt)    # gradient wrt W
        gw0 = np.sum(gXt,axis=0)    # gradient wrt w0

        # update weight matrices
        U -= lrate*gU
        V -= lrate*gV
        W -= lrate*gW
        # update bias vectors
        u0 -= lrate*gu0
        v0 -= lrate*gv0
        w0 -= lrate*gw0

    # compute and print final accuracy and cross entropy
    O = forward(Xtest2,U,V,W,u0,v0,w0)[0]
    print('Test accuracy =',accuracy(O,Ttest2))
    print('Cross entropy =',cross_entropy(O,Ttest2))


# execute batch gradient descent
J = 100    # number of hidden units in layer 1
K = 100    # number of hidden units in layer 2
epochs = 10    # number of epochs of training
lrate = 0.1    # learning rate
batch_gd(J,K,epochs,lrate)




# Question 5(g)
print('\nQuestion 5(g):')

# do N epochs of stochastic gradient descent for a neural net
# with J neurons in hidden layer 1
#  and K neurons in hidden layer 2.
# lrate is the learning rate.
# return the weights and bias terms
def stochastic_gd(J,K,N,lrate,batch_size):
    rnd.seed(0)
    Ntrain,I = np.shape(Xtrain2)    # input shape
    # initialise the weights and biases
```

```
    W = rnd.randn(I,J)    # random initial weights for hidden layer 1
    V = rnd.randn(J,K)    # random initial weights for hidden layer 2
    U = rnd.randn(K,1)    # random initial weights for output layer
    w0 = np.zeros([1,J])    # initial bias term for hidden layer 1
    v0 = np.zeros([1,K])    # initial bias terms for hidden layer 2
    u0 = 0    # initial bias term for output layer

    Xtrain = Xtrain2
    Ttrain = Ttrain2
    lrate = lrate/batch_size
    # perform N epochs of stochastic gradient descent
    for n in range(N):
        # compute and print test accuracy
        Otest = forward(Xtest2,U,V,W,u0,v0,w0)[0]
        accTest = accuracy(Otest,Ttest2)
        print('Epoch', n,'     Test accuracy =',accTest)

        # copy and shuffle the training data randomly
        Xtrain,Ttrain = utils.shuffle(Xtrain,Ttrain)
        # perform one epoch of stochastic gradient descent
        ptr1 = 0    # pointer to start of mini-batch
        while ptr1 < Ntrain:
            # process one mini-batch of data.

            # get the next mini-batch
            ptr2 = np.min([ptr1+batch_size,Ntrain])    # pointer to end of mini-batch
            X = Xtrain[ptr1:ptr2]
            T = Ttrain[ptr1:ptr2]
            ptr1 = ptr2

            # forward pass (compute hidden values and output)
            O,G,H = forward(X,U,V,W,u0,v0,w0)

            # backward pass (compute gradients of C)
            # output layer
            gGt = O - T    # gradient wrt Gt
            gU = np.matmul(G.T,gGt)    # gradient wrt U
            gu0 = np.sum(gGt)    # gradient wrt u0
            # hidden layer 2
            gG = np.matmul(gGt,U.T)    # gradient wrt G
            gHt = (1-G**2)*gG    # gradient wrt Ht
            gV = np.matmul(H.T,gHt)    # gradient wrt V
            gv0 = np.sum(gHt,axis=0)    # gradient wrt v0
            # hidden layer 1
            gH = np.matmul(gHt,V.T)    # gradient wrt H
            gXt = (1-H**2)*gH    # gradient wrt Xt
            gW = np.matmul(X.T,gXt)    # gradient wrt W
            gw0 = np.sum(gXt,axis=0)    # gradient wrt w0

            # update weight matrices
            U -= lrate*gU
            V -= lrate*gV
            W -= lrate*gW
            # update bias vectors
            u0 -= lrate*gu0
            v0 -= lrate*gv0
            w0 -= lrate*gw0

    # compute and print final accuracy and cross entropy
    O = forward(Xtest2,U,V,W,u0,v0,w0)[0]
    print('Test accuracy =',accuracy(O,Ttest2))
    print('Cross entropy =',cross_entropy(O,Ttest2))


# execute stochastic gradient descent
J = 100    # number of hidden units in layer 1
K = 100    # number of hidden units in layer 2
epochs = 10    # number of epochs of training
lrate = 0.1    # learning rate
batch_size = 10
stochastic_gd(J,K,epochs,lrate,batch_size)
```

PRINTED OUTPUT

QUESTION 1.

Question 1(b):
K = 3
Training error = 8.658268242282475
Test error = 11.905786549444745
Weight vector = [114.30512878 -187.717837 31.24228605 14.25033792
-23.28707996
-96.93067091 8.90033976]

Question 1(c):
K = 9
Training error = 3.1039124268428613
Test error = 86.30061324917065
Weight vector =
[4.19153582e+06 -7.78801461e+06 -6.11451061e+04 4.26697403e+06
5.19610952e+04 -1.18585189e+06 -1.64583891e+04 1.33819261e+05
1.61411573e+03 -2.77362459e+03 3.75946168e+04 -6.23166541e+06
-6.45948312e+04 2.47048666e+06 3.29936800e+04 -4.57000348e+05
-6.21109464e+03 2.67547714e+04 2.20973775e+02]

Question 1(d):
K = 12
Training error = 3.868623271896202e-06
Test error = 72421239311.001
Weight vector =
[-1.38272028e+12 2.60469320e+12 1.29933907e+11 -1.60349497e+12
-1.28998833e+11 5.86984324e+11 5.67945136e+10 -1.17534133e+11
-1.17050514e+10 1.07734907e+10 9.17753346e+08 -2.89113662e+08
-1.20176015e+07 -7.71131185e+10 2.17421044e+12 1.46000988e+11
-1.03921593e+12 -9.37831785e+10 2.85504945e+11 2.85405566e+10
-3.99309947e+10 -3.79413395e+09 2.16750778e+09 1.48118948e+08
-1.91312306e+07]

Question 1(f):
for the best fit:
K = 2
Training error = 13.741160276011058
Test error = 12.090256909518182
Weight vector =
[66.22002474 -107.19437051 15.21342666 -9.79486139 -52.43545557]

Question 2.

Question 2(b):
accuracy1 = 0.7766666666666666
accuracy2 = 0.7766666666666666
accuracy1 - accuracy2 = 0.0

Question 2(c):
Accuracy1 = 0.8227777777777778
Accuracy2 = 0.8227777777777778
Accuracy1 - Accuracy2 = 0.0

Question 2(d):
Accuracy1 = 0.7838888888888889
Accuracy2 = 0.7838888888888889
Accuracy1 - Accuracy2 = 0.0

Question 3.

Question 3(b):
Accuracy = 0.6861111111111111

Question 3(c):
Accuracy = 0.7844444444444445

Question 3(d):
Accuracy = 0.8111111111111111

Question 3(e):
Accuracy = 0.5555555555555556
Accuracy = 0.6277777777777778
Accuracy = 0.7044444444444444
Accuracy = 0.7205555555555555
Accuracy = 0.7927777777777778
Accuracy = 0.79
Accuracy = 0.7927777777777778
Accuracy = 0.8027777777777778
Accuracy = 0.8122222222222222

Question 3(f):
Accuracy = 0.8105555555555556
Accuracy = 0.81

Accuracy = 0.8144444444444444
Accuracy = 0.8122222222222222
Accuracy = 0.8116666666666666
Accuracy = 0.8116666666666666
Accuracy = 0.8116666666666666
Accuracy = 0.8116666666666666
Accuracy = 0.81

Question 3(g):
Accuracy1 = 0.8111111111111111
Accuracy2 = 0.8111111111111111
Accuracy1 - Accuracy2 = 0.0

Question 3(h):
CE1 = 0.4318194397858982
CE2 = 0.4318194397858982
CE1-CE2 = 0.0

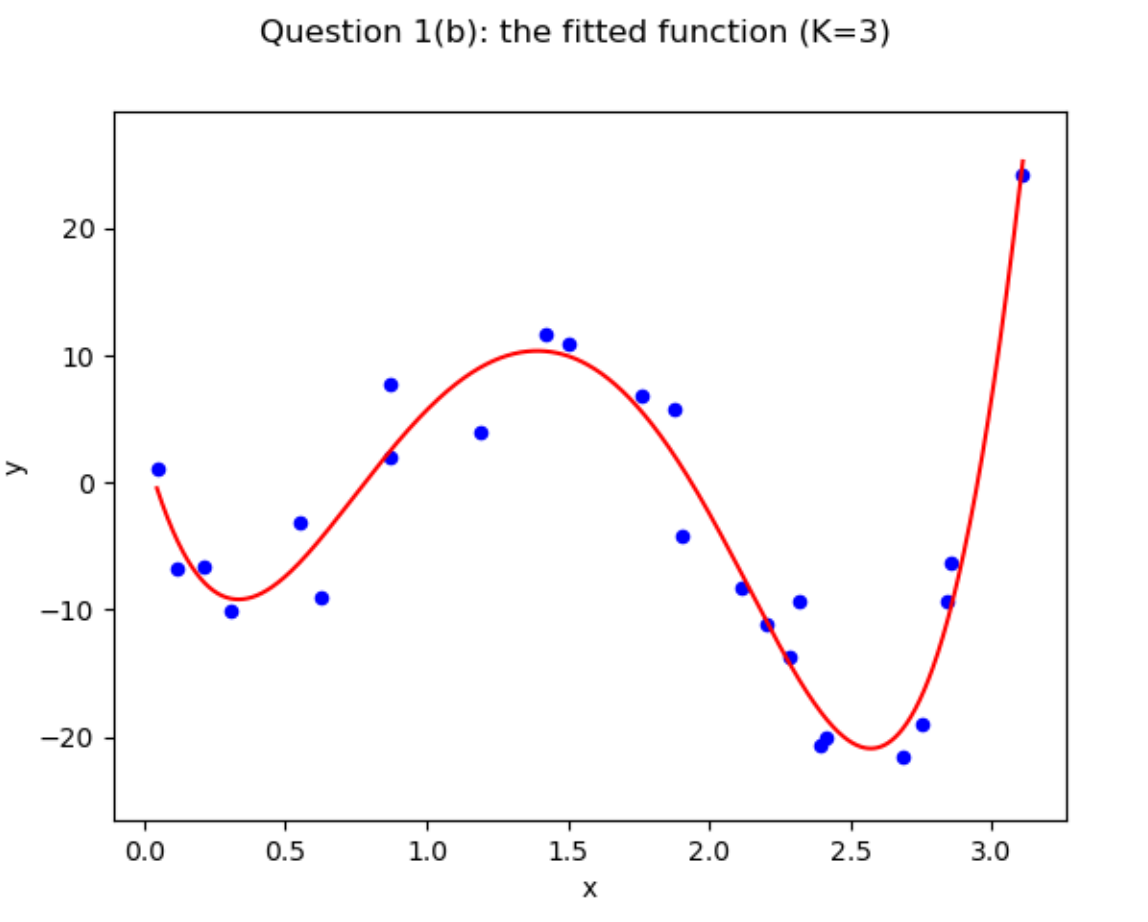
Question 5.

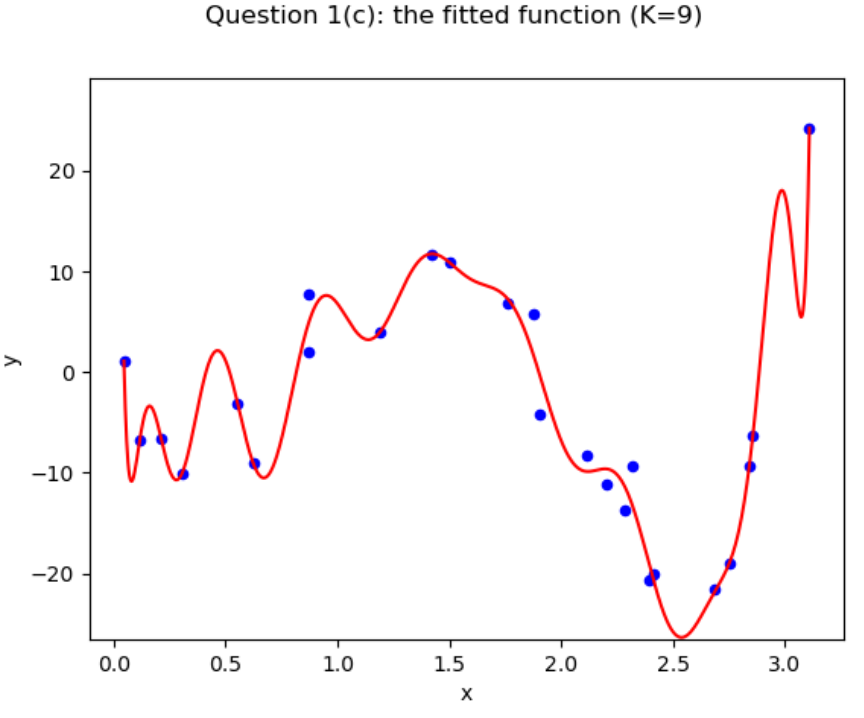
Question 5(c):
accuracy1 = 0.9913513513513513
accuracy2 = 0.9913513513513513
accuracy1 - accuracy2 = 0.0
CE1 = 0.04597586349442823
CE2 = 0.04597586349442823
CE1-CE2 = 0.0

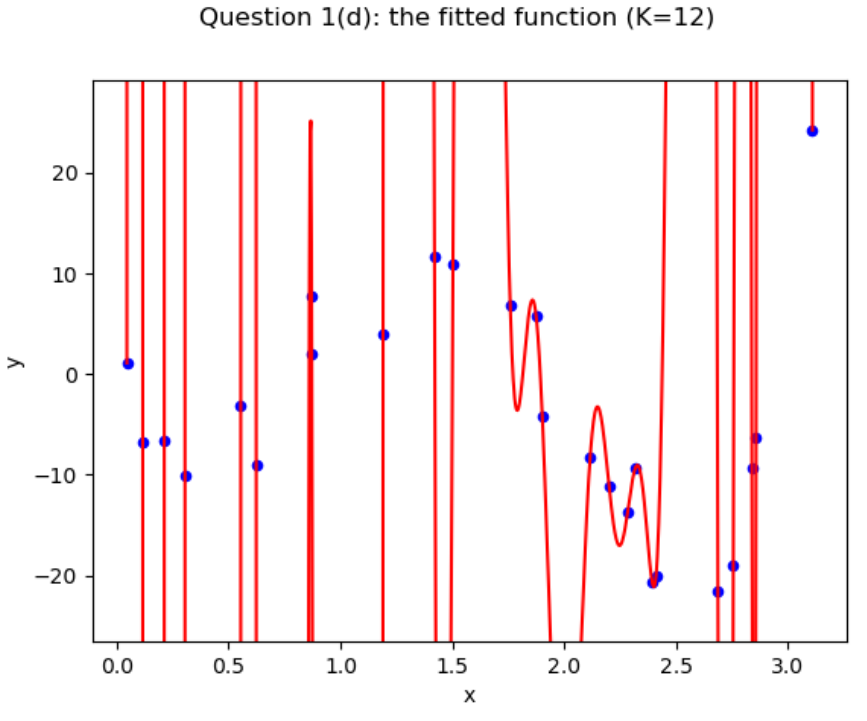
Question 5(f):
Epoch 0 	Test accuracy = 0.6302702702702703
Epoch 1 	Test accuracy = 0.6491891891891892
Epoch 2 	Test accuracy = 0.6616216216216216
Epoch 3 	Test accuracy = 0.6767567567567567
Epoch 4 	Test accuracy = 0.6897297297297297
Epoch 5 	Test accuracy = 0.7037837837837838
Epoch 6 	Test accuracy = 0.7113513513513513
Epoch 7 	Test accuracy = 0.72
Epoch 8 	Test accuracy = 0.7297297297297297
Epoch 9 	Test accuracy = 0.7367567567567568
Test accuracy = 0.7427027027027027
Cross entropy = 1.1803255469619718

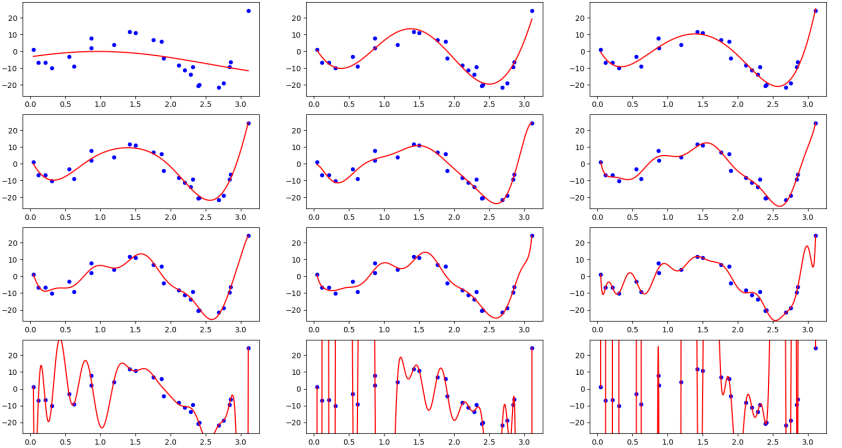
Question 5(g):
Epoch 0 	Test accuracy = 0.6302702702702703
Epoch 1 	Test accuracy = 0.9513513513513514

```
Epoch 2      Test accuracy = 0.9572972972972973
Epoch 3      Test accuracy = 0.9664864864864865
Epoch 4      Test accuracy = 0.9702702702702702
Epoch 5      Test accuracy = 0.9691891891891892
Epoch 6      Test accuracy = 0.9637837837837838
Epoch 7      Test accuracy = 0.9697297297297297
Epoch 8      Test accuracy = 0.972972972972973
Epoch 9      Test accuracy = 0.9740540540540541
Test accuracy = 0.9751351351351352
Cross entropy = 0.11300144660452133
```

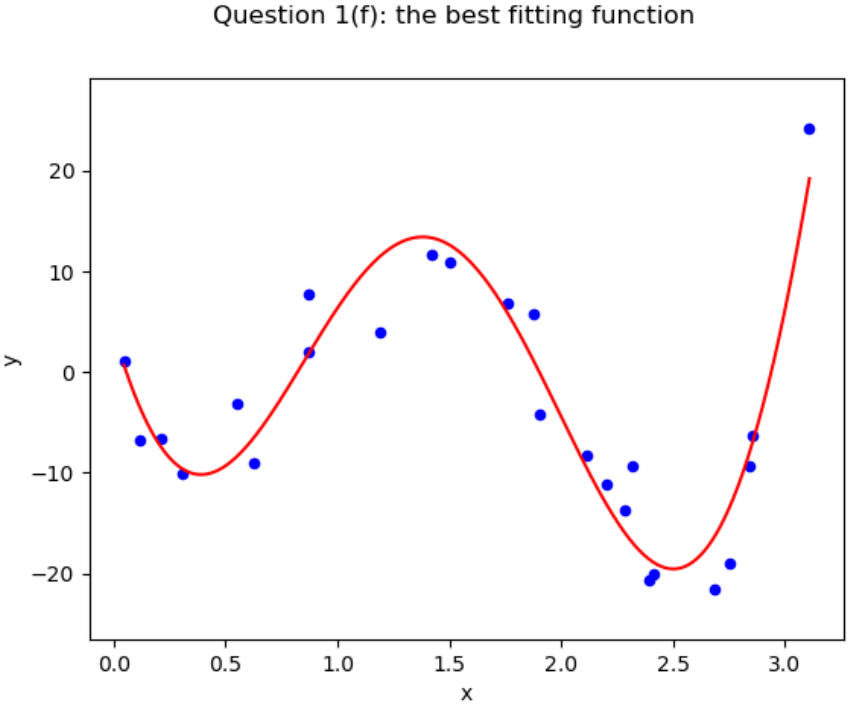

Question 1(b): the fitted function (K=3)
20
10
0
−10
−20
y
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x

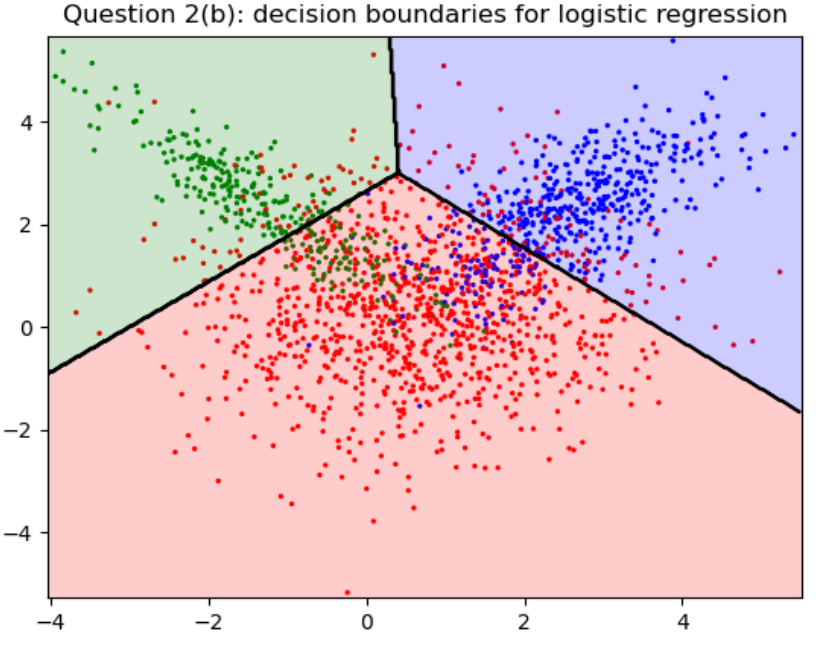

Question 1(c): the fitted function (K=9)
20
10
0
−10
−20
y
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x


Question 1(d): the fitted function (K=12)
20
10
0
−10
−20
y
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x

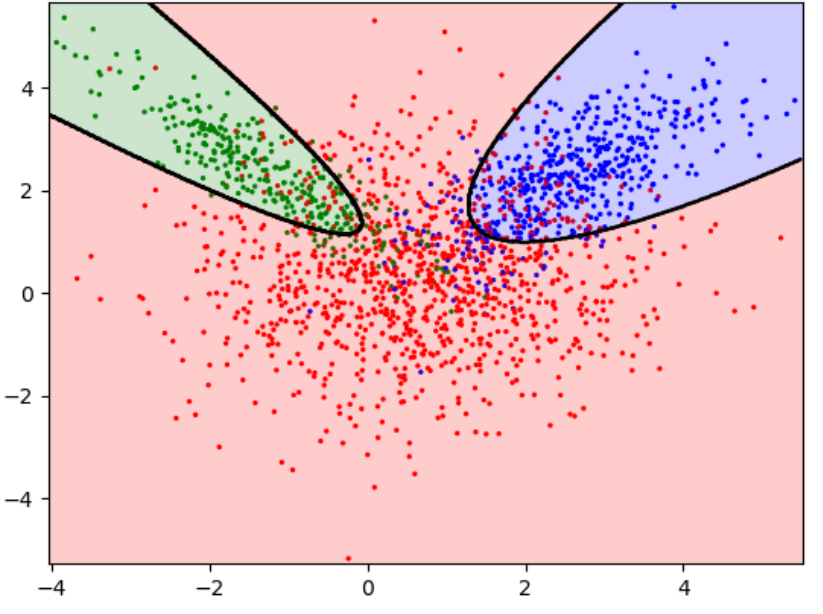

Question 1(e): fitted functions for many values of K

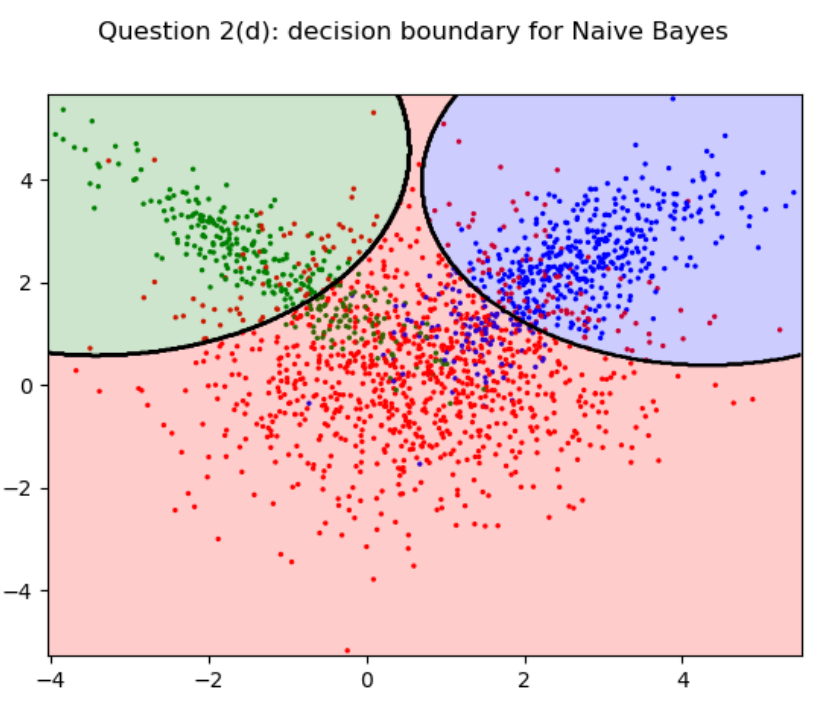

Question 1(f): mean training and validation error
Mean error
10^{15}
10^{11}
10^{7}
10^{3}
10^{-1}
10^{-5}
10^{-9}
10^{-13}
0
2
4
6
8
10
12
K

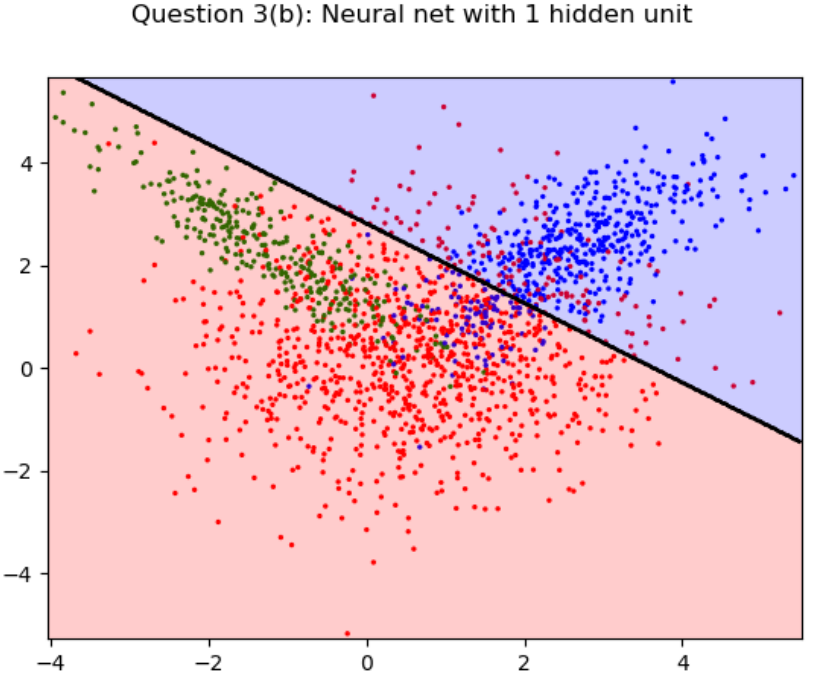

Question 1(f): the best fitting function
20
10
0
−10
−20
y
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x

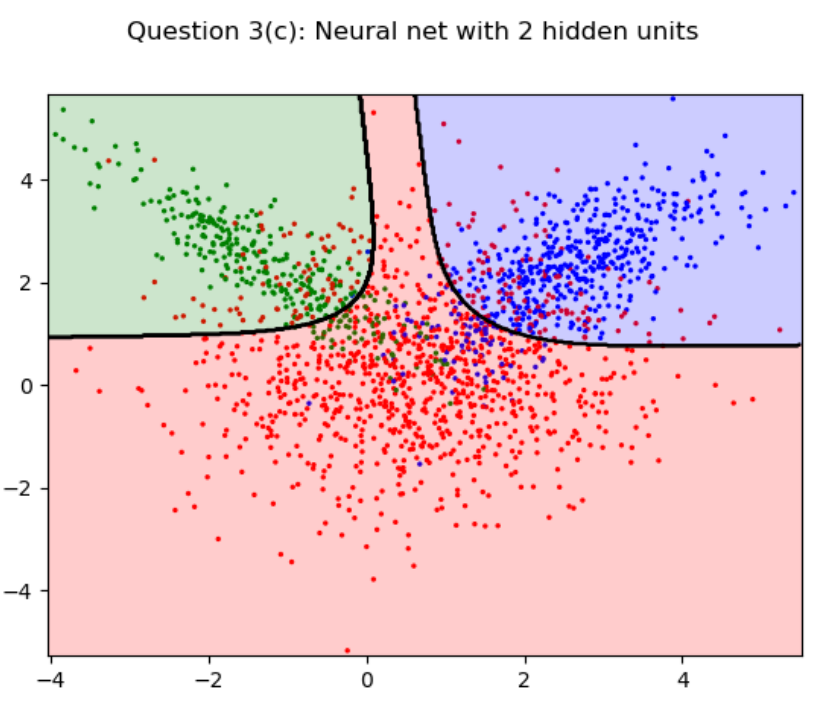

Question 2(b): decision boundaries for logistic regression
4
2
0
−2
−4
−4
−2
0
2
4

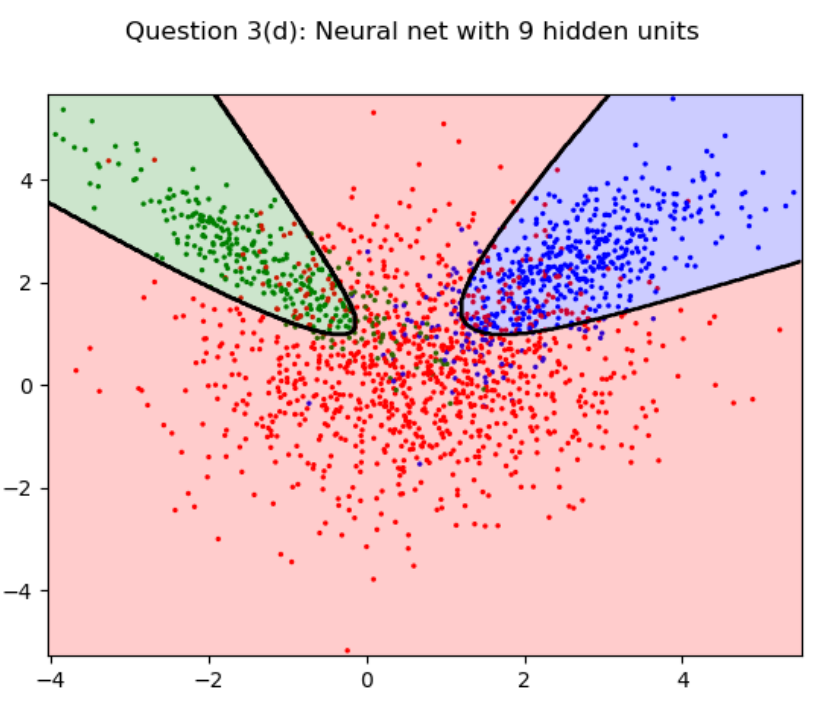
Question 2(c): decision boundaries for quadratic discriminant analysis



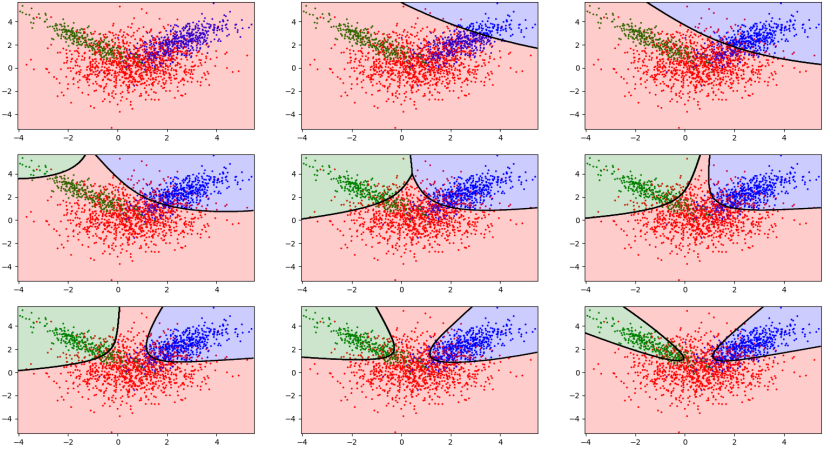

Question 2(d): decision boundary for Naive Bayes
4
2
0
−2
−4
−4
−2
0
2
4


Question 3(b): Neural net with 1 hidden unit
4
2
0
−2
−4
−4
−2
0
2
4

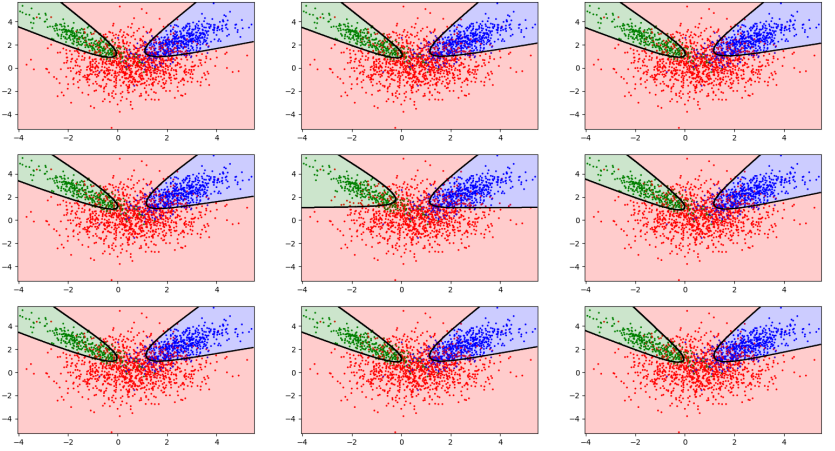

Question 3(c): Neural net with 2 hidden units
4
2
0
−2
−4
−4
−2
0
2
4

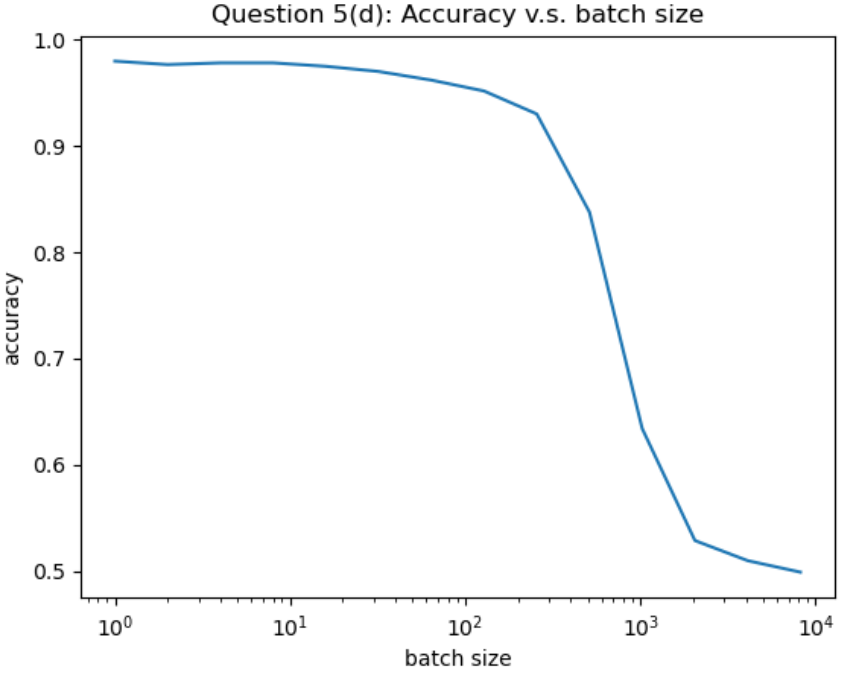

Question 3(d): Neural net with 9 hidden units
4
2
0
−2
−4
−4
−2
0
2
4

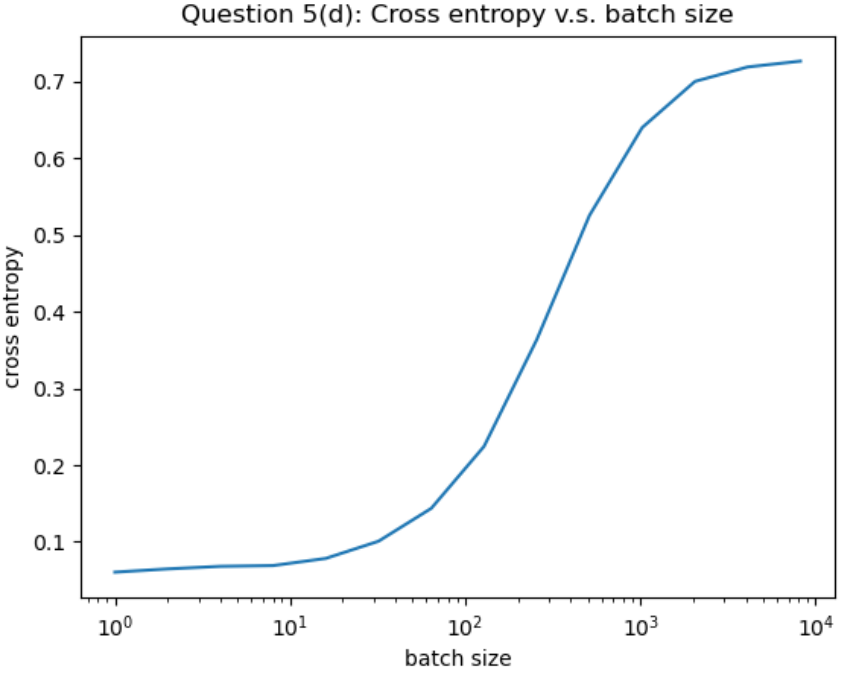
Question 3(e): different numbers of epochs



Question 3(f): different initial weights




Question 5(d): Accuracy v.s. batch size
accuracy
batch size
1.0
0.9
0.8
0.7
0.6
0.5
10^0
10^1
10^2
10^3
10^4


Question 5(d): Cross entropy v.s. batch size
cross entropy
0.7
0.6
0.5
0.4
0.3
0.2
0.1
10^0
10^1
10^2
10^3
10^4
batch size

University of Toronto Mississauga
CSC 311 - Introduction to Machine Learning

Cover sheet for Assignment 2

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: _____