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

Assignment 1: Questions and Solutions

Due date: Thursday October 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 progams 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.*

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. Be sure to read about indexing and slicing Numpy arrays. First, 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. 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`.

Whenever possible, *do not use loops*, which 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.

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. For generating and labelling plots, the following SciPy functions in `matplotlib.pyplot` are needed: `plot`, `scatter`, `xlabel`, `ylabel`, `title`, `suptitle` and `figure`. You can use Google to conveniently look up SciPy functions. e.g., you can google "numpy matmul" and "pyplot suptitle".

Again, because they are very slow, you should avoid the use of loops and iteration in your Python programs. For the same reason, you should not use recursion or higher-order functions (such as the python `map` function or any numpy function listed under "functional programming", such as `apply-along-axis`, which are just just loops in disguise), unless otherwise specified. Sometimes, you will need to use loops to iterate over short lists or to implement iterative algorithms, such as gradient descent. This is OK. However, 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 implement all the hard stuff for you. You may, of course, use Numpy's array-indexing facilities to vectorize operations on all or part of an array.

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. Also, unless specified otherwise, you may assume that all inputs are correct, so no error checking is required.

1. *Vectorized Code and Indexing.* (15 points total)

   This simple warm-up question illustrates Numpy's facilities for indexing and computing with arrays without using loops. In each question below, you should use at most one assignment statement and exactly one print statement (in addition to printing the question number). In questions (h) to (o) you should use one print statement and *no* assignment statements. Each question below has a simple solution. Do not use any loops. (1 point each.)

   Your code should look like this:

```
import numpy as np
import numpy.random as rnd

rnd.seed(3)

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

print('\nQuestion 1(a):')
B = ...
print(B)

print('\nQuestion 1(b):')
y = ...
print(y)

print('\nQuestion 1(c):')
C = ...
print(C)
```

   The statement `rnd.seed(3)` above initializes the seed of the random number generator to 3. This ensures that everyone will get exactly the same "random" vectors and

matrices and exactly the same answers to all the questions below. For this to work as intended, you should execute all your code for the questions below at once, in order, and you should only execute the rand function twice, to answer parts (a) and (b). Any additional executions of the `rand` function, between questions, will change the random matrices that are returned, which will change all your answers.

If you have done everything correctly, your anwers to parts (a) and (b) below should be exactly the following:

```
Question 1(a):
[[0.5507979  0.70814782 0.29090474 0.51082761 0.89294695]
 [0.89629309 0.12558531 0.20724288 0.0514672  0.44080984]
 [0.02987621 0.45683322 0.64914405 0.27848728 0.6762549 ]
 [0.59086282 0.02398188 0.55885409 0.25925245 0.4151012 ]]

Question 1(b):
[[0.28352508]
 [0.69313792]
 [0.44045372]
 [0.15686774]]
```

(a) Construct a random 4×5 matrix. Call it `B`. The 4 rows are numbered 0,1,2,3, and the 5 columns are numbered 0,1,2,3,4. We shall 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.

(b) Construct a random 4-dimensional column vector (that is, a 4×1 matrix). Call it `y`.

(c) Reshape `B` into a 2×10 matrix. Call the result `C`. `B` itself does not change. (The first row of `C` should consist of the first row of `B` followed by the second row of `B`.)

(d) Subtract vector `y` from all the columns of matrix `B`. Call the resulting matrix `D`. `D` has the same dimensions as `B`.

(e) Reshape `y` so that it is a 4-dimensional vector instead of a 4×1 matrix. That is, change its shape from (4,1) to (4). Call the resulting vector `z`. `y` itself does not change. (Note that `z` is neither a column vector nor a row vector. We say it has rank 1, since it has 1 dimension; while `y` and `B` have rank 2, since they each have 2 dimensions.)

(f) Change column 3 of matrix `B` to have the same value as vector `z`. (Note that column 3 is the 4th column, since column 0 is the first.)

(g) Add vector `z` to column 2 of matrix `B` and assign the result to column 0 of matrix `D`. Only matrix `D` changes.

(h) Print the first three rows of matrix `B` as a single matrix.

(i) Print columns 1 and 3 of matrix `B` as a single matrix.

(j) Compute the natural logarithm of each element in matrix `B`. The result is a matrix with the same dimensions as `B`.

(k) Compute the sum of all the elements in matrix `B`. The result is a single real number.

(l) Compute the maximum of each column of matrix `B`. The result is a 5-dimensional vector.

(m) Sum the elements in each row of matrix `B` and print the maximum sum. The result is single real number.

(n) Using matrix multiplication, compute $\mathtt{B}^T\mathtt{D}$, where $\mathtt{B}^T$ is the transpose of matrix `B`. The result is a 5×5 matrix.

(o) Compute $\mathtt{y}^T\mathtt{D}\mathtt{D}^T\mathtt{y}$. The result is a 1×1 matrix (which contains a single real number).

You should use the functions `reshape`, `sum`, `max` and `matmul` in `numpy`, as well as the function `random` in `numpy.random`. The expression `B.T` computes the transpose of matrix `B` (as does the Numpy function `transpose`). You may also find the Numpy function `shape` useful. You will have to look these functions up in the Numpy manual (simply google them) and read their specifications carefully.

2. *Vectorized v.s. non-Vectorized Code.* (16 points)

This simple warm-up question is meant to illustrate the vast difference in execution speed between iteration in Python (which is slow) and vectorized code (which is fast), using matrix operations in Numpy. As a fringe benefit, you will also learn how to implement matrix multiplication. To receive full marks for parts (a) and (b), the function executions in part (c) must also be correct. If these executions are not carried out or if they give completely incorrect results, then no marks will be given for parts (a) and (b).

(a) (6 points) Write a Python function `matrix_poly(A)` that computes $\mathtt{A}+\mathtt{A}^2+\mathtt{A}^3$, a simple polynomial of the square matrix `A`. Here, $\mathtt{A}^3$`=A*A*A` and $\mathtt{A}^2$`=A*A`, where `*` denotes matrix multiplication. Recall that matrix multiplication is defined as follows:

$$E_{ij} \;=\; \sum_k C_{ik} D_{kj} \tag{1}$$

where C and D are matrices and $E = CD$ is the matrix product of `C` and `D`. However, do not implement your function by naively evaluating the polynomial as written. Instead, implement it as `A+A*(A+A*A)`, which is faster since it does two matrix multiplications instead of three. Your program will need to use a triply-nested Python loop.

Your program should *not* use any NumPy operations that operate on whole matrices or large chunks of matrices. These include matrix addition and multiplication. Nor should you use fancy array indexing to operate on multiple array elements at

a time, as in Question 1. Instead, you should use loops to operate on matrices one element at a time, using assignment statements such as `C[i,j]=D[i,j]*E[i,j]`. You may define subroutines that take matrices as input and return matrices as output. You may also use the NumPy operations `shape` (to determine the dimensions of the input matrix) and `zeros` (to initialize your computations). You should not use any NumPy operations other than these.

(b) (4 points) Write a Python function `timing(N)` to measure execution speed. Specifically, the function should do the following:

- Use the function `random` in `numpy.random` to create a random `N` × `N` matrix, `A`.
- Execute your function `matrix_poly` with matrix `A` as its argument. Call the result `B1`. Use the function `time.time` to measure the execution time of this step. Print out the execution time.
- Use the functions `numpy.matmul` and `+` to compute `A+A*(A+A*A)`. Call the result `B2`. Do not use any loops. This is vectorized code. Use the function `time.time` to measure the execution time of this step. Print out the execution time.
- Compute and print out the magnitude of the difference matrix, `B1-B2`. There are many ways to define the magnitude of a matrix, but for the purpose of this question, we define it to be the square root of the sum of the squared values of the matrix elements. That is, the magnitude of a matrix, A, is $\sqrt{\sum_{ij} A_{ij}^2}$. Do *not* use iteration to compute this magnitude. Instead, use only NumPy operations. You can do this in one line of vectorized code.

If your function `matrix_poly` is working correctly, the last step should produce a very small number (much less than 10^{-5}), which is due to numerical error. You should also find that the vectorized code is *much* faster than using your `matrix_poly` function. In fact, when N is large, it can be thousands of times faster.

(c) (6 points total) Execute `timing(100)`, `timing(300)` and `timing(1000)`, and hand in the printed results (3 points). Be sure it is clear which measurement each printed value refers to. *In each case, how many floating-point multiplications does* `matrix_poly` *perform?* (3points)[1] You should observe that the execution time of `timing(N)` increases rapidly with `N`. This is because matrix multiplication is an $O(N^3)$ operation, so increasing N by a factor of 10 will increase execution time by a factor of 1000. Depending on your computer, `timing(1000)` could take 15-30 minutes to compute. If your computer is very slow, you may want to let it run over night.

ANSWER: Each matrix multiplication involving $N \times N$ matrices performs N^3 floating-point multiplications. (There are N^2 entries in the product matrix, and each one requires N floating-point multiplications to compute). There are two

[1] A floating-point multiplication is a single multiplication of two floating-point (*i.e.*, real) numbers.

matrix multiplications in `matrix_poly`, both involving $N \times N$ matrices, so a total of $2N^3$ floating-point multiplications are performed. Thus, `timing(100)` performs $2 * 100^3 = 2 * 10^6$, or two million floating-point multiplications. Likewise, `timing(300)` performs $2 * 300^3 = 2 * 27 * 10^6$, or 54 million floating-point multiplications. Finally, `timing(1000)` performs $2 * 1000^3 = 2 * 10^9$, or 2 billion floating-point multiplications.

Because loops and iteration in Python are so slow, your programs in the rest of this assignment should avoid using them to operate on large vectors and matrices. Instead, you should vectorize your code and use NumPy operations whenever possible to speed up computations.

3. *Basic Linear Least-Squares Regression.* (20 points)

   This warm-up exercise is intended to familiarize you with using `numpy` for basic data analysis and with using `pyplot` for basic data visualization. Specifically, you will write a Python program to fit a simple linear function to data using least-squares regression, measure the error, and plot the results.

   As described in class, the data for linear regression consists of a set of pairs, $(x^{(1)}, t^{(1)})$, ... $(x^{(N)}, t^{(N)})$, where each $x^{(n)}$ is an input and each $t^{(n)}$ is a target value. Each pair $(x^{(n)}, t^{(n)})$ is called a data point. In general, $x^{(n)}$ can be a vector, but in this question, it will simply be a real number. The function you will fit to the data takes a real-number, x, as input and returns a real number, $y(x)$, as output. It has the form

$$y(x) = ax + b \tag{2}$$

   Your job is to find values for a and b so that the function $y(x)$ best fits the data. In particular, you will minimize the loss function,

$$l(a, b) \;=\; \sum_{n=1}^{N} [t^{(n)} - y(x^{(n)})]^2 \tag{3}$$

   where the sum is over all training points, $(x^{(n)}, t^{(n)})$. Recall from Lecture 2 that the values of a and b that minimize this loss are given by the following equation:

$$w \;=\; (X^T X)^{-1} X^T t \tag{4}$$

   where $w = (b, a)$ is the weight vector, t is a column vector of the target values, and X is the data matrix. In this case, because the input, x, is a single real number, not a vector, X has only two columns: the first column is all 1's, and the second column consists of the input values $x^{(1)}$, $x^{(2)}$ ... $x^{(N)}$.

   In addition to fitting a linear function to data, your program should also compute the mean squared training and test errors of the fitted function. These are given by the

following equations:

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

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

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 `dataA1Q3.pickle.zip` on the course web site. Download and uncompress this file. (Your browser may uncompress it automatically.) 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:

```
with open('dataA1Q3.pickle','rb') as f:
    dataTrain,dataTest = pickle.load(f)
```

The variable `dataTrain` will now contain the training data, and `dataTest` will contain the test data. Specifically, `dataTrain` is a 2×30 Numpy array, where the first row gives the input values, and the second row gives the target values. Likewise, `dataTest` is a 2×1000 array of test data. The training data is illustrated in the scatter plot in Figure 1.

In answering the questions below, do not use any Python loops. Instead, all code should be vectorized.

(a) Write a Python function `least_squares(x,t)` that returns the optimal values of a and b. Here, `x` is a vector of input values, and `t` is a vector of target values. They are the training data. Your program should construct the data matrix X and use equation (4) to solve for a and b. You may find the function `inv` in `numpy.linalg` useful, and the function `numpy.ones`. The entire function can be written in at most 7 lines of highly-readible code, not counting comment lines.

(b) Write a function `plot_data(x,t)` that takes training data as input and plots the data and also plots the line that best fits the data in the least-squares sense. Here `x` and `t` define the training data, and are vectors of inputs and target values, respectively. You should call the function `least_squares` from part (a) to compute the values of a and b for the fitted line. Plot the training data as blue dots (as in Figure 1), and plot the fitted line ($y = ax + b$) in red. You can't plot the entire line, of course, because it is infinite. So just plot the fitted line from x_{min} to x_{max}, where x_{min} is the smallest value of x in the training set, and x_{max} is the largest. Use the functions `scatter` and `plot` in `matplotlib.pyplot` to plot the training points and draw the fitted line segment, respectively. Title the figure, *Question 3(b): the fitted line.* Finally, your function should return the values of a and b.

The entire function can be written in at most 11 lines of highly-readible code, not counting comment lines.

Figure 1:

(c) Write a function `error(a,b,X,T)` that measures how well a line fits a data set. The data is defined by the vectors `X` and `T`, and the line is defined by the real numbers `a` and `b`. That is, `X` is a vector of input values, `T` is a vector of corresponding target values, and the line is given by the equation $y = ax + b$. The function should return the mean squared error of the line with the data. In particular, you should be able to use this function to compute the training and test errors of your fitted function. You may find the function `numpy.mean` useful.

This function can be written in at most three lines of highly-readible code.

(d) Write and execute a simple Python script to test your functions above. The script should do the following:

- Read the training and test data from the file `dataA1Q3.pickle`.
- Call the function `plot_data` to fit a line to the training data and plot the results.
- Print the values of `a` and `b` for the fitted line.
- Compute and print the training error.
- Compute and print the test error.

If you have done everything correctly, the training and test errors should both be between 0.8 and 1.0, and the test error should be greater than the training error. Hand in the plot and the printed values.

4. *Binary Logistic Regression.* (28 points total)

In this question you will use logistic regression to generate a classifier for 3-dimensional (3D) cluster data, and you will derive some vectorized equations so you can implement logistic regresion yourself in Question 5.

Here, you will use the Python class `LogisticRegression` in `sklearn.linear_model`, which generates a Python object that does logistic-regression. Using this class, the following code will train a classifier and retrieve the weight vector and bias term:

```
import sklearn.linear_model as lin
clf = lin.LogisticRegression()    # create a classification object, clf
clf.fit(Xtrain,Ttrain)    # learn a logistic-regression classifier
w = clf.coef_[0]    # weight vector
w0 = clf.intercept_[0]    # bias term
```

In this question, you may use this code fragment, but do not use any other functions from `sklearn`, and unless otherwise specified, do not use any attributes or methods from `LogisticRegression`. The point is to implement things yourself.

The data you will use is in the file `dataA1Q4v2.pickle.zip` on the course web site. Download and uncompress this file. You can read the file with the following Python command:

```
with open('dataA1Q4v2.pickle','rb') as f:
    Xtrain,Ttrain,Xtest,Ttest = pickle.load(f)
```

The variables `Xtrain` and `Ttrain` will now contain training data, that is, 3D input points and corresponding target values, respectively. Likewise, `Xtest` and `Ttest` contain test data. There are 1,000 training points and 1,000 test points from each class. The training data is illustrated in the scatter plot in Figure 2. The figure shows two clusters, one red and one blue, which are called cluster 0 and cluster 1, respectively, in the target data.

In addition, download the file `bonnerlib3D.py.zip` from the course web site and import it into your program with the statement `import bonnerlib3D as bl3d`. It contains functions for displaying 3-dimensional data and decision boundaries. To fully utilize these functions and to animate your data, you will need to put the backend of your Spyder environment into Qt mode. The the following short video explains how to do this: `https://www.youtube.com/watch?v=RyiLaaflmN4`.

Use the function `bl3d.plot_data` to display and view the data from different angles in 3D. The function `bl3d.movie_db` will slowly rotate the data for you. A comment at the beginning of the file describes the function arguments. Try executing the expression `bl3d.plot_data(Xtrain,Ttrain,30,10)`, and then use your mouse to rotate the data. Do not hand any of this in.

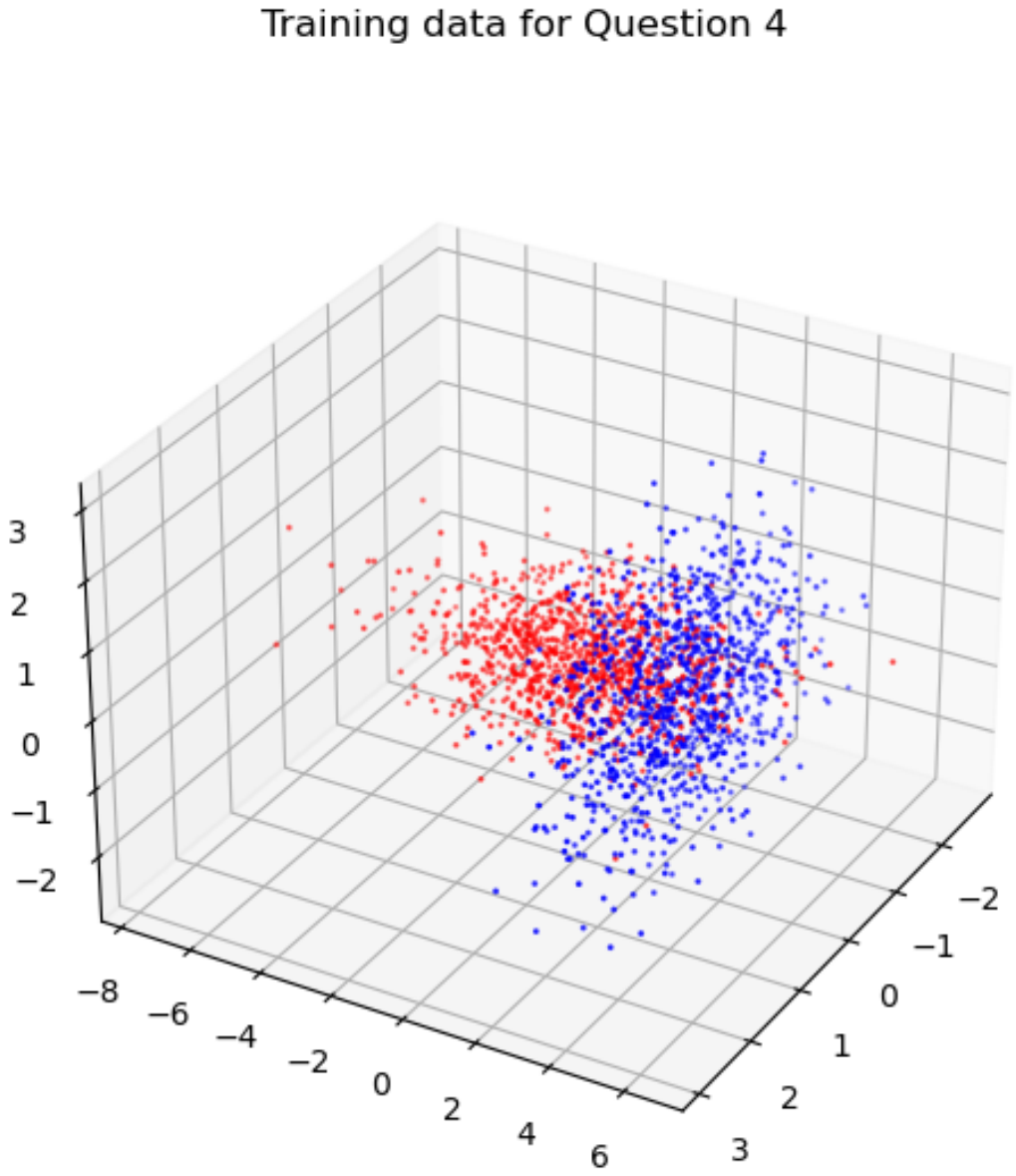
In answering the questions below, do not use any Python loops. Instead, all code should be vectorized. To receive full marks for parts (a) and (b), parts (c) and (d) must also be correct.

(a) (2 points) Retrieve the training and test data. Use the training data to train a logistic-regression classifier. Print out the values of the weight vector and the bias term.

(b) (4 points) Compute the accuracy of your logistic-regression classifier on the test data. Do this in two ways: (1) using the `score` method of the `LogisticRegression` class, and (2) from the weight vector and bias term without using any attributes or methods of `LogisticRegression` or any functions in `sklearn`. In the second approach, you will have to make predictions on the test data. The accuracy is then the average number of correct predictions. Call the two estimates of accuracy `accuracy1` and `accuracy2`, respectively. Print out the two estimates of accuracy and their difference. The two estimates should be the same and the difference should be 0. This can all be done in 8 lines of highly-readible code (not counting comment lines) without using loops,.

(c) (1 point) Use the function `plot_db` in `bonnerlib3D` to plot the decision boundary on top of the training data. Use an elevation of 30 degrees and an azimuth of 5 degrees. Using `suptitle` in `matplotlib.pyplot`, title the plot, *Question 4(c): Training data and decision boundary.* Hand in this plot from this viewing direction; *i.e.*, do not rotate the plot before you save it and hand it in. To get a better impression of the decision boundary, you can use the function `movie_db` in `bonnerlib3D` to see a movie of the decision boundary slowly rotated. Do not hand this in.

Figure 2:



(d) (1 point) Repeat part (c) using the same elevation but an azimuth of 20 degrees (and use `4(d)` instead of `4(c)` in the figure title).

(e) (15 points) Slide number 61 of Lecture 3 (Linear Classification) gives the following equation for the gradient of the loss function for logistic regression:

$$\frac{\partial \mathcal{J}}{\partial w_j} = \sum_{i=1}^{N} (y^{(i)} - t^{(i)}) x_j^{(i)} / N \tag{5}$$

Prove that this equation is equivalent to the following vectorized version:

$$\frac{\partial \mathcal{J}}{\partial w} = X^T (y - t)/N \tag{6}$$

Here, w is a column vector whose j^{th} element is w_j. Likewise, y and t are column vectors whose i^{th} elements are $y^{(i)}$ and $t^{(i)}$, respectively. X is the data matrix, in which each row is an input vector, that is, $X_{ij} = x_j^{(i)}$. Note that Equation (6) involves matrix transpose, matrix-vector multiplication and the gradient of a function, so you will need to use the definitions of these three concepts in your proof. (And for full marks you must explicitly state where you use them.)

Also note that both sides of Equation (6) evaluate to vectors, so part of your proof involves showing that two vectors are equal. One way to do this is to show that corrresponding elements of the vectors are all equal. In this case, you must show that

$$\left[\frac{\partial \mathcal{J}}{\partial w}\right]_j = [X^T (y - t)/N]_j \tag{7}$$

for all j. Here, the notation $[V]_j$ refers to element j of vector V. Note that in some cases we don't need the square brackets. For example, $[w]_j = w_j$. However, when V is a complicated expression, as in the above equation, we do need them. We can do the same thing for matrices. Here are some equalities you may find useful:

$$\left[X^T\right]_{ji} = X_{ij} \qquad \text{and} \qquad \left[\frac{\partial \mathcal{J}}{\partial w}\right]_j = \frac{\partial \mathcal{J}}{\partial w_j}$$

The first is just the definition of matrix transpose, and the second is the definition of the gradient of a function. You will need other such equalities in your proof. Some may be trivial (such as $[w]_j = w_j$). Others should be justified.

To prove that Equations (5) and (6) are equivalent, you must prove that Equation (5) is true for all j *if-and-only-if* Equation (6) is true. The proof requires some subtle logic and the proper handling of *if-and-only-if* statements. Alternatively, you can prove that for all j, Equation (5) is true *if-and-only-if* Equation (7) is true. In either case, be sure to prove both directions (*i.e.*, both *if* and *only if*). If you are clever, you will not need separate proofs for each direction, but can prove them both at once by using *if-and-only-if* statements. (This is the preferred and more-elegant approach, and will receive more marks.) For full marks, the proof must be entirely rigorous, and every step should be justified. Do not skip any steps.

Hint: As a warm-up exercise, prove that if Equation (5) is true for j, then Equation (7) is true for j. Proving only this hint is worth 10 out of the 15 points for this question. Proving equivalence amounts to also showing that if Equation (5) is *not* true for j, then Equation (7) is *not* true for j.

ANSWER: We first prove that the following equality holds for all j:

$$\begin{aligned}
\sum_{i=1}^{N}(y^{(i)}-t^{(i)})x_j^{(i)} &= \sum_{i=1}^{N}(y^{(i)}-t^{(i)})X_{ij} && \text{by the definition of } X \\
&= \sum_{i=1}^{N}([y]_i-[t]_i)X_{ij} && \text{by the definition of } [y]_i \text{ and } [t]_i \\
&= \sum_{i=1}^{N}[y-t]_i X_{ij} && \text{by the definition of vector subtraction} \\
&= \sum_{i=1}^{N}[y-t]_i [X^T]_{ji} && \text{by the definition of matrix transpose} \\
&= \sum_{i=1}^{N}[X^T]_{ji}[y-t]_i && \text{trivially} \\
&= [X^T(y-t)]_j && \text{by the definition of matrix-vector multiplication}
\end{aligned}$$

We thus have the following equivalence, for all j,

$$\begin{aligned}
& \partial\mathcal{J}/\partial w_j = \textstyle\sum_{i=1}^{N}(y^{(i)}-t^{(i)})x_j^{(i)}/N \\
\textit{iff}\quad & [\partial\mathcal{J}/\partial w]_j = \textstyle\sum_{i=1}^{N}(y^{(i)}-t^{(i)})x_j^{(i)}/N && \text{by the definition of gradient} \\
\textit{iff}\quad & [\partial\mathcal{J}/\partial w]_j = [X^T(y-t)]_j/N && \text{by the equality proved above} \\
\textit{iff}\quad & [\partial\mathcal{J}/\partial w]_j = [X^T(y-t)/N]_j && \text{by the definition of matrix-scalar multiplication}
\end{aligned}$$

This shows that for all j, Equation (5) is equivalent to Equation (7), which is enough to show its equivalence to Equation (6). The following additional logic, which connects Equations (5) and (6) directly, is optional:

$$\begin{aligned}
& \partial\mathcal{J}/\partial w_j = \textstyle\sum_{i=1}^{N}(y^{(i)}-t^{(i)})x_j^{(i)}/N \quad \text{for all } j \\
\textit{iff}\quad & [\partial\mathcal{J}/\partial w]_j = [X^T(y-t)/N]_j \quad \text{for all } j, \qquad \text{by the equivalence proved above} \\
\textit{iff}\quad & \partial\mathcal{J}/\partial w = X^T(y-t)/N \qquad \text{by the definition of vector equality}
\end{aligned}$$

Thus, Equation (5) is equivalent to Equation (6).

To prove only the hint, assume that Equation (5) is true for j. Then,

$$\begin{aligned}
[\partial \mathcal{J}/\partial w]_j &= \partial \mathcal{J}/\partial w_j \qquad \text{by the definition of gradient} \\
&= \sum_{i=1}^{N} (y^{(i)} - t^{(i)}) x_j^{(i)}/N \qquad \text{by Equation (5)} \\
&= [X^T(y-t)]_j/N \qquad \text{by the first equality proved above} \\
&= [X^T(y-t)/N]_j \qquad \text{by the definition of matrix-scalar multiplication}
\end{aligned}$$

Thus Equation (7) is true for j.

(f) (5 points) Prove the equation for the logistic cross entropy in the third bullet on slide number 58/70 of Lecture 3 on Linear Classification. This equation is for a single data point, (x, t). (Note: The left-hand equation in bullet 3 is the definition of logistic cross entropy. The right-hand equation is what you are to prove.) You will need to use the definition of cross entropy and the definition of the logistic function. For full marks, the proof must be entirely rigorous, and every step should be justified. Do not skip any steps.

ANSWER: We first note the following equality:

$$\begin{aligned}
1 - \sigma(z) &= 1 - \frac{1}{1+e^{-z}} \qquad \text{by the definition of } \sigma(z) \\
&= \frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}} \qquad \text{trivially} \\
&= \frac{e^{-z}}{1+e^{-z}} \qquad \text{by basic algebra} \\
&= \frac{e^{-z}e^{z}}{(1+e^{-z})e^{z}} \qquad \text{trivially} \\
&= \frac{1}{1+e^{z}} \qquad \text{by basic algebra}
\end{aligned}$$

Thus,

$$\begin{aligned}
\mathcal{L}_{LCE}(z,t) &= \mathcal{L}_{CE}(\sigma(z), t) \qquad \text{by the definition of } \mathcal{L}_{LCE} \\
&= -t \log \sigma(z) - (1-t) \log [1 - \sigma(z)] \qquad \text{by the definition of } \mathcal{L}_{CE} \\
&= -t \log [1/(1+e^{-z})] - (1-t) \log [1 - \sigma(z)] \qquad \text{by the definition of } \sigma(z) \\
&= -t \log [1/(1+e^{-z})] - (1-t) \log [1/(1+e^{z})] \qquad \text{by the equality noted above} \\
&= t \log (1+e^{-z}) + (1-t) \log (1+e^{z}) \qquad \text{since } \log(1/x) = -\log x
\end{aligned}$$

5. *Gradient Descent.* (30 points)

Implement (batch) gradient descent for binary logistic regression using the cross-entropy loss function. Your implementation should work for input data of any dimensionality, but you should test it on the 3-dimensional data of Question 4. In particular,

you should define a Python function `gd_logreg(lrate)` that performs gradient descent where `lrate` is the learning rate. You may assume that the training and test data are stored in global variables. The function should use only one loop, but no nested loops, and all other code should be vectorized. Use the results from Question 4 to vectorize the weight updates. You should not use any functions from `sklearn`. The point is to implement logistic regression yourself using simple linear-algebra operations.

In addition to gradient descent, the `gd_logreg` function should do the following:

(a) (0 points) The first statement in your function should be `numpy.random.seed(3)`. This ensures that everyone will use the same randomly-initialized weight vector and get the same final answers (if their programs work correctly).

(b) (1 point) Initialize the weight vector (including the bias term) by using `randn` in `numpy.random` to generate a random vector, and then dividing this vector by 1000. This ensures that the initial weights are both random and near zero (which makes the performance of gradient descent more predictable, as there is now a smaller range of inital values.)

(c) (5 points) Recall that gradient descent is an iterative algorithm that performs weight updates at each iteration. In addition, at each iteration your function should also compute the average cross entropy of the classifier on the training and test data, and the accuracy of the classifier on the training and test data.[2] You should store these cross entropies and accuracies in four separate lists, to record the progress of gradient descent. You should not compute these values until after the first weight update has been performed. We shall refer to the two average cross entropies as the training cross entropy and the test cross entropy, respectfully.

(d) (5 points, including computation of the gradient) Perform weight updates until the training cross entropy changes by less that 10^{-10} between two successive updates.

(e) (6 points) After gradient descent has terminated, print out the final weight vector (including the bias term), the number of iterations that were performed, and the learning rate. Also print out the weight vector and bias term that you computed in Question 4. It should be very similar to your weight vector here, with 1 or 2 significant digits of similarity in each weight. (Most of the points are for this last sentence, which proves the correctness of the implementation.)

(f) (3 points) In a single figure, plot the list of training cross entropies (in blue), and the list of test cross entropies (in red). Title the figure, *Question 5: Training and test loss v.s. iterations*. Label the vertical axis *Cross entropy* and the horizontal axis *Iteration number*. If everything is working correctly, the figure should show the cross entropy decreasing very rapidly and then flattening out.

(g) (2 points) To see the behavior of the gradient-descent algorithm more clearly, replot the previous figure using a log scale on the horizontal axis. You can use the function `semilogx` in `matplotlib.pyplot` to do this. Title this figure, *Question*

[2]Recall that accuracy is the average number of correct predictions.

5: Training and test loss v.s. iterations (log scale), and label the axes as before. If everything is working correctly, the cross entropy should decrease smoothly from left to right (with a possible short, flat segment at the very start).

(h) (4 points) In a single figure, plot the list of training accuracies (in blue), and the list of testing accuracies (in red). Use a log scale on the horizontal axis. Title the figure, *Question 5: Training and test accuracy v.s. iterations (log scale)*. Label the vertical axis *Accuracy* and the horizontal axis *Iteration number*. Both accuracies should increase somewhat jaggedly from left to right and eventualy flatten out.

(i) (1 point) The plots of cross entropy eventually flatten out, suggesting that the cross entropy has stopped changing. To see that it is in fact still changing, plot the last 100 training cross entropies in a figure by themselves (without the test cross entropies). Title the figure, *Question 5: last 100 training cross entropies*. Label the axes as before. You should observe that cross entropy is still decreasing.

(j) (2 points) Although the training cross entropy is decreasing, the test cross entropy should have bottomed out and begun to increase. This suggests that too many weight updates were performed and that over-fitting is now taking place. To see this, plot all but the first 50 test cross entropies in a single plot by themselves (without any training cross entropies). Use a log scale on the horizontal axis. Title the figure, *Question 5: test loss from iteration 50 on (log scale)*. Label the axes as before. If everything is working correctly, you should observe that the test cross entropy initially decreases, then bottoms out and increases, and finally flattens out.

(k) (1 point) Use `plot_db` in `bonnerlib3D` to plot the decision boundary and training data in 3D. Use an elevation of 30 degrees and an azimuth of 5 degrees, as in Question 4(c). Title the figure, *Question 5: Training data and decision boundary*.

Using the data from Question 4, you will have to experiment to find a good learning rate. To do this, try performing 200 iterations of gradient descent using these five learning rates: 10, 3, 1, 0.3, and 0.1. Use the largest learning rate that gives smooth curves of cross entropy that behave as described above. Do not hand in any of the output generated during this exploration. Instead, using the best of these five learning rates, run `gd_logreg` until the training cross entropy changes by less than 10^{-10}, as described above. Hand in all the figures and output generated during this run. You should find that the training cross entropy is generally lower than the test cross entropy, and that the training accuracy is generally greater. You should also get the same results as in Question 4. That is, after gradient descent terminates, the weight vector, the decision boundary, the training accuracy and the test accuracy should all be very similar to those in Question 4.

6. *Nearest Neighbours.* (35 points)

In this question, you will use K nearest neighbours (KNN) to classify images of handwritten digits from MNIST, a benchmark machine-learning dataset. There are ten different digits (0 to 9), but to save time, you will use a reduced data set consisting of

only two digits, so this will be a binary classification problem. Your main job is to use validation data to determine the best value of the hyperparameter K, the number of neighbours. To perform KNN, you will use the Python class `KNeighborsClassifier` in `sklearn.neighbors`. It is used in much the same way as the `LogisticRegression` class in Question 4.

To start, download and uncompress (if necessary) the MNIST data file from the course web page. The file, called `mnistTVT.pickle.zip`, contains training, validation and test data. Next, start the Python interpreter and read the file with the following command:

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

The variables `Xtrain` and `Ttrain` contain training data, while `Xval` and `Tval` contain validation data, and `Xtest` and `Test` contain test data.

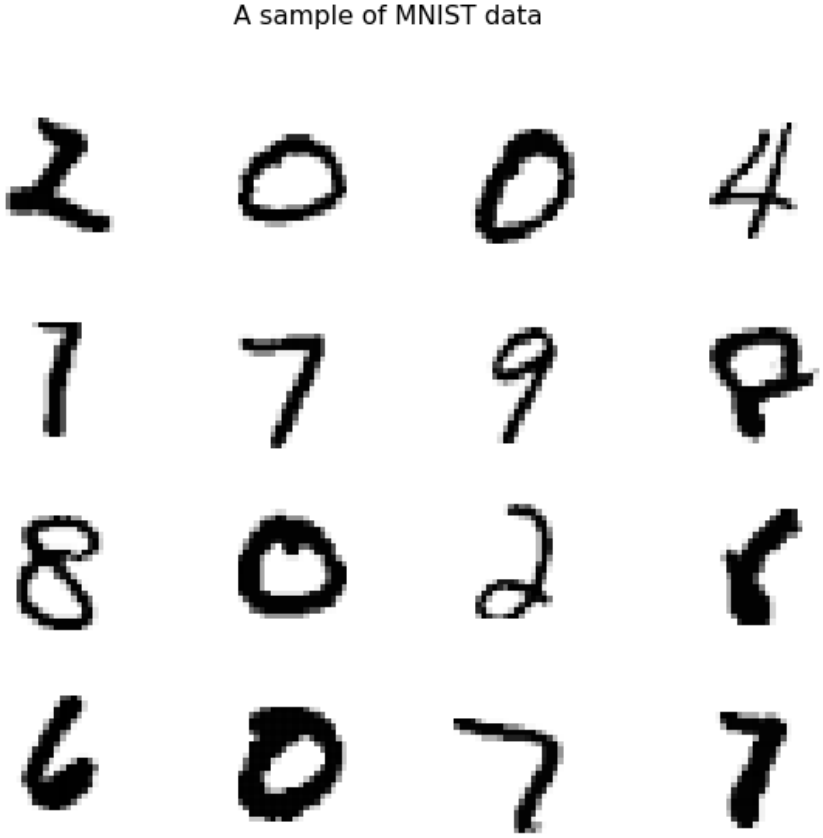
`Xtrain` is a Numpy array with 50,000 rows and 784 columns. Each row represents a hand-written digit. Although each digit is stored as a row vector with 784 components, it actually represents an array of pixels with 28 rows and 28 columns ($784 = 28 \times 28$). Each pixel is stored as a floating-point number, but has an integer value between 0 and 255 (i.e., the values representable in a single byte). The variable `Ttrain` is a vector of 50,000 image labels, where a label is an integer betwen 0 and 9. For example, if row `n` of `Xtrain` is an image of the digit 7, then `Ttrain[n]` $= 7$. Likewise for the validation and testing data, which contain 10,000 images each.

To view a digit, you must first convert it to a 28×28 array using the function `numpy.reshape`. To display a 2-dimensional array as an image, you can use the function `imshow` in `matplotlib.pyplot`. To see an image in black-and-white, add the keyword argument `cmap='Greys'` to `imshow`. To remove the smoothing and see the 784 pixels clearly, add the keyword argument `interpolation='nearest'`. Try displaying a few digits as images. (Figure 3 shows some examples.) For comparison, try printing them as vectors. (Do not hand this in.)

In answering the questions below, do not use any Python loops, except where explicity allowed. All code should be vectorized.

(a) (5 points) Because KNN is so slow, you will not use the entire MNIST data set. Instead, construct a reduced data set consisting of only the digits 5 and 6. You will need reduced versions of the training, validation and test data. The reduced training set should have just under 10,000 images, and the reduced validation and test sets should have just under 2,000 images each. You should preserve the order of the data. For instance, if image i comes before image j in the training data, then i should also come before j in the reduced version of the training data. Since the training set has been randomly shuffled, your reduced data sets should have fives and sixes randomly interleaved. In particular, all the fives should not come before all the sixes in the reduced data set (nor vice-versa). Hint: use boolean index arrays (see the Numpy manual).

Figure 3:



In addition, to save even more time, you should construct two versions of the reduced training set. One version containing all the fives and sixes (the full version), which you will use as the training data; and a smaller version containing the first 2000 elements of the full version (the small version), which you will use for estimating training error.

(b) (5 points) From the reduced training data (full version), extract and display the first 16 digits. Display them in a single figure, arranged on a 4×4 grid, using the function `matplotlib.pyplot.subplot`. Turn off the axes in each image, using the function `matplotlib.pyplot.axis`. The images should be in black-and-white and should not use any smoothing. Title the figure, *Question 6(b): 16 MNIST training images*. You should display these 16 digits in order from left to right and top to bottom. (So, the first image is in the top left corner, and the fourth image is in the top right corner). You may use one loop for this question, but no nested loops.

(c) (10 points) Write a python program that uses the validation data to determine the best value of K, the number of neighbors in KNN. You should use the `fit` and `score` methods of the `KNeighborsClassifier` class in `sklearn.neighbors`, but no other methods or functions in `sklearn`. Specifically, your program should do the following:

i. For odd values of K from 1 to 19, inclusive,
   - Fit a KNN classifier to the full version of the reduced training data.
   - Compute the accuracy of the fitted classifier on the reduced validation data.
   - Compute the accuracy of the classifier on the small version of the reduced training data.
   - Record the values of the validation and training accuracy.

   You may, of course, use a loop to iterate over the values of K. The entire loop may take about 10 minutes or more to run (at least, it did on my laptop). If everything is working properly, you should find that the training and validation accuracies are both very high, about 0.99 and higher.

ii. In a single figure, plot the training accuracy (in blue) and the validation accuracy (in red). Use the function `plot` in `matplotlib.pyplot`. Title the figure, *Question 6(c): Training and Validation Accuracy for KNN, digits 5 and 6*. Label the horizontal axis, *Number of Neighbours, K*. Label the vertical axis, *Accuracy*. If everything is working properly, you should find that the training accuracy generally decreases as K increases, whereas the validation accuracy increases slightly at first, then peaks, then decreases. Also, for every value of K, the training accuracy should be greater than the validation accuracy.

iii. Determine the best value of K, that is, the value that produces the greatest validation accuracy. If two values of K have the same validation accuracy, then choose the smaller one.

iv. Compute the accuracy of KNN on the reduced test data using the best value of K determined above.

v. Print the best value of K.

vi. Print the validation and test accuracies for the best value of K.

(d) (5 points) Repeat parts (a), (b) and (c) using digits 4 and 7, instead of 5 and 6. Adjust the figure titles appropriately. If your code is modular (and for full marks, it should be), you can do this part in just a handful of lines with very little effort.

(e) (4 points) You should find that the validation accuracy in part (d) is considerably higher than in part (c). In particular, you should find that the curve of validation accuracy overlaps the curve of training accuracy in part (d), while it is well below the training curve in part (c). You should also find that the best value of K is considerably higher in part (d). Suggest reasons that would explain both these observations.

ANSWER: The digits 5 and 6 look very similar, especially when handwritten, while the digits 4 and 7 look quite different. This means that a 5 could easily have a 6 in its neighbourhood, especially when one or the other is sloppily written. It is less likely that a 4 will have a 7 in its neighbourhood. This in turn means that for KNN, prediction is harder for 5s and 6s, so we should expect the accuracy for 5s and 6s to be lower than for 4s and 7s. Distinguishing 4s from 7s is simply an easier problem for KNN. (This explanation, or one like it, must be given. 2 points)

For the second observation, consider a sloppily written 4 that looks a bit like a 7. Such a 4 is a rare thing and is likely to have many 7s as nearby neighbours, so this 4 may be erroneously classified as a 7 by KNN. Only by using a higher value of K, and thus looking further away, will other 4s be found, and only then can a correct prediction be made. This is not the case for a sloppily written 5, which may be just as likely to have other (sloppy) 5s as nearby neighbours as 6s. (Any reasonable explanation can be given. The point is to think about the problem. 2 points)

(f) (2 points) In the program described in part (c), why do we only consider odd values of K?

ANSWER: Becuase this is a *binary* classification problem, an odd number of eightbours ensures that there will never be any tie votes, where an equal number of neighbours votes for each class.

(g) (4 points) Explain why KNN produces such high accuracies on the MNIST data, especially for these binary classification problems.

ANSWER: The high accuracy reflects the large number of training examples in MNIST. There are 50,000 training examples, which is 5,000 for each of the ten digits on average. With so many examples of each digit, it is very likely

that for almost any handwritten digit, there will be many similar-looking digits from the same class in the training set; that is, each digit will be surrounded by neighbours from the same class. Under these conditions, KNN will make many correct predictions, and thus have high accuracy.

140 points total

University of Toronto Mississauga
CSC 311 - Introduction to Machine Learning

Cover sheet for Assignment 1

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: _____

```
###########  Programming solutions for Assignment 1  ###############


import numpy as np
import numpy.random as rnd
import numpy.linalg as la
import matplotlib.pyplot as plt
import sklearn.neighbors
import sklearn.linear_model as lin
import pickle
import time
import bonnerlib3D as bl3d



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

rnd.seed(3)

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

print('\nQuestion 1(a):')
B = rnd.random((4,5))
print(B)

print('\nQuestion 1(b):')
y = rnd.random((4,1))
print(y)

print('\nQuestion 1(c):')
C = np.reshape(B,(2,10))
print(C)

print('\nQuestion 1(d):')
D = B - y
print(D)

print('\nQuestion 1(e):')
z = np.reshape(y,(4))
print(z)

print('\nQuestion 1(f):')
B[:,3] = z
print(B)

print('\nQuestion 1(g):')
D[:,0] = B[:,2] + z
print(D)
```

```
print('\nQuestion 1(h):')
print(B[:3])

print('\nQuestion 1(i):')
print(B[:,[1,3]])

print('\nQuestion 1(j):')
print(np.log(B))

print('\nQuestion 1(k):')
print(np.sum(B))

print('\nQuestion 1(l):')
print(np.max(B,axis=0))

print('\nQuestion 1(m):')
print(np.max(np.sum(B,axis=1)))

print('\nQuestion 1(n):')
print(np.matmul(B.T,D))

print('\nQuestion 1(m):')
print(np.matmul(y.T,np.matmul(D,np.matmul(D.T,y))))




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

# Question 2(a).

# Multiply matrices A and B
def mymult(A,B):
    L,M = np.shape(A)
    M,N = np.shape(B)
    C = np.zeros((L,N))    # space for the product matrix
    for l in range(L):
        for n in range(N):
            # compute element l,n of the product matrix
            x = 0.0
            for m in range(M):
                x += A[l,m]*B[m,n]
            C[l,n] = x
    return C


# add matrices A and B
def myadd(A,B):
```

```
    M,N = np.shape(A)
    C = np.zeros((M,N))    # space for storing A+B
    for m in range(M):
        for n in range(N):
            C[m,n] = A[m,n] + B[m,n]
    return C


def matrix_poly(A):
    return myadd(A,mymult(A,myadd(A,mymult(A,A))))



# Question 2(b).
# Compare two methods of evaluating a matrix expression.
def timing(N):
    # create a random NxN matrix
    A = rnd.random([N,N])

    # measure execution time of myfun(A,B)
    t1 = time.time()
    B1 = matrix_poly(A)
    t2 = time.time()
    print('')
    print('Execution time of matrix_poly({}):  {}'.format(N,t2-t1))

    # measure execution time of equivalent NumPy operations
    t1 = time.time()
    B2 = A + np.matmul(A, A + np.matmul(A,A))
    t2 = time.time()
    print('Execution time of equivalent NumPy operations:
{}'.format(t2-t1))

    # compute the magnitude of B1-B2
    mag = np.sqrt(np.sum((B1-B2)**2))
    print('Magnitude of B1-B2:  {}'.format(mag))



# Question 2(c).
print('\n')
print('QUESTION 2(c).')
print('-------------')
timing(100)
timing(300)
timing(1000)
```

```
##########  QUESTION 3  ############

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


# Question 3(a).
# fit a straight line to the data set x,t
def least_squares(x,t):
    N = len(x)    # number of data points
    X = np.ones([N,2])    # space for the data matrix
    X[:,1] = x    # insert the training data into the data matrix
    C = np.matmul(la.inv(np.matmul(X.T,X)),X.T)
    w = np.matmul(C,t)    # compute the weights
    return w[1],w[0]


# Question 3(b).
# plot the data set x,t and a fitted line.
def plot_data(x,t):
    a,b = least_squares(x,t)
    x1 = np.min(x)
    x2 = np.max(x)
    y1 = a*x1 + b
    y2 = a*x2 + b
    plt.figure()
    plt.scatter(x,t)
    plt.plot([x1,x2],[y1,y2],c='r')
    plt.title('Question 3(b): the fitted line')
    return a,b


# Question 3(c).
# compute the mean squared-error of a linear function on a data set.
def error(a,b,X,T):
    Y = a*X + b    # predicted values
    return np.mean((T-Y)**2)


# Question 3(d).
# read the training and testing data from a file
with open('dataA1Q3.pickle','rb') as f:
    dataTrain,dataTest = pickle.load(f)
# extract the input and target values
Xtrain,Ytrain = dataTrain
Xtest,Ytest = dataTest

# fit a line to the training data and plot the result
```

```
a,b = plot_data(Xtrain,Ytrain)
print('a =',a)
print('b =',b)
print('')

# compute training and test errors
errTrain = error(a,b,Xtrain,Ytrain)
errTest = error(a,b,Xtest,Ytest)
print('Training error =',errTrain)
print('Test error =',errTest)
print('')



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

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


print('Question 4(a)')
# read the training and testing data
with open('dataA1Q4v2.pickle','rb') as f:
    Xtrain,Ttrain,Xtest,Ttest = pickle.load(f)
clf = lin.LogisticRegression()    # classification object
clf.fit(Xtrain,Ttrain)    # learn a logistic-regression classifier
w = clf.coef_[0]    # weight vector
w0 = clf.intercept_[0]    # bias term
print('weight vector =',w)
print('bias term =',w0)


print('\nQuestion 4(b)')
accuracy1 = clf.score(Xtest,Ttest)
# make predictions and compute accuracy
z = Xtest @ w + w0    # the fitted linear function
predictions = (z > 0)    # vector of predictions
errors = np.abs(Ttest - predictions)    # vector of 0-1 losses
accuracy2 = 1 - np.mean(errors)
print('accuracy1 =',accuracy1)
print('accuracy2 =',accuracy2)
print('accuracy1 - accuracy2 =',accuracy1-accuracy2)


# Question 4(c)
bl3d.plot_db(Xtrain,Ttrain,w,w0,30,5)
plt.suptitle('Question 4(c): Training data and decision boundary')
```

```
# Question 4(d)
bl3d.plot_db(Xtrain,Ttrain,w,w0,30,20)
plt.suptitle('Question 4(d): Training data and decision boundary')



##########  QUESTION 5  ############

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

with open('dataA1Q4v2.pickle','rb') as f:
    Xtrain,Ttrain,Xtest,Ttest = pickle.load(f)


# T is a vector of target values.
# Z is a vector of logits.
# return the average cross entropy
def cross_entropy(Z,T):
    CE = T @ np.logaddexp(0,-Z) + (1-T) @ np.logaddexp(0,Z)    # total
cross entropy
    N = len(Z)
    return CE/N    # avergae cross entropy


# compute accuracy.  Z and T as above.
def accuracy(Z,T):
    predictions = (Z > 0)    # vector of predictions
    errors = np.abs(T - predictions)    # vector of 0-1 losses
    accuracy = 1 - np.mean(errors)    # mean number of correct
predictions
    return accuracy


# extend data matrix X by adding a column of 1's
def extend(X):
    N,M = np.shape(X)
    ones = np.ones([N,1])    # column vector of 1's.
    EX = np.concatenate([ones,X],axis=1)    # shape = [N,M+1]
    return EX


# the logistic function
def logistic(Z):
    return 1/(1+np.exp(-Z))
```

```
# fit a function to the training data using gradient descent.
def gd_logreg(lrate):
    rnd.seed(3)
    # extend the data matrices
    EXtrain = extend(Xtrain)
    EXtest = extend(Xtest)
    Ntrain,M = np.shape(EXtrain)
    # initialization phase
    W = rnd.randn(M)/1000    # initialize the weight vector randomly
    Ztrain = EXtrain @ W    # initial value of the linear function on
the training data
    # initialize lists for recording error measures
    CEtrainList = []    # training cross entropy
    CEtestList = []   # test cross entropy
    accTrainList = []    # training accuracy
    accTestList = []    # test accuracy
    # initialize loop parameters
    CEchange = np.inf    # change in cross entropy between iterations
    epsilon = 10**(-10)    # threshold for loop termination
    I = 0   # number of iterations so far

    # perform gradient descent
    # while I < 200:
    while CEchange > epsilon:
        # perform one step of gradient descent
        Ytrain = logistic(Ztrain)
        gradW = EXtrain.T @ (Ytrain-Ttrain)/Ntrain    # gradient of
loss function wrt W
        W = W - lrate*gradW    # update the weight vector
        # evaluate the linear function on the training and test data
        Ztrain = EXtrain @ W
        Ztest = EXtest @ W
        # compute error measures
        CEtrain = cross_entropy(Ztrain,Ttrain)
        CEtest = cross_entropy(Ztest,Ttest)
        accTrain = accuracy(Ztrain,Ttrain)
        accTest = accuracy(Ztest,Ttest)
        # record error measures
        CEtrainList.append(CEtrain)
        CEtestList.append(CEtest)
        accTrainList.append(accTrain)
        accTestList.append(accTest)
        # change in training error
        if I > 0:
            CEchange = CEtrainList[-2] - CEtrain
        I = I+1

    # plot the lists of training and test losses
    plt.figure()
    plt.plot(CEtrainList,'b')
```

```
    plt.plot(CEtestList,'r')
    plt.suptitle('Question 5: training and test loss v.s. iterations')
    plt.xlabel('Number of iterations')
    plt.ylabel('Cross entropy')

    # plot the lists of training and test losses (log scale)
    plt.figure()
    plt.semilogx(CEtrainList,'b')
    plt.semilogx(CEtestList,'r')
    plt.suptitle('Question 5: training and test loss v.s. iterations
(log scale)')
    plt.xlabel('Iteration number')
    plt.ylabel('Cross entropy')

    # plot the lists of training and test accuracies (log scale)
    plt.figure()
    plt.semilogx(accTrainList,'b')
    plt.semilogx(accTestList,'r')
    plt.suptitle('Question 5: training and test accuracy v.s.
iterations (log scale)')
    plt.xlabel('Iteration number')
    plt.ylabel('Accuracy')

    # plot the last 100 training losses
    plt.figure()
    plt.plot(range(I-100,I),CEtrainList[-100:],'b')
    plt.suptitle('Question 5: last 100 training losses')
    plt.xlabel('Iteration number')
    plt.ylabel('Cross entropy')

    # plot all but the first 50 test losses
    plt.figure()
    plt.semilogx(range(50,I),CEtestList[50:],'r')
    plt.suptitle('Question 5: test loss from iteration 50 on (log
scale)')
    plt.xlabel('Iteration number')
    plt.ylabel('Cross entropy')

    # plot the training data and decision boundary in 3D
    bl3d.plot_db(Xtrain,Ttrain,W[1:],W[0],30,5)
    plt.suptitle('Question 5: Training data and decision boundary')


    print('learning rate =',lrate)
    print('number of iterations =',I)
    print('weight vector =',W)
    print('bias term and weight vector')
    print(' from Question 4:',w0,w)
```

```
gd_logreg(1)




##########  QUESTION 6  ############

print('\n')
print('QUESTION 6.')
print('----------')

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


# Question 6(a)
# construct reduced data consisting of digits d1 and d2.
# all data sets are global variables.
def reduce(d1,d2):
    global Xtrain2, Ttrain2, Xval2, Tval2, Xtest2, Ttest2,
XtrainSmall, TtrainSmall
    # reduced training data
    idx = (Ttrain==d1) | (Ttrain==d2)    # index to digits d1 and d2
    Xtrain2 = Xtrain[idx]
    Ttrain2 = Ttrain[idx]
    # reduced validation data
    idx = (Tval==d1) | (Tval==d2)
    Xval2 = Xval[idx]
    Tval2 = Tval[idx]
    # reduced test data
    idx = (Ttest==d1) | (Ttest==d2)
    Xtest2 = Xtest[idx]
    Ttest2 = Ttest[idx]
    # small version of the reduced training set
    N = 2000
    XtrainSmall = Xtrain2[:N]
    TtrainSmall = Ttrain2[:N]


reduce(5,6)



# Question 6(b)
# display the first 16 digits of X in a 4x4 grid
def show16(X):
    X = np.reshape(X,[-1,28,28])
    plt.figure()
```

```
    for d in range(16):
        plt.subplot(4,4,d+1)
        d = d+1
        plt.imshow(X[d],cmap='Greys')
        plt.axis('off')

show16(Xtrain2)
plt.suptitle('Question 6(b): 16 MNIST training images')



# Question 6(c)
# determine the best value of K for KNN
def bestK():
    accTrain = []
    accVal = []
    Klist = range(1,20,2)    # list of K values under consideration
    accMax = 0    # the highest accuracy so far
    for K in Klist:    # test each value of K in turn
        clf = sklearn.neighbors.KNeighborsClassifier(n_neighbors=K)
        clf.fit(Xtrain2,Ttrain2)
        acc = clf.score(XtrainSmall,TtrainSmall)    # Training
accuracy
        accTrain.append(acc)
        acc = clf.score(Xval2,Tval2)    # Validation accuracy
        accVal.append(acc)
        if acc > accMax:    # Is the current K better than the
previous best?
            accMax = acc
            Kbest = K    # the best value of K so far
        print(acc)
    # plot training and validation error
    plt.figure()
    plt.plot(Klist,accTrain,c='b')
    plt.plot(Klist,accVal,c='r')
    plt.title('Question 6(c): Training and Validation Accuracy for
KNN, digits 5 and 6')
    plt.xlabel('Number of Neighbours, K')
    plt.ylabel('Accuracy')
    # compute test error
    clf = sklearn.neighbors.KNeighborsClassifier(Kbest)
    clf.fit(Xtrain2,Ttrain2)
    acc = clf.score(Xtest2,Ttest2)
    # print results
    print('Best K =',Kbest)
    print('Validation accuracy =',accMax)
    print('Test accuracy =',acc)


print('\nQuestion 6(c).')
```

```
print('--------------')
bestK()



# Question 6(d)
print('\nQuestion 6(d).')
print('--------------')
reduce(4,7)
show16(Xtrain2)
plt.suptitle('Question 6(d): 16 MNIST training images')
bestK()
plt.title('Question 6(d): Training and Validation Accuracy for KNN,
digits 4 and 7')
```

PRINTED OUTPUT

QUESTION 1.

Question 1(a):
```
[[0.5507979  0.70814782 0.29090474 0.51082761 0.89294695]
 [0.89629309 0.12558531 0.20724288 0.0514672  0.44080984]
 [0.02987621 0.45683322 0.64914405 0.27848728 0.6762549 ]
 [0.59086282 0.02398188 0.55885409 0.25925245 0.4151012 ]]
```

Question 1(b):
```
[[0.28352508]
 [0.69313792]
 [0.44045372]
 [0.15686774]]
```

Question 1(c):
```
[[0.5507979  0.70814782 0.29090474 0.51082761 0.89294695 0.89629309
  0.12558531 0.20724288 0.0514672  0.44080984]
 [0.02987621 0.45683322 0.64914405 0.27848728 0.6762549  0.59086282
  0.02398188 0.55885409 0.25925245 0.4151012 ]]
```

Question 1(d):
```
[[ 0.26727282  0.42462274  0.00737966  0.22730252  0.60942187]
 [ 0.20315517 -0.56755261 -0.48589504 -0.64167072 -0.25232807]
 [-0.41057751  0.01637951  0.20869033 -0.16196644  0.23580118]
 [ 0.43399508 -0.13288586  0.40198635  0.10238471  0.25823346]]
```

Question 1(e):
```
[0.28352508 0.69313792 0.44045372 0.15686774]
```

Question 1(f):
```
[[0.5507979  0.70814782 0.29090474 0.28352508 0.89294695]
 [0.89629309 0.12558531 0.20724288 0.69313792 0.44080984]
 [0.02987621 0.45683322 0.64914405 0.44045372 0.6762549 ]
 [0.59086282 0.02398188 0.55885409 0.15686774 0.4151012 ]]
```

Question 1(g):
```
[[ 0.57442982  0.42462274  0.00737966  0.22730252  0.60942187]
 [ 0.9003808  -0.56755261 -0.48589504 -0.64167072 -0.25232807]
 [ 1.08959777  0.01637951  0.20869033 -0.16196644  0.23580118]
 [ 0.71572183 -0.13288586  0.40198635  0.10238471  0.25823346]]
```

Question 1(h):
[[0.5507979 0.70814782 0.29090474 0.28352508 0.89294695]
 [0.89629309 0.12558531 0.20724288 0.69313792 0.44080984]
 [0.02987621 0.45683322 0.64914405 0.44045372 0.6762549]]

Question 1(i):
[[0.70814782 0.28352508]
 [0.12558531 0.69313792]
 [0.45683322 0.44045372]
 [0.02398188 0.15686774]]

Question 1(j):
[[-0.59638732 -0.34510242 -1.23475942 -1.26045469 -0.1132281]
 [-0.10948781 -2.07476999 -1.57386385 -0.36652628 -0.81914169]
 [-3.51069274 -0.78343689 -0.43210063 -0.81994991 -0.3911852]
 [-0.52617141 -3.73045663 -0.58186686 -1.85235226 -0.87923294]]

Question 1(k):
9.087621365532033

Question 1(l):
[0.89629309 0.70814782 0.64914405 0.69313792 0.89294695]

Question 1(m):
2.7263225002245983

Question 1(n):
[[1.57884629 -0.35284012 -0.187686   -0.3942709   0.26913377]
 [1.03478465 0.23371525 0.04918167 0.0088431   0.51378681]
 [1.46099184 -0.05772761 0.26157029 -0.11477974 0.42237427]
 [1.37914438 -0.28663212 -0.17972284 -0.4355982   0.14225624]
 [1.94377489 0.08489845 0.1003952 -0.14691625 0.69960743]]

Question 1(m):
[[2.22648013]]

QUESTION 2(c).

Execution time of matrix_poly(100): 1.1242170333862305
Execution time of equivalent NumPy operations: 0.0005371570587158203
Magnitude of B1-B2: 1.6402453908878153e-11

Execution time of matrix_poly(300): 30.018122911453247

Execution time of equivalent NumPy operations: 0.002783060073852539
Magnitude of B1-B2: 1.8391737004456768e-09

Execution time of matrix_poly(1000): 1084.6853008270264
Execution time of equivalent NumPy operations: 0.05500006675720215
Magnitude of B1-B2: 1.0240295446928734e-07

QUESTION 3.
---
a = 3.129414619191207
b = 4.719354385844762

Training error = 0.8557483910540564
Test error = 0.9608049758277348

QUESTION 4.

Question 4(a)
weight vector = [0.01694442 1.49601981 0.03738886]
bias term = -2.6250489555396475

Question 4(b)
accuracy1 = 0.856
accuracy2 = 0.856
accuracy1 - accuracy2 = 0.0

QUESTION 5.
---
learning rate = 1
number of iterations = 413
weight vector = [-2.638166   0.01740979 1.50266708 0.03800213]
bias term and weight vector
 from Question 4: -2.6250489555396475 [0.01694442 1.49601981 0.03738886]

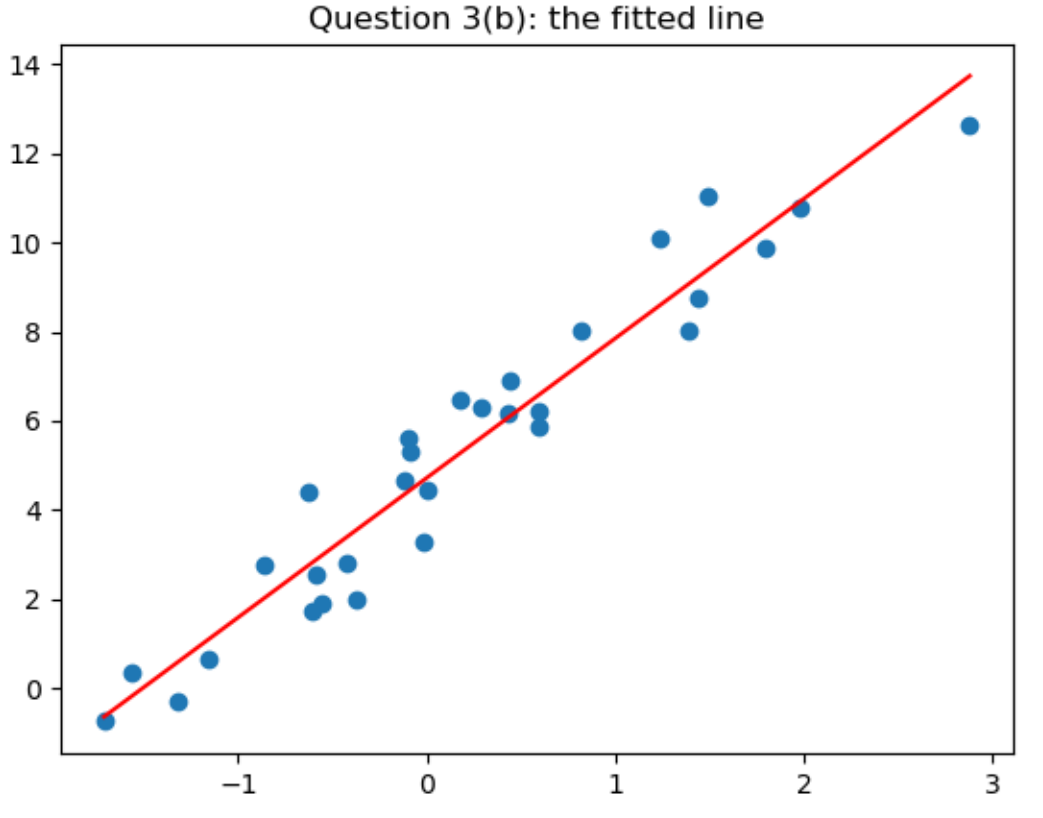
QUESTION 6.

Question 6(c).

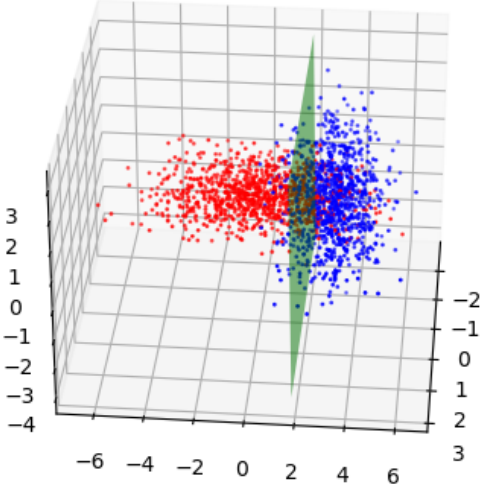
Best K = 3
Validation accuracy = 0.9905013192612138
Test accuracy = 0.9929729729729729

Question 6(d).

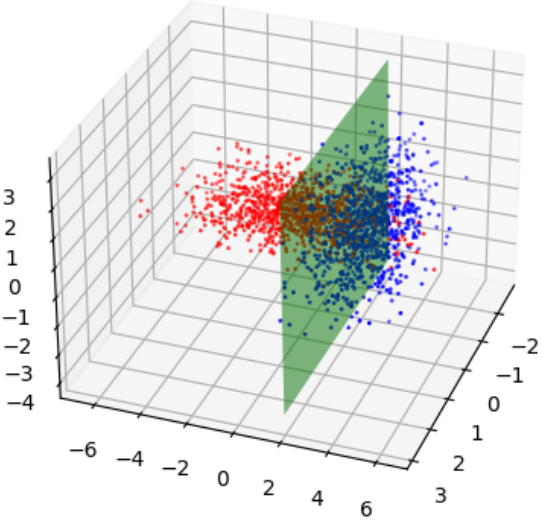
Best K = 9
Validation accuracy = 0.9975996159385502
Test accuracy = 0.9965174129353234

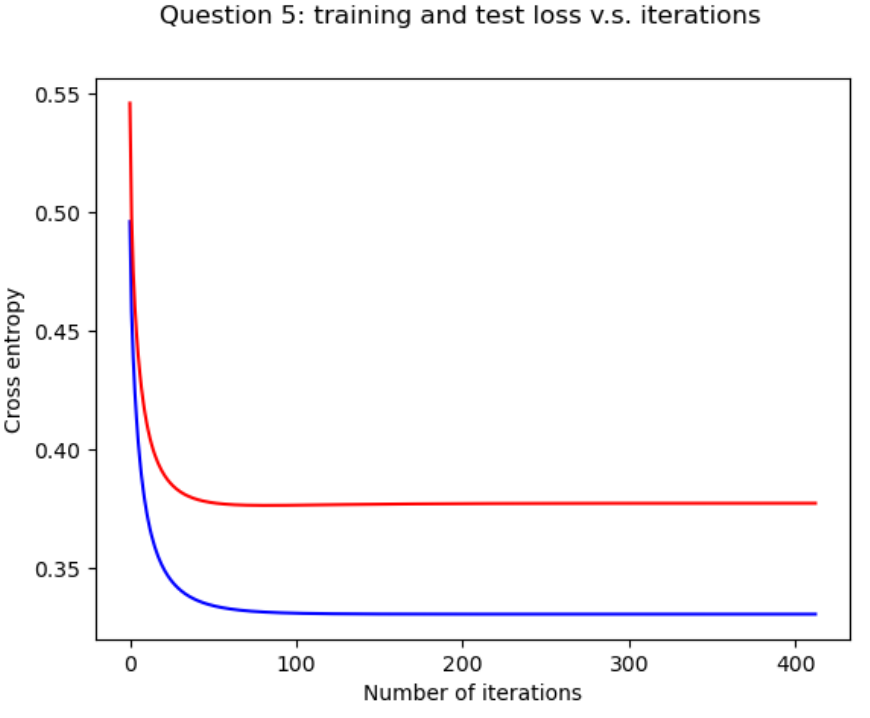

Question 3(b): the fitted line
14
12
10
8
6
4
2
0
−1
0
1
2
3

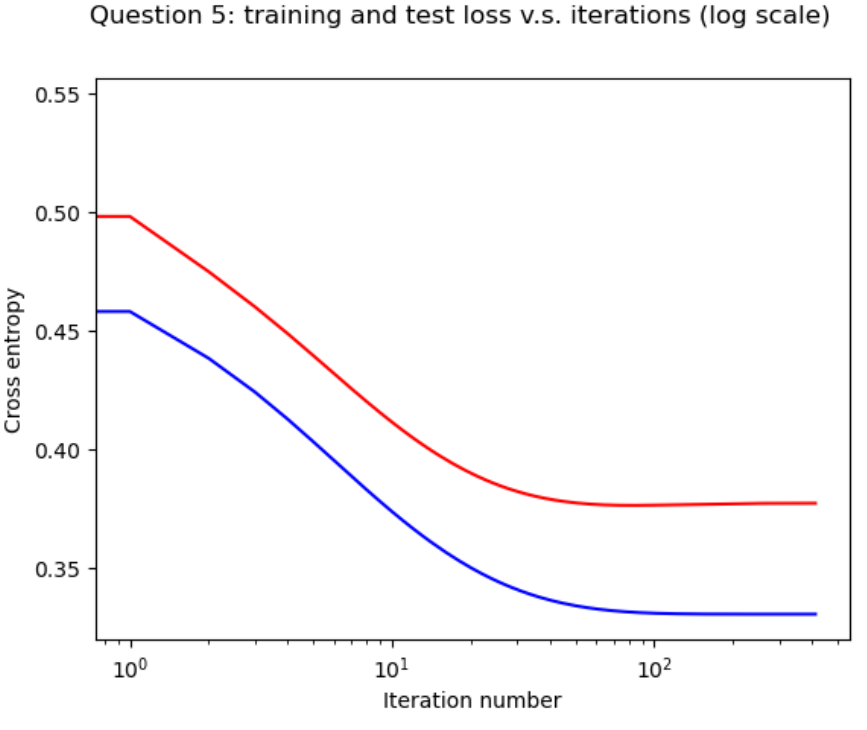
Question 4(c): Training data and decision boundary

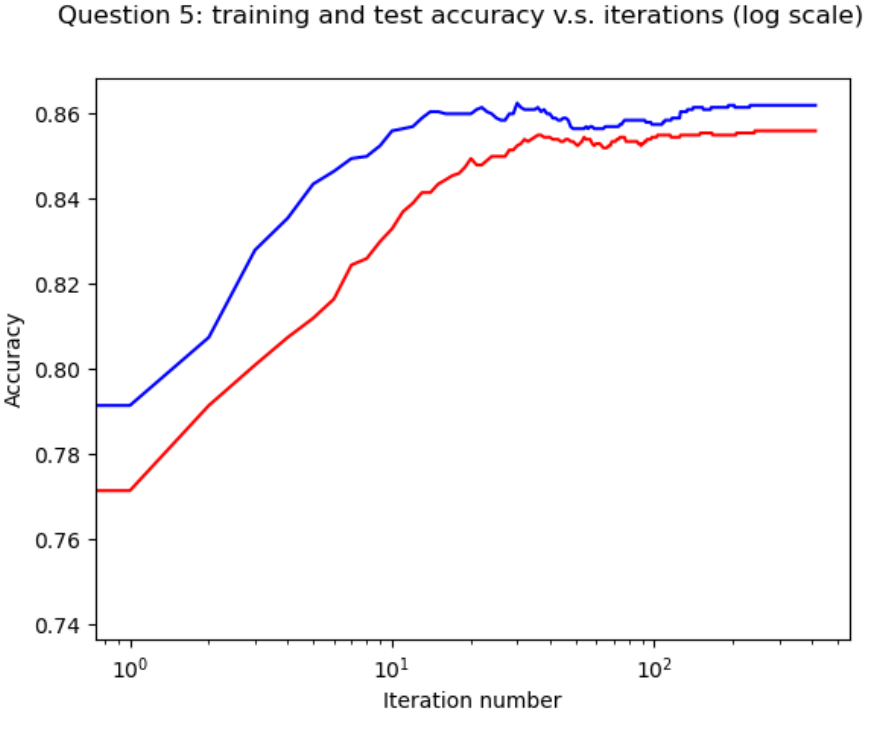


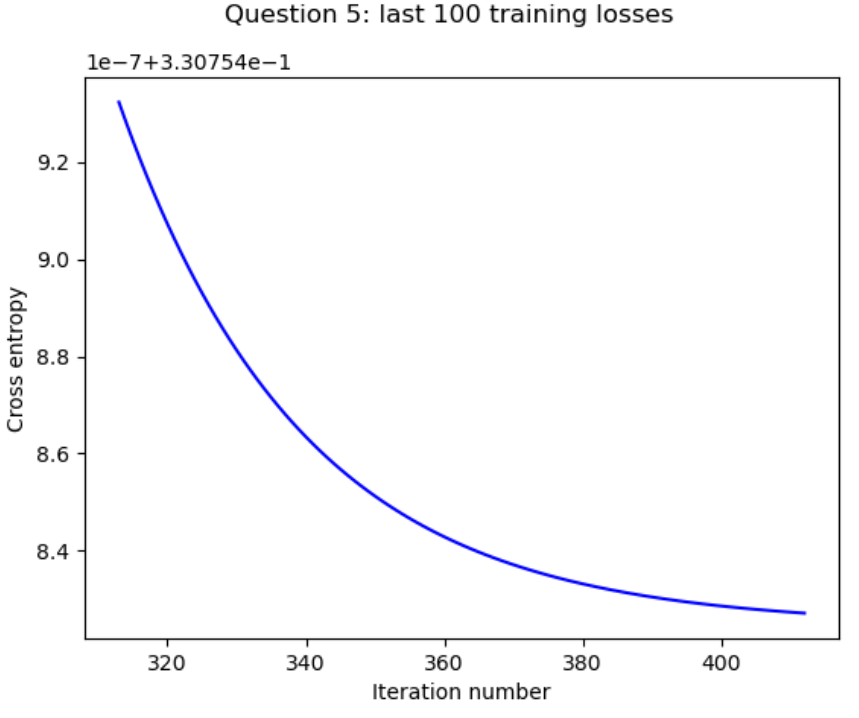
Question 4(d): Training data and decision boundary

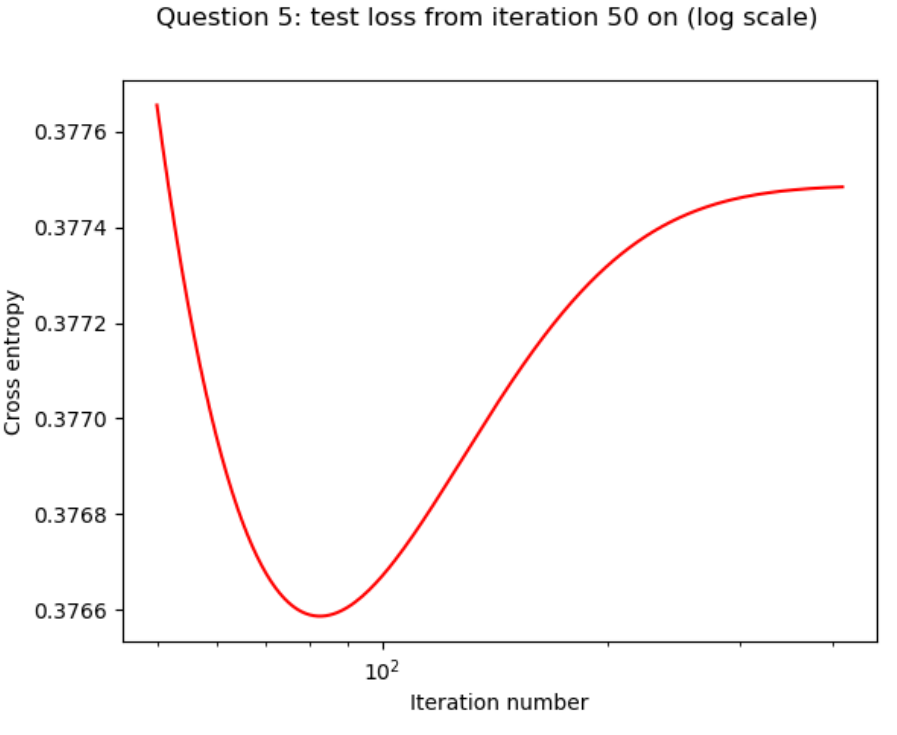



Question 5: training and test loss v.s. iterations
Cross entropy
0.55
0.50
0.45
0.40
0.35
0
100
200
300
400
Number of iterations

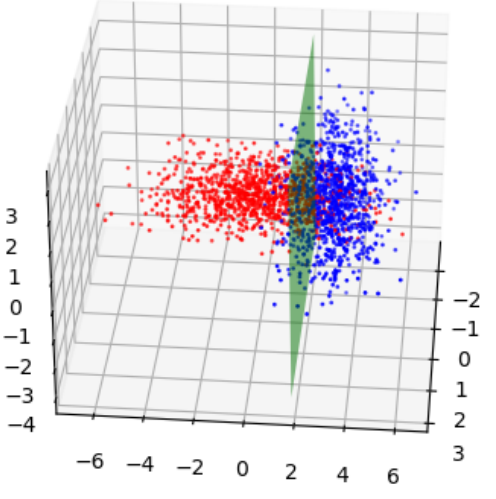

Question 5: training and test loss v.s. iterations (log scale)
0.55
0.50
0.45
0.40
0.35
Cross entropy
10^0
10^1
10^2
Iteration number


Question 5: training and test accuracy v.s. iterations (log scale)
Accuracy
0.86
0.84
0.82
0.80
0.78
0.76
0.74
10^0
10^1
10^2
Iteration number

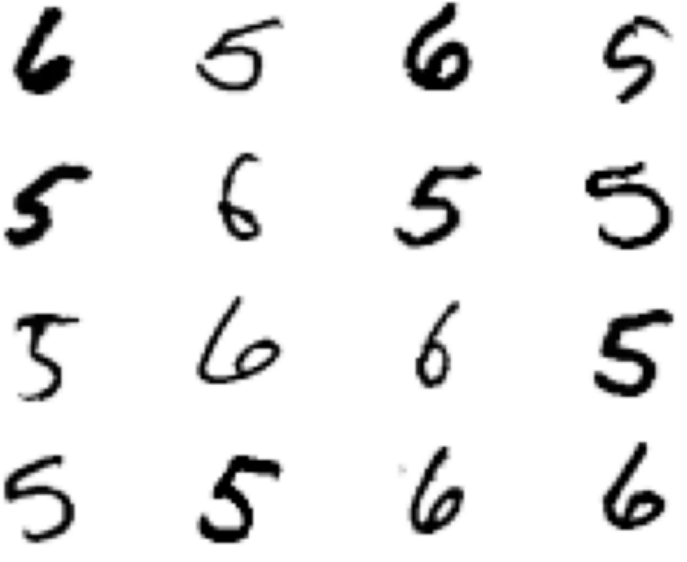

Question 5: last 100 training losses
1e−7+3.30754e−1
9.2
9.0
8.8
8.6
8.4
Cross entropy
320
340
360
380
400
Iteration number

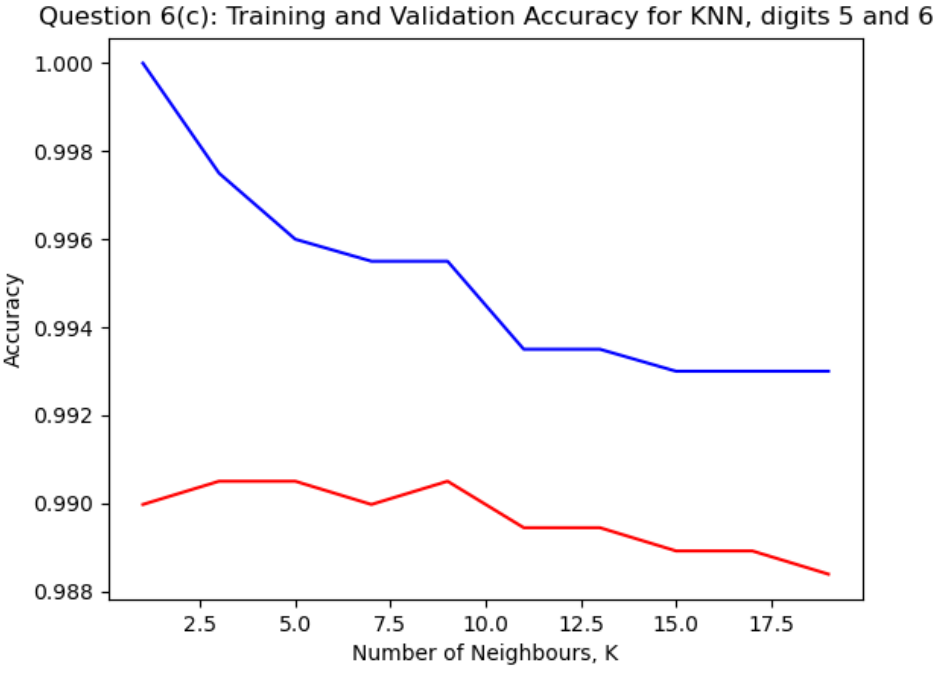

Question 5: test loss from iteration 50 on (log scale)
0.3776
0.3774
0.3772
0.3770
0.3768
0.3766
Cross entropy
10^2
Iteration number

Question 5: Training data and decision boundary

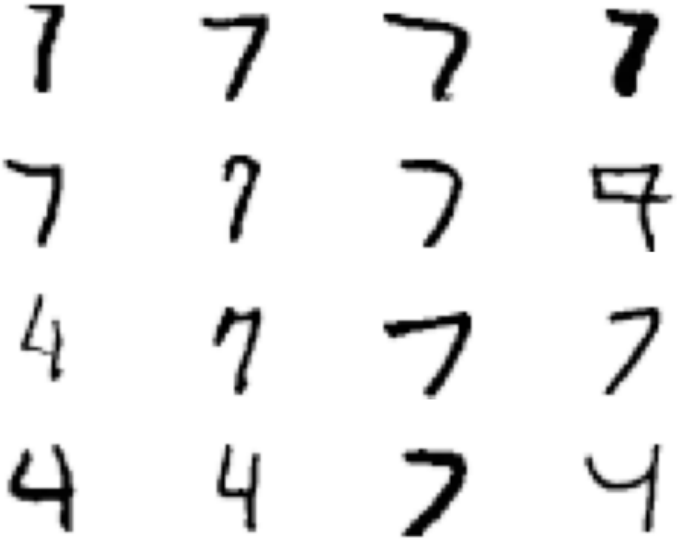


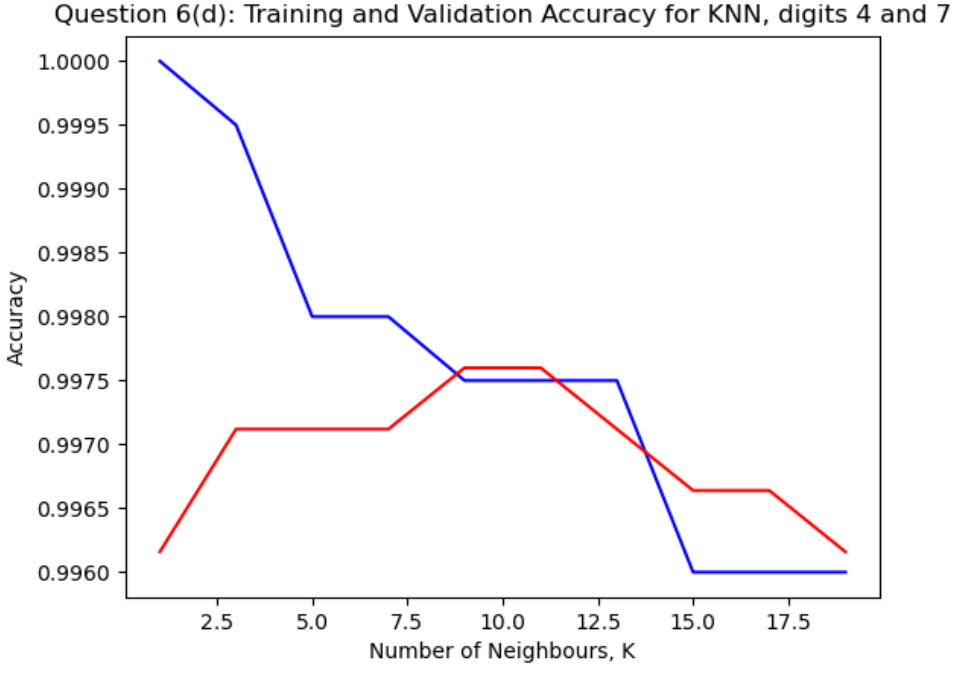
Question 6(b): 16 MNIST training images




Question 6(c): Training and Validation Accuracy for KNN, digits 5 and 6
1.000
0.998
0.996
0.994
0.992
0.990
0.988
Accuracy
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Number of Neighbours, K

Question 6(d): 16 MNIST training images




Question 6(d): Training and Validation Accuracy for KNN, digits 4 and 7
Accuracy
1.0000
0.9995
0.9990
0.9985
0.9980
0.9975
0.9970
0.9965
0.9960
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Number of Neighbours, K

University of Toronto Mississauga
CSC 311 - Introduction to Machine Learning

Cover sheet for Assignment 1

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: _____