University of Toronto Mississauga
Department of Mathematical and Computational Sciences
CSC 311 - Introduction to Machine Learning, Fall 2021

Assignment 2

Due date: Monday November 8, 11:59pm.
No late assignments will be accepted.

As in all work in this course, 20% of your grade is for quality of presentation, including the use of good English, properly commented and easy-to-understand programs, and clear proofs. In general, short, simple answers are worth more than long, complicated ones. Unless stated otherwise, all answers should be justified. The TA has a limited amount of time to devote to each assignment, so what you hand in should be legible, 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.

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.

Read the file **important_guidelines.pdf**
in the Files area on Quercus under Assignments.

More questions will be added shortly.

1. *Multi-class Linear Classification.* (? points total) This question is a warm-up and preparation for Question 2, in which you will implement a learning program for linear classification. First, you will use programs in `scikit-learn` to train a linear classifier on 2-dimensional (2D) cluster data. The results produced by this classifier should be similar to that of your own implementation in Question 2, so you can check the correctness of your implementation. Second, you will derive some vectorized equations and write some helper functions that you will use in Question 2.

   The data you will use to train your classifier (both here and in Question 2) consists of three classes (or clusters), which we denote 0, 1 and 2. The data is in the file `cluster_data.pickle.zip` in the Files area on Quercus under Assignment 2. Download and uncompress this file. You can then read the file with the following Python command:

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

   The variables `Xtrain` and `Ttrain` will now contain training data. Specifically, `Xtrain` is a 2000×2 numpy array, where each row represents a 2D input point; and `Ttrain` is a vector of length 2000, where each element represents a target value, *i.e.*, a class. Likewise, `Xtest` and `Ttest` contain test data. The training data is illustrated in the scatter plot in Figure 1. Each dot in the figure corresponds to an input point in `Xtrain`. The figure shows three clusters, coloured red, blue and green, which correspond to the target values 0, 1 and 2, respectively. *e.g.*, all input points with a target value of 2 are coloured green in Figure 1. Your main task is to write a learning program that trains a linear classifier to predict which cluster a test point is in.

   To check your classifier, you will compare it to another classifier that is presumably correct. You will train this second classifier using the Python class `LogisticRegression` in `sklearn.linear_model`, which generates a Python object that does linear classification. Use the following code to do this:

```
import sklearn.linear_model as lin
clf = lin.LogisticRegression(multi_class='multinomial',solver='lbfgs')
clf.fit(Xtrain,Ttrain)
```

   Here, the keyword arguments `multi_class='multinomial'` and `solver='lbfgs'` specify multi-class classification with cross-entropy loss. The `fit` method then trains a classifier on the training data. This classifier should give similar (if not identical) results to the one you implement. You should read about the `LogisticRegression` class, its attributres and methods in the `sklearn` user guide. (Simply google, *skearn LogisticRegression*.)

   In addition, download the file `bonnerlib2D.py.zip` from the course web site and import it into your program with the statement `import bonnerlib2D as bl2d`. It

Figure 1:

contains functions for displaying cluster data and 2-dimensional decision boundaries. Note that you can add items (like a title) to a figure after calling these functions. You should not modify these functions in any way.

Unless specified otherwise, in answering the questions below, do not use any Python loops. Instead, all code should be vectorized.

(a) *Training with sklearn.* (? points) Retrieve the training and test data from Quercus. Train a linear classifier using `sklearn` as described above. Using the `score` method of the `LogisticRegression` class, compute and print out the accuracy of the classifier on the training data and on the test data. (Accuracy is the proportion of predictions that are correct.) This can easily be implemented as a Python script with at most 8 lines of highly readible code (not counting comments) with no loops. (Save the two scores for use in Question 2.)

(b) *Displaying the results.* (? points) Generate a plot of the training data with the decision boundaries of the classifier superimposed on top. You can do this using functions in `bonnerlib2D`. First, use `plot_data` to plot the training data. Then use `boundaries` to draw the decision boundaries. (Be sure to call `plot_data` before you call `bounaries`, to set the limits on the axes.) If you have done everything properly, the decision boundaries should appear as three black line segments meeting at a common point between the three clusters. Title the figure, *Question 1(b): decision boundaries for linear classification.* This can easily be implemented as a Python script with at most 4 lines of highly readible code (not counting comments) that extends the script from part (a) and has no loops.

(c) *Gradient equations* (? points) Slide 8 of Lecture 4 gives the following equations for linear classification with cross-entropy loss:

$$\begin{aligned} z &= Wx + b \\ y &= \text{softmax}(z) \\ \mathcal{L}_{CE} &= -t^T \log(y) \end{aligned}$$

Here, (x, t) is a single data point, where x is the input and t is the target value. Note z, y, x, t and b are column vectors, and W is a weight matrix. Recall that t is a binary vector representing a one-hot encoding of class membership.

These equations are vectorized versions of the following:

$$z_n = \sum_m W_{nm} x_m + b_n \tag{1}$$

$$y_n = \exp z_n / \sum_m \exp z_m \tag{2}$$

$$\mathcal{L}_{CE} = -\sum_n t_n \log y_n \tag{3}$$

To these, we add the following equation for total cost:

$$\mathcal{J} = \frac{1}{N} \sum_i \mathcal{L}_{CE}^{(i)} \tag{4}$$

Here, the sum is over all training points, N is the number of training points, and $\mathcal{L}_{CE}^{(i)}$ is the cross entropy loss on the i^{th} training point, $(x^{(i)}, t^{(i)})$.
The goal of learning is to find values of W and b that fit the data well, that is, which minimize $\mathcal{J}$. You will do this later using gradient descent, so you will need equations for the gradient of $\mathcal{J}$ wrt W and b. You will derive a vectorized version of these equations in several steps in the questions below. Before attempting these questions be sure to study the final section — *on proving theorems* — in the *important_guidelines* on Quercus. Be sure to justify every step of your proofs. Use only the numbered equations above. You may also use the following fact about softmax:

$$\frac{\partial y_n}{\partial z_m} = y_n(\delta_{nm} - y_m) \tag{5}$$

where δ_{nm} is the Kronecker delta (see the *guidelines*). All proofs should be typed and easy to understand.

i. Prove that

$$\frac{\partial \mathcal{L}_{CE}}{\partial z_j} = y_j - t_j \tag{6}$$

Hint: recall that t is a one-hot encoding.

ii. Use the results above to prove that

$$\frac{\partial \mathcal{L}_{CE}}{\partial W_{jk}} = (y_j - t_j)x_k \tag{7}$$

Be sure to justify why you use the univariate or multi-variate chain rule. (Most points are for this justification.)

iii. Use the results above to prove the following vectorized equation:

$$\frac{\partial \mathcal{J}}{\partial W} = \frac{1}{N}(Y - T)^T X \tag{8}$$

Here, X, Y and T are data matrices that contain one row for each training point. Specifically, their i^{th} rows are the vectors $x^{(i)}$, $y^{(i)}$ and $t^{(i)}$ (transposed), respectively. Formally, this means that $X_{ij} = x_j^{(i)}$, that is, the ij^{th} component of matrix X is the j^{th} element of the i^{th} input vector. Likewise for the matrices Y and T. Note that both sides of Equation (8) are matrices with the same dimensions as W.

In what follows, it will sometimes be convenient to assume that the bias vector, b, has been absorbed into the weight matrix, W, and that the data matrix, X, has been augmented with a column of 1s, in the usual way. In this case, the bias vector can be ignored and the gradient vector $\partial\mathcal{J}/\partial b$ does not need to be computed. At other times, it will be convenient to assume that the weight matrix and bias vector are separate. Each question below will make it clear which assumption you should use.

(d) Using the notation of part (c), let Z be the matrix whose i^{th} row is the vector $z^{(i)}$ (transposed), that is, the z vector that results from input $x^{(i)}$. Write down a vectorized equation for Z, one that you can translate directly into vectorized code to efficiently compute Z in your program in Question 2. For convenience, assume that the bias term has been absorbed into the weight matrix and can be ignored, as discussed above.

(e) Define a Python function `predict(X,W,b)` that makes class predictions. Here, `W` is a weight matrix, `b` is a bias vector, and `X` is a data matrix, as in part (c). The function should return a class prediction for each row vector in `X`. That is, it should return an integer between 0 and $J-1$, inclusive, where J is the number of classes. Thus, if `X` has N rows, then the function should return a vector of N integers. You may find the Numpy function `argmax` useful.

Test your `predict` function by comparing it to the `predict` method of the `LogisticRegession` class in `sklearn`. In particular, let `clf` be the classifier you trained in part (a). Then, the expression `clf.predict(Xtest)` will return the class predictions of the classifier on the test data. In addition, the following two statements will retrieve the fitted weight matrix and bias vector of the classifier:

```
W = clf.coef_
b = clf.intercept_
```

(Here, `coef_` and `intercept_` are called *attributes* of the `LogisticRegression` class, because they simply retrieve stored values. In contrast, `predict` is called a *method*, because it performs computation.)

Let Y_1 be the vector of class predictions returned by `clf` on the test data, and let Y_2 be the vector of class predictions returned by your `predict` program on the test data using the same weight matrix and bias vector as `clf`. Y_1 and Y_2 should be identical. Thus, $Y_1 - Y_2$ should be a vector of all zeros. To prove that your function is correct, compute and print out the squared magnitude[1] of $Y_1 - Y_2$. It should be 0.

The `predict` function and the code for testing it should be vectorized, should not have any loops, and should work for any number of classes. You should use this function in Question 2. The function can easily be defined and tested in at most 11 lines of highly-readible Python code, not counting comments.

(f) *One-hot encodings.* Write a Python function `one_hot(Tint)` that converts integer target values to one-hot encodings. Here, `Tint` is a vector of non-negative integers representing classes (target values). The function should return a binary matrix, `Thot`, where each row of `Thot` is a one-hot encoding of the corresponding entry in `Tint`. Thus, `Thot` has shape $[N, J]$, where N is the number of entries in `Tint` and J is the number of classes. You should use the maximum value in `Tint` to determine the number of classes. Your function should be vectorized, should not contain any loops, and should work for any number of classes. You should use this function in Question 2.

[1] If v is a vector, then its squared magnitude in $\sum_i v_i^2$.

Test your function by computing the value of `one_hot([0,1,2,3,0,1,2,3])`. The result should be a binary matrix of shape $[8, 4]$ that looks like two identity matrices stacked one above the other. Do *not* hand this in. Instead, print and hand in the value of `one_hot([4,3,2,3,2,1,2,1,0])`.

The `one_hot` function and the code for testing it should be vectorized and should not contain any loops. The function can easily be defined and tested in at most 11 lines of highly-readible Python code, not counting comments.

Hint: First, construct a vector `C` of shape $[1, J]$ where `C[0,j]` = `j`. Also, change the shape of `Tint` to $[N, 1]$. Then the Python expression `Tint==C` generates a boolean matrix of shape $[N, J]$. If we call this matrix `Thot`, then `Thot[n,j]` = `True` *iff* `Tint[n,0]` = `C[0,j]`, that is, *iff* `Tint[n,0])` = `j`. This is a one-hot encoding. Use numpy casting to convert the boolean values to numbers. (*i.e.*, to convert *True* and *False* to 1 and 0.) This is an example of *broadcasting.*

2. *Implementation.* (? points) In this question, you will implement multi-class classification in two ways, one based on (batch) gradient descent, and one based on stochastic gradient descent. Use the results from Question 1(c) to vectorize the weight updates. Your programs should work for input data of any dimension, but you will test them on the 2D cluster data of Question 1. You should assume that the training and test data are stored in global variables called `Xtrain`, `Ttrain`, `Xtest` and `Ttest`, respectively. Your code should be vectorized and should use no more than the number of loops specified below. You can (and should) use helper functions, but they may not contain any loops. Unless specified otherwise, you should not use any functions from `sklearn`. The point is to implement linear classification yourself using simple linear-algebra operations.

   (a) *Gradient descent.* Implement (batch) gradient descent for multi-class linear classification with a cross-entropy loss function. In particular, define a Python function `GDlinear(I,lrate)` that performs `I` iterations of gradient descent where `lrate` is the learning rate. The function should use only one loop, but no nested loops, and all other code should be vectorized. In addition to gradient descent, the `GDlinear` function should do the following:

      i. (0 points) The first statement should be `numpy.random.seed(7)`. This ensures that everyone will use the same randomly-initialized weight vector and get the same final answers (if their programs work correctly).

      ii. Print the learning rate. That is, the second line of your function should be `print('learning rate =',lrate)`.

      iii. (? points) Extend the training and testing data matrices with a column of 1s, so that the bias term can be treated as just another weight, in the usual way. The 1s should be in column 0 of each extended data matrix.

      iv. (? points) Convert the target values from integers to one-hot vectors.

      v. (? point) Initialize the weight matrix (including the bias term) by using `randn` in `numpy.random` to generate a random matrix, and then dividing this matrix

by 10,000. 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.)

vi. (? points) Recall that gradient descent is an iterative algorithm that performs weight updates at each iteration. At each iteration your function should also compute the average cross entropies 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.

vii. (? points) In a single figure, plot the list of training cross entropies (in blue), and the list of test cross entropies (in red). Use the function `semilogx` in `matplotlib.pyplot` to plot the lists, which puts a log scale on the horizontal axis. Title the figure, *Question 2(a): 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 cross entropy should decrease smoothly from left to right, bottoming-out and becoming flat on the right side. (There may also be a short, flat segment at the very left.)

viii. (? points) In a single figure, plot the list of training accuracies (in blue), and the list of testing accuracies (in red). Put a log scale on the horizontal axis. Title the figure, *Question 2(a): Training and test accuracy v.s. iterations*. Label the vertical axis *Accuracy* and the horizontal axis *Iteration number*. Both accuracies should increase somewhat jaggedly from left to right, then decreases more steeply, and finally flatten out.

ix. (? points) Take a closer look at the test cross entropies by plotting all but the first 50 of them in a single plot by themselves as a red curve (without any training cross entropies). Use a log scale on the horizontal axis. Title the figure, *Question 2(a): test loss from iteration 50 on*. 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.

x. (? points) Take a closer look at the training cross entropies by plotting all but the first 50 of them in a single plot by themselves as a blue curve (without any test cross entropies). Use a log scale on the horizontal axis. Title the figure, *Question 2(a): training loss from iteration 50 on*. Label the axes as before. If everything is working correctly, you should observe that the training cross entropy decreases smoothly and flattens out, but never increases.

xi. (? points) Print the training accuracy after the final iteration of gradient descent. Print the training accuracy from Question 1(a). Print their difference. For full marks, the difference should be 0.

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

xii. (? points) Print the test accuracy after the final iteration of gradient descent. Print the test accuracy from Question 1(a). Print their difference. For full marks, the difference should be 0.

xiii. (? points) As in Question 1(b), generate a plot of the training data with the decision boundaries superimposed on top. You can do this with the functions `plot_data` and `boundaries2` in the latest version of `bonnerlib2D` on Quercus. You will also need your `predict` function from Question 1(e). In particular, you will need to pass it as an argument to the function `boundaries2`. Title the figure, *Question 2(a): decision boundaries for linear classification.* This plot should look exactly the same as the one you generated in Question 1(b). (Note that you will have to figure out what to do about the bias vector that the `predict` function expects. There is more than one solution to this problem.)

Use the function `GDlinear` to perform 10,000 iterations of gradient descent on the 2D cluster data in Question 1. Do this for five learning rates: 10, 1, 0.1, 0.001 and 0.00001. Do *not* hand in all the output from all of these runs. Instead, choose the run with the largest learning rate that produces smooth curves of cross entropy, and which produces the same results as Question 1(a) and (b), *i.e.*, the same training and test accuracies, and the same decision boundaries. Hand in all the output from this run. (Be sure to include the print out of the learning rate.)

In addition, for each learning rate, hand in the curve of training and test cross entropy (the curves generated in item vii, above). Hand them in in order of decreasing learning rate (so the curve generated with a learning rate of 10 is first). You should find that some of these curves behave as expected, that is, as described in the items above. However, others do not. For instance, some are not smooth and are extrememly jagged, while others are smooth but do not flatten out.

(b) Explain the various curves you generated in the last paragraph above. In particular, why are some curves extremely jagged, and why do some smooth curves not flatten out?

(c) Explain the difference in the graphs in items xi and x of part (a). In particular, why does test error increase during learning, while training error does not?

(d) *Stochastic gradient descent.* Modify your program in part (a) to perform stochastic gradient descent with mini-batches. That is, instead of computing the gradient of the cost function on the entire training set at once, compute the gradient on a small, random subset of the training data (called a mini-batch), perform weight updates, and then move on to the next mini-batch, and so on.

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

sweep of the training data is called an epoch. Program comments should clearly indicate where an epoch begins and where mini-batches are created.

In addition, except for an initial "burn in" period, you should decrease the learning rate before each epoch. In particular, for the first κ epochs, the learning rate should be held constant. We call this the initial learning rate, λ_0. After that, the learning rate in epoch i should be $\lambda = \lambda_0/(1+k)$ where $k = \alpha(i-\kappa)$. Here, $\alpha > 0$ is the *decay rate*, which determines how fast the learning rate decreases. If the decay rate is 0, then stochastic gradient descent will not converge to a minimum, but will hop around near the minimum.

Detailed specifications: You should implement the ideas above as a Python function called `SGDlinear(I,batch_size,lrate0,alpha,kappa)` that performs `I` epochs of stochastic gradient descent, where `batch_size` is the number of training points in each mini batch, `lrate0` is the initial learning rate, λ_0, `alpha` is the decay rate, α, and `kappa` is the burn-in period, κ. The function should have exactly two loops, one nested inside the other. All other code should be vectorized.

The function should first print out the batch size, initial learning rate, decay rate and burn-in period. It should also carry out the same the pre-processing as in `GDlinear`, such as setting the random seed to 7, extending the data matrices with a column of 1s, initializing the weight matrix randomly, and converting the target values to one-hot vectors.

The function should then execute `I` epochs of stochastic gradient descent. At the beginning of each epoch, you should use the function `shuffle` in `sklearn.utils` to randomly shuffle the training data. Note that both the input vectors and target values must be shuffled the same way. You should then sweep through the shuffled training data, updating the weight matrix one mini-batch at time, as described above. After each epoch, the program should record the average cross entropy and accuracy on the training data and on the test data, as in `GDlinear`.

After the `I` epochs have completed, the function should generate all the same output as `GDlinear`, that is, the output described in items vii to xiii of part (a), with figure titles modified appropriately. The main difference you should observe is that many of the graphs are now much more jagged.

What output to hand in: Use the function `SGDlinear` to perform 500 epochs of stochastic gradient descent on the 2D cluster data in Question 1 using a batch size of 30. Experiment with different learning rates, decay rates and burn-in times. Choose a run that produces the same results as Question 1(a) and (b), *i.e.*, the same training and test accuracies and the same decision boundaries. In particular, your program should print out a difference of 0 for the training accuracies and for the test accuracies. In addition, all the graphs should be flat during the last 100 epochs, which indicates convergence. Hand in all the output (both printed and graphical) produced by your function during this run. (Do not hand in output from any other runs.)

The graphs you hand in should look similar to (but different from) those in Figures 2 to 5 at the end of this assignment, which are based on a similar (but different) data set. Notice that most of the graphs are jagged most of the time, but they are flat at the right-hand end, for about the last 200 epochs. Your graphs should be flat for at least the last 100 epochs. It is not hard to get the graphs of cross entropy to be flat during the last 100 epochs. It is much harder to get the graphs of accuracy to be flat, but for full marks, they must be flat.

(e) As can be seen in Figures 2 to 5, there is a rapid transition at the right-hand end of the graphs from jagged to flat. Explain why this is.

3. *Probabilistic Generative Classifiers.* (? points) This question focuses on Gaussian Discriminant Analysis, often called Quadratic Discriminant Analysis, or QDA. Gaussian Naive Bayes is a special case of QDA. You will be running programs in `sklearn`, implementing programs of your own, and proving equations needed before implementation can begin. You will run your programs on the data used in Question 1, but they should work for data with any number of classes. Do not use any functions from `sklearn` other than those specified. Before attempting the proofs, be sure to study the final section — *on proving theorems* — in the *important guidelines* on Quercus. Be sure to justify every step of your proofs.

(a) *QDA in sklearn.* (? points) Repeat Question 1 (a) and (b) using the Python class `QuadraticDiscriminantAnalysis` in `sklearn.discriminant_analysis` to define the classifier. Use the keyword argument `store_covariance=True`, to store the covariance matrices for each class. Modify the figure titles appropriatey. You should find that the accuracy (both training and testing) is about 0.89

(b) *Gaussian Naive Bayes.* (? points) Repeat Questions 1(a) and (b) using the Python class `GaussianNB` in `sklearn.naive_bayes` to define the classifier.

(c) (? points) You should find that the accuracy in part (b) is lower than in part (a), and that the decision boundaries in part (b) are much more circular. Explain both of these results.

(d) (? points) *Mean Vectors: theory.* On slide 49 of Lecture 5/6, the second last equation gives the Maximum Likelihood estimate of the mean vector for each class in QDA. Starting from this equation, prove the following:

$$\mu_{ki} \;=\; \sum_n T_{nk} X_{ni} / N_k \tag{9}$$

where $N_k = \sum_n T_{nk}$. Here μ is a matrix whose k^{th} row is the estimated mean vector for class k, X is a data matrix of input vectors, and T is a one-hot matrix of class labels. That is, using the notation of slide 49, X_{ni} is the i^{th} feature of the input vector $\mathbf{x}^{(n)}$, and μ_{ki} is the i^{th} feature of the mean vector $\hat{\mu}_k$. In addition, $T_{nk} = I\{t^{(n)} = k\}$. You may use each of these facts in your proof.

(e) (? points) *Covariance Matrices: theory.* On slide 49 of Lecture 5/6, the last equation gives the Maximum Likelihood estimate of the covariance matrix for each class in QDA. Starting from this equation, prove the following:

$$\Sigma_{kij} \;=\; \sum_n T_{nk}(X_{ni} - \mu_{ki})(X_{nj} - \mu_{kj})/N_k \tag{10}$$

where we are using the notation from part (d). Here, Σ_{kij} is the ij^{th} entry in the covariance matrix for class k. That is, using the notation of slide 49, Σ_{kij} is the ij^{th} entry of $\hat{\Sigma}_k$.

(f) (? points) *Mean Vectors: implementation.* Write a Python function `EstMean(X,T)` that estimates the mean vector for each class in QDA from the data in `X` and `T`. Here, `X` is a data matrix of input vectors and `T` is a one-hot matrix of class labels. Your function should return a matrix, `mu`, that contains the mean vectors for all the classes.[3] Specifically, `mu[k]` should be the estimated mean vector for class `k`. In defining your function, do not use any loops, and do not use any functions from `sklearn`. Your code should be completely vectorized and should work for any number of classes. Hint: express Equation (9) in terms of matrix multiplication.

Test your function on the training data of Question 1 by comparing its estimate of the mean vectors to those produced by the `sklearn` classifier in part (a).[4] In particular, if μ is the matrix of mean vectors returned by your function, and μ^{sk} is the matrix returned by the `sklearn` classifier, then you should find that $\mu = \mu^{sk}$ almost exactly. To demonstrate this, print out the total squared difference in the two matrices. That is, print out $\sum_{ki}(\mu_{ki} - \mu^{sk}_{ki})^2$. It should be less than 10^{-25}.

(g) (? points) *Covariance Matrices: implementation.* Write a Python function `EstCov(X,T)` that estimates the covariance matrix for each class in QDA from the data in `X` and `T`. As above, `X` is a data matrix of input vectors, and `T` is a one-hot matrix of class labels. Your function should return an array, `Sigma`, of rank 3 that contains the covariance matrices for all the classes.[5] Specifically, `Sigma[k]` should be the covariance matrix for class `k`. In defining your function, do not use any loops, and do not use any functions from `sklearn`. Your code should be completely vectorized and should work for any number of classes.

Hint: Use *broadcasting*. First, read about broadcasting in the *important guidelines* on Quercus. Second, following the examples in the *guidelines*, implement Equation (10) in several steps and define several intermediate arrays — A, B, C

[3] μ is the Greek letter mu. Consequently, when μ is used in mathematical equations, it is often represented by a variable called `mu` in computer programs.

[4] See the documentation for `QuadraticDiscriminantAnalysis` in `sklearn` to find out how to retrieve the estimated mean vectors.

[5] Σ is the upper-case Greek letter sigma. Consequently, when Σ is used in mathematical equations, it is often represented by a variable called `Sigma` in computer programs. Note that the symbol Σ is often used in two different ways: (*i*) to represent covariance matrices, and (*ii*) to represent summation. This is common in machine learning and in statistics (and in this assignment). The meaning of Σ should be clear from context.

and D — as follows:

$$\begin{aligned} A_{nki} &= X_{ni} - \mu_{ki} \\ B_{nkij} &= A_{nki} A_{nkj} \\ C_{nkij} &= T_{nk} B_{nkij} \\ D_{kij} &= \sum_n C_{nkij} \\ \Sigma_{kij} &= D_{kij}/(N_k - 1) \end{aligned}$$

where most of these steps require broadcasting. Note that B and C are rank 4 arrays, because they have 4 subscripts.

Note also the division by $N_k - 1$ in the last line above. This is slightly different from the maximum likelihood estimate in Equation (10), which divides by N_k. This is often done in statistics when estimating variance and covariance because it gives an unbiased estimate. In particular, it is done by the class `QuadraticDiscriminantAnalysis` in `sklearn`. You should therefore do it is well, so you can compare your estimate to theirs.

Test your function on the training data of Question 1 by comparing its estimate of the covariance matrices to those produced by the `sklearn` classifier in part (a). In particular, if Σ is the array of covariance matrices returned by your function, and Σ^{sk} is the array returned by the `sklearn` classifier, then you should find that $\Sigma = \Sigma^{sk}$ almost exactly. To demonstrate this, print out the total squared difference in the two arrays. That is, print out $\sum_{kij}(\Sigma_{kij} - \Sigma^{sk}_{kij})^2$. It should be less than 10^{-25}. (This formula demonstrates the overloaded use of the Σ symbol. Here, the large Σ is a summation sign, and the two smaller ones represent covariance matrices.)

(h) (? points) *Prior Probabilities.* Write a Python function `EstPrior(T)` that estimates the prior probability of each class in QDA from `T`, a one-hot matrix of class labels. Your function should return a vector whose k^{th} element is the estimated prior probability of class k. Note that the elements of the vector should sum to 1. In defining your function, do not use any loops, and do not use any functions from `sklearn`. Your code should be completely vectorized and should work for any number of classes.

Test your function on the training data of Question 1 by comparing its estimate of the prior probabilities to those produced by the `sklearn` classifier in part (a). In particular, if P is the vector of probabilities returned by your function, and P^{sk} is the vector returned by the `sklearn` classifier, then you should find that $P = P^{sk}$. To demonstrate this, print out the total squared difference in the two vectors. It should be exactly 0.

(i) (? points) *Posterior Probabilities.* By fitting mean vectors, covariance matrices and prior probabilities to the training data, you have now trained a QDA classifier. In the rest of this question, you will evaluate this classifier on the test data. The first step is to estimate the posterior distribution of the test data.

Write a Python function `EstPost(mean,cov,prior,X)` that uses Bayes rule to estimate the posterior probability of each class in QDA for each input vector in data matrix `X`. Here, `mean`, `cov` and `prior` are the mean vectors, covariance matrices and prior probabilities of each QDA class. Your function should return a matrix whose nk^{th} entry is $p(k|\mathbf{x}^{(n)})$, the posterior probability of class k for input vector n.

In defining your function, you may use the function `multivariate_normal.pdf` in `scipy.stats`. You may not use any other functions in `scipy` and no functions in `sklearn`. You may also use exactly one loop with exactly one line in its body. Other than this, your code should be completely vectorized. It should also work for any number of classes. Do not use any nested loops, and do not use helper functions.

Use your function to estimate the posterior probability of the *test data* of Question 1. In doing this, you should use the mean vectors, covariance matrices and prior probabilities as estimated by your functions above on the *training data*.

Compare this estimate to the one generated by the `sklearn` classifier in part (a). That is, let P be the vector of probabilities returned by `EstPost`, and let P^{sk} be the vector of posterior probabilities estimated by the `sklearn` classifier. You should find that $P = P^{sk}$ almost exactly. To demonstrate this, print out the total squared difference in the two vectors. It should be less than 10^{-25}.

Note: if you did not complete all of the functions above, you may test `EstPost` by providing it with mean vectors, covariance matrices and/or prior probabilities estimated by the `sklearn` classifier from part (a), instead of values estimated by your own functions. In this case, your testing code should print out messages stating which values are provided by `sklearn` by using one or more of the following print statements:

```
print('mean vectors provided by sklearn')
print('covariance matrices provided by sklearn')
print('prior probabilities provided by sklearn')
```

(j) (? points) *Cross Entropy.* In the previous question, you generated two posterior distributions of the test data, one based on your QDA classifier, and one based on `sklearn`'s QDA classifier. Estimate the average cross entropy of these two posterior distributions. Print these two estimates and their difference. It should be clear which estimate is based on your QDA classifier, and which is based on the `sklearn` classifier. The two estimates should be almost exactly the same, and the difference should be less than 10^{-15}. Do not use any loops, and do not use any functions from `sklearn`. All code should be vectorized.

(k) (? points) *Accuracy.* Use the posterior distribution of the test data to estimate the test accuracy of your QDA classifier. In addition, use the `score` method of the `sklearn` classifier to estimate its test accuracy. Print these two estimates and their difference. It should be clear which estimate is based on your QDA classifier, and which is based on the `sklearn` classifier. The two estimates should be exactly the same, and the difference should be 0. Do not use any loops, and

do not use any functions from `sklearn` other than the `score` method. All code should be vectorized.

Figure 2:

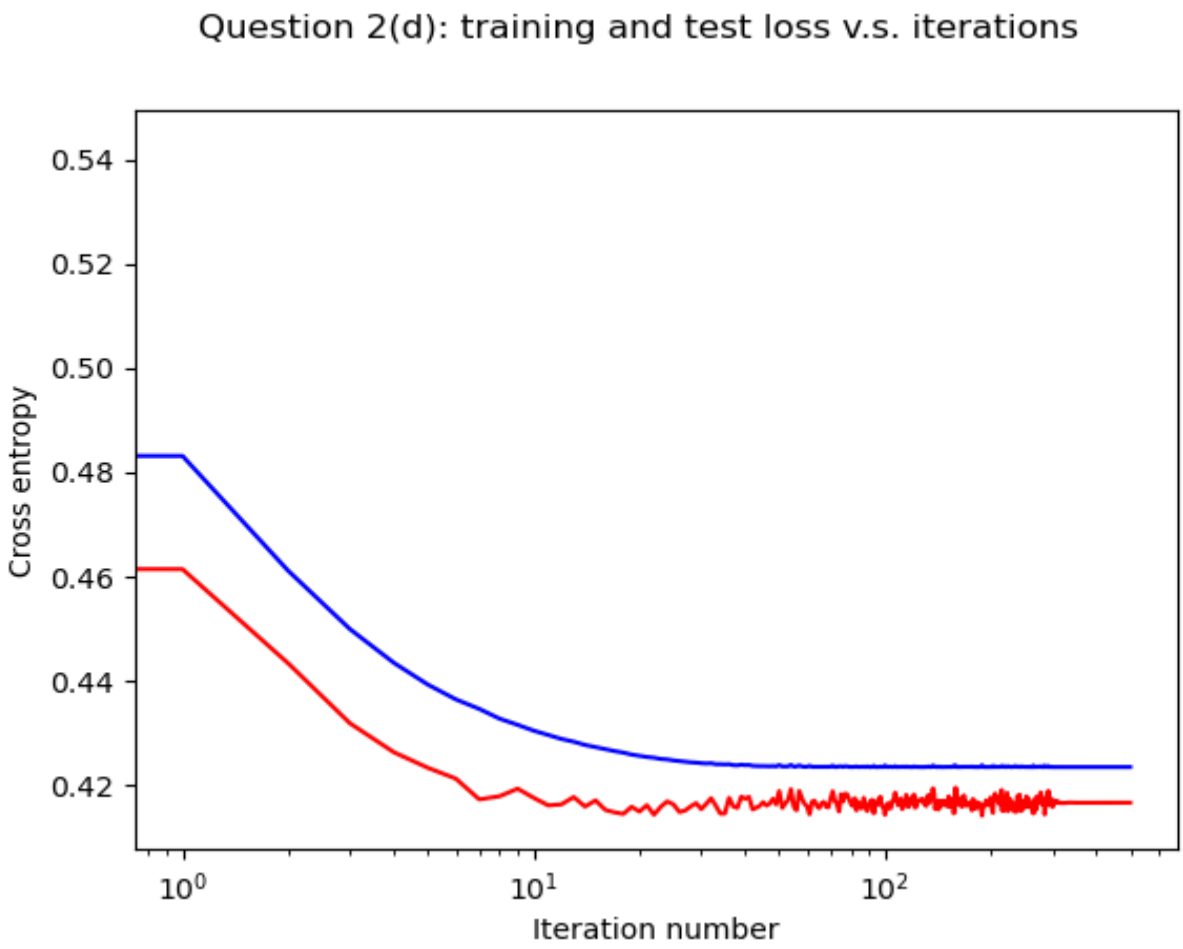

Question 2(d): training and test loss v.s. iterations
Cross entropy
0.54
0.52
0.50
0.48
0.46
0.44
0.42
10^0
10^1
10^2
Iteration number

Figure 3:

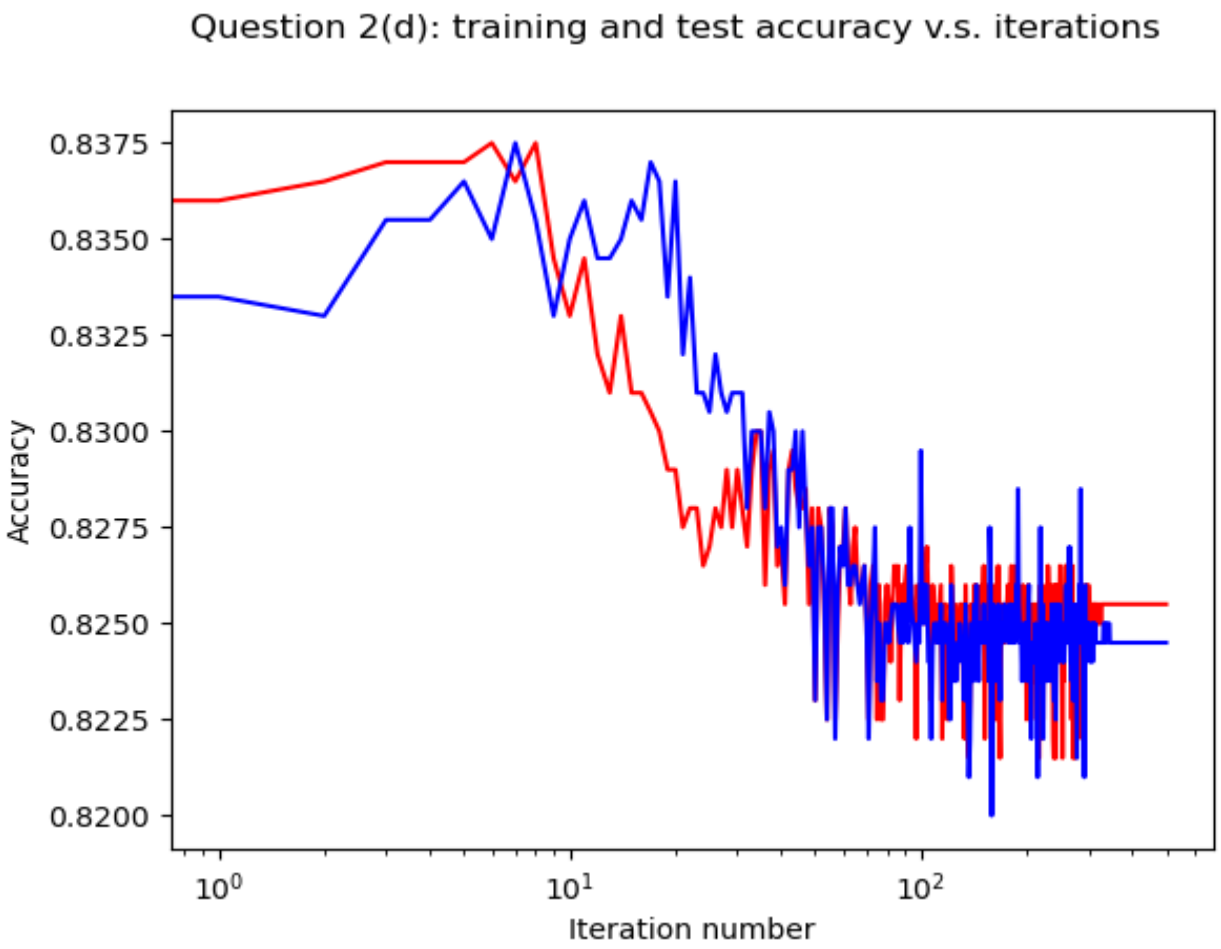

Question 2(d): training and test accuracy v.s. iterations
Accuracy
0.8375
0.8350
0.8325
0.8300
0.8275
0.8250
0.8225
0.8200
10^0
10^1
10^2
Iteration number

Figure 4:

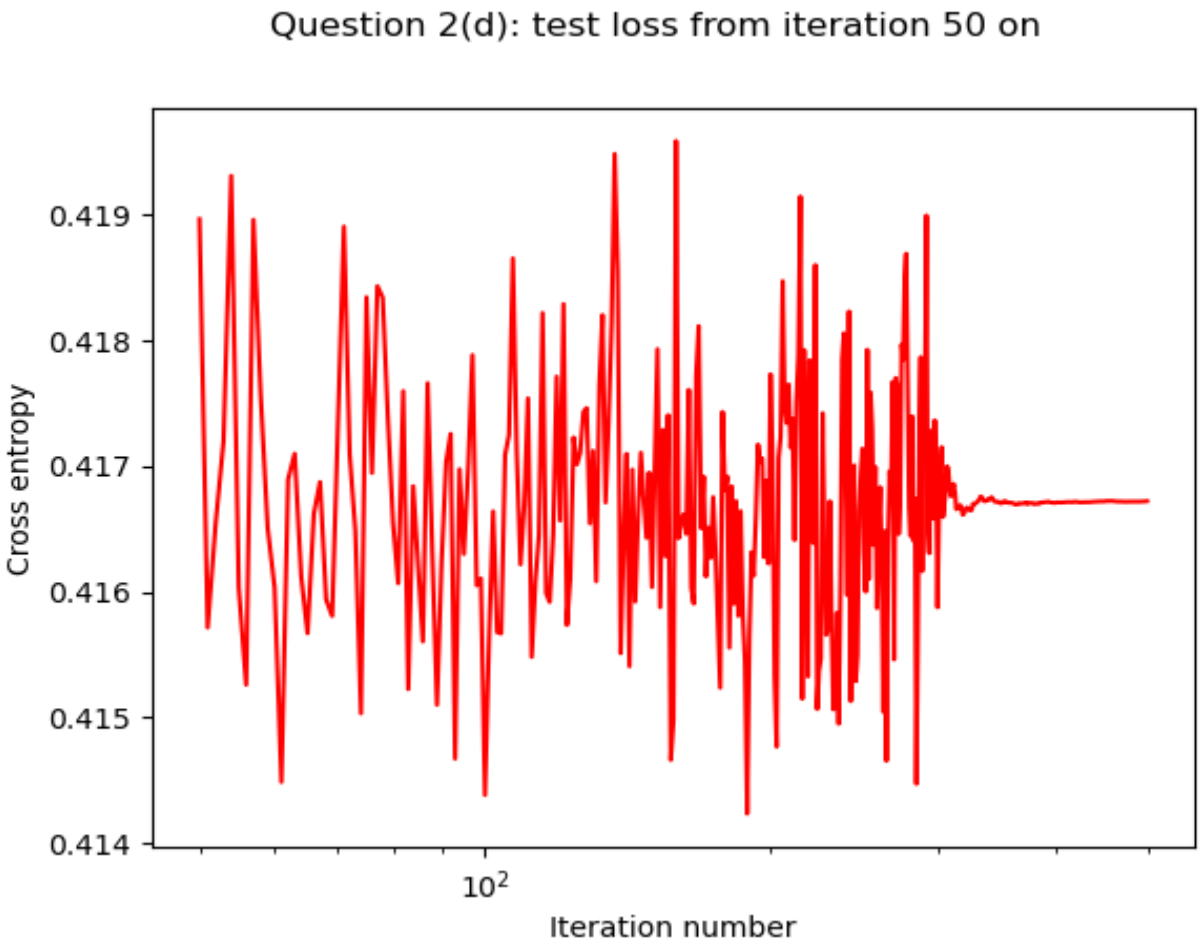
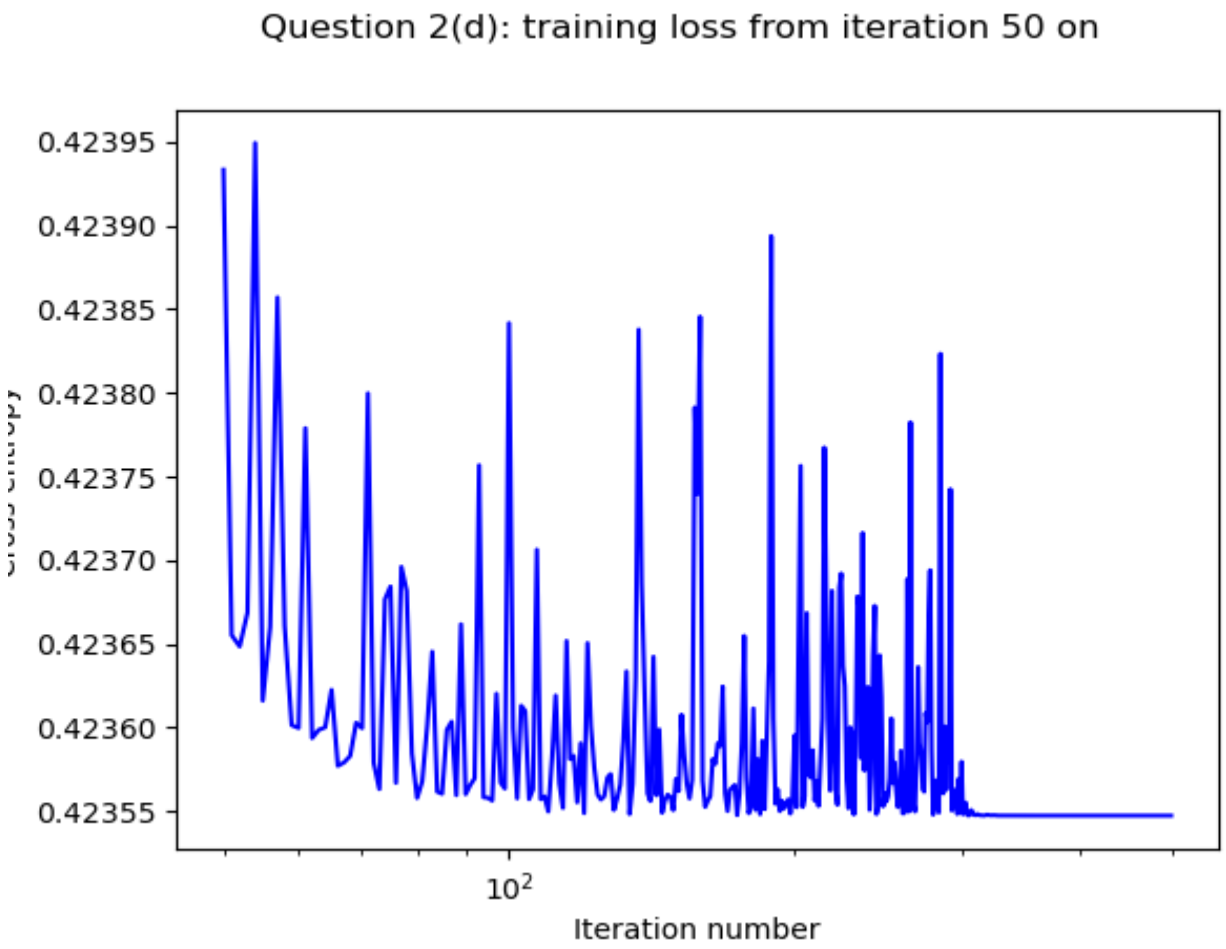

Question 2(d): test loss from iteration 50 on
Cross entropy
0.419
0.418
0.417
0.416
0.415
0.414
10^2
Iteration number

Figure 5:


Question 2(d): training loss from iteration 50 on
0.42395
0.42390
0.42385
0.42380
0.42375
0.42370
0.42365
0.42360
0.42355
10^2
Iteration number