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

Assignment 1

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 do not produce output 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.

No more questions will be added.

Tips on Scientific Programming in Python

If you haven't already done so, please read the NumPy tutorial on the course web page. 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)`. 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 part of a question prints any output, you should precede all code for that part with lines like the following:

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

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. (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 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. (15 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.

   (a) 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) 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) Execute `timing(100)`, `timing(300)` and `timing(1000)`, and hand in the printed results. Be sure it is clear which measurement each printed value refers to. *In each case, how many floating-point multiplications does* `matrix_poly` *perform?*[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.

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. *Linear Least-Squares Regression.* (?? points)

In this question, you will write a Python program to fit a simple linear function to data using least-squares regression.

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

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

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:

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

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 an array containing 1000 test points. 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.

Figure 1:

(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 in red. To plot the fitted line, draw a line segment between two points on the line. This line segment should extend from the smallest value of x in the training set to the largest. You will need to compute the value of y at these two endpoints. 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.

(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. (?? points total) *Binary Logistic Regression.*

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.

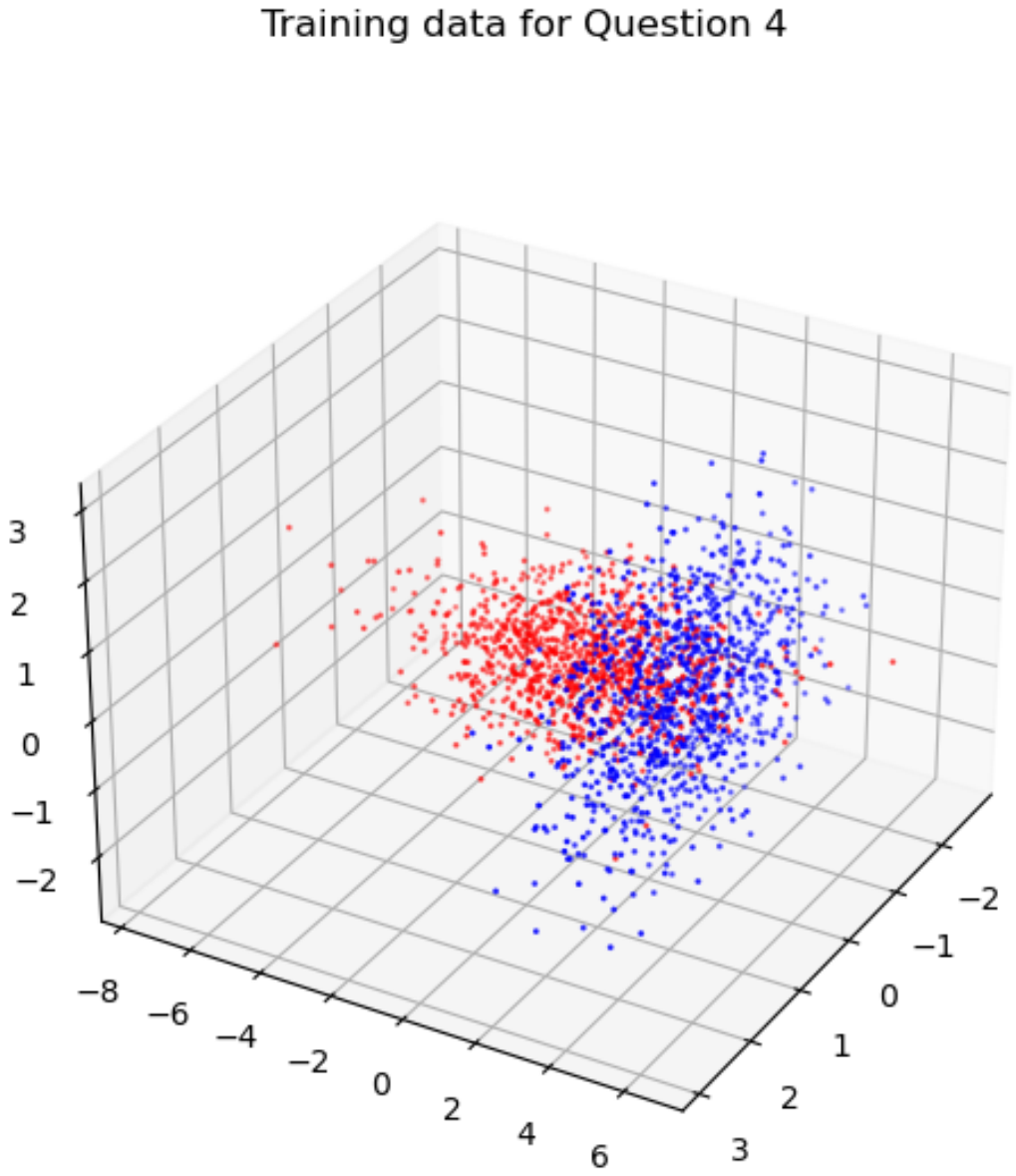
You will also need to download the file `bonnerlib3D.py.zip` from the course web site. It contains functions for displaying 3-dimensional data and decision boundaries. To fully utilize these functions and to animate your data, you will need to put the backend of your Spyder environment into Qt mode.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

Use the function `plot_data` in `bonnerlib3D` to display and view the data from different angles in 3D. The function `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 `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.

(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.

Figure 2:



(b) (? 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) 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.

(d) 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) Slide number 61 of Lecture 3 (Linear Classification) shows the following result 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} elments 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 both sides of Equation (6) evaluate to vectors, so you are being asked to show 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. More precisely, you must show that Equation (7) is equivalent to Equation (5). Here, the notation $[V]_j$ refers element j of vector V. Note that in some cases we don't need the square brackets. For example, $[y]_j = y_j$. 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 $[y]_j = y_j$). Others should be justified.

(f) Prove the equation for the logistic cross entropy in the third bullet on slide number 58 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.)

5. *Gradient Descent.*

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, and all other code should be vectorized. 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) 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) 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) 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 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.

(d) Perform weight updates until the cross entropy on the training data changes by less that 10^{-10} between two successive updates.

[2]Recall that accuracy is the average number of classification errors.

(e) 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 similae to your weight vector here, with 1 or 2 significant digits of similarity in each weight.

(f) In a single figure, plot the list of training cross entropies (in blue), and the list of testing 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*.

(g) If everything is working correctly, the previous figure should show the cross entropy decreaing very rapidly and then flattening out. 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 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) 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) 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) 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) 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*.

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 curves 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.

6. *Nearest Neighbours.*

   In this question, you will use K nearest neighbours (KNN) to classify images of hand-written 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:

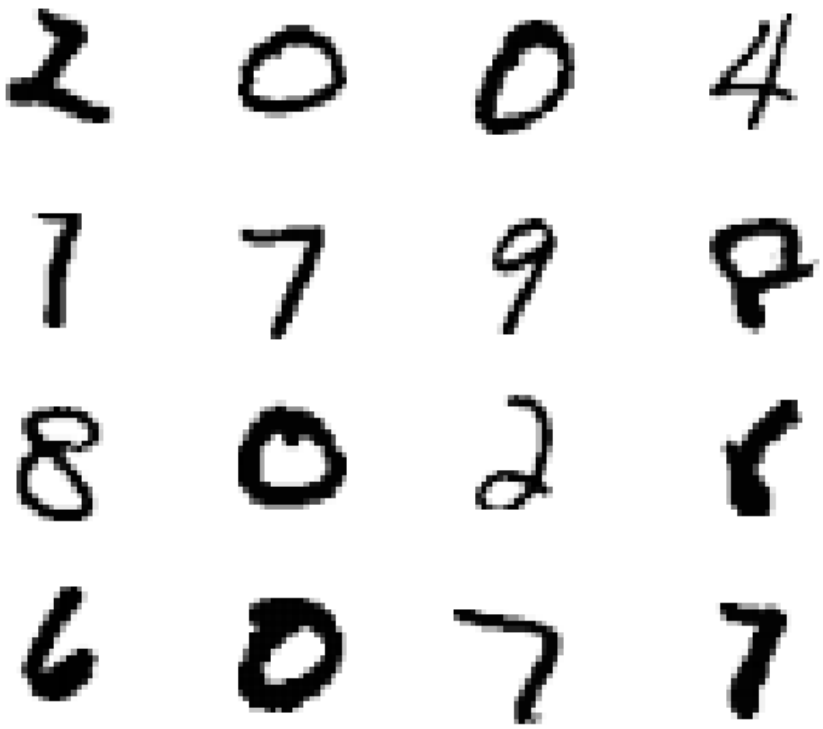
   ```
   with open('mnistTVT.pickle','rb') as f:
       Xtrain,Ttrain,Xval,Tval,Xtest,Ttest = pickle.load(f)
   ```

   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.)

Figure 3:
A sample of MNIST data



What to do: Write Python programs to carry out the tasks below. Except where explicity allowed, do not use any loops. All code should be vectorized whenever possible.

(a) 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 Xtrain, then i should also come before j in the reduced version of Xtrain. 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).

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) 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.subpot`. 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.

(c) 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`. 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 testing 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, `Error`. If everything is working properly, you should find that the training accuracy generally decreases as K increases, whereas the test accuracy increases slightly at first, then peaks, then decreases. Also, for every value of K, the training accuracy should be greater than the test 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) 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) 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.

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

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

?? 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: _____