CSC338. Homework 6

Due Date: Wednesday March 4, 9pm

Please see the guidelines at https://www.cs.toronto.edu/~lczhang/338/homework.html

What to Hand In

Please hand in 2 files:

- Python File containing all your code, named `hw6.py`.
- PDF file named `hw6_written.pdf` containing your solutions to the written parts of the assignment. Your solution can be hand-written, but must be legible. Graders may deduct marks for illegible or poorly presented solutions.

If you are using Jupyter Notebook to complete the work, your notebook can be exported as a .py file (File -> Download As -> Python). Your code will be auto-graded using Python 3.6, so please make sure that your code runs. There will be a 20% penalty if you need a remark due to small issues that renders your code untestable.

Make sure to remove or comment out all matplotlib or other expensive code before submitting your homework!

Submit the assignment on **MarkUs** by 9pm on the due date. See the syllabus for the course policy regarding late assignments. All assignments must be done individually.

```
import math
import numpy as np
```

Question 1

Part (a) – 3 pt

Write a function `solve_normal_equation` that takes an $m \times n$ matrix A (with $m > n$ and $rank(A) = n$) and a vector b, and solves the linear least squares problem $Ax \approx b$ using the Normal Equation method.

Use the `cholesky_factorize` method and other methods that you wrote in previous homeworks, but do not use any functions from the package `np.linalg`

```
def solve_normal_equation(A, b):
    """
    Return the solution x to the linear least squares problem
        $$Ax \approx b$$ using normal equations,
    where A is an (m x n) matrix, with m > n, rank(A) = n, and
          b is a vector of size (m)
    """
```

Part (b) – 1 pt

Consider the linear least square system below, where A is a 50×3 matrix, and b is a vector of size 50. The system is derived from the "iris" dataset.

Use the `solve_normal_equation` function you wrote in part (a) to find the vector `x` that minimizes the norm of `matmul(A, x) - b`.

Store the solution in the variable `iris_x` and the 2-norm of the residual in `iris_residual`. You may use the function `np.linalg.norm`.

```
irisA = np.array([[4.8, 6.3, 5.7, 5.1, 7.7, 5.6, 4.9, 4.4, 6.4, 6.2, 6.7, 4.5, 6.3,
                   4.8, 5.8, 4.7, 5.4, 7.4, 6.4, 6.3, 5.1, 5.7, 6.5, 5.5, 7.2, 6.9,
                   6.8, 6. , 5. , 5.4, 5.6, 6.1, 5.9, 5.6, 6. , 6. , 4.4, 6.9, 6.7,
```

```
                    5.1, 6. , 6.3, 4.6, 6.7, 5. , 6.7, 5.8, 5.1, 5.2, 6.1],
                   [3.1, 2.5, 3. , 3.7, 3.8, 3. , 3.1, 2.9, 2.9, 2.9, 3.1, 2.3, 2.5,
                    3.4, 2.7, 3.2, 3.7, 2.8, 2.8, 3.3, 2.5, 4.4, 3. , 2.6, 3.2, 3.2,
                    3. , 2.9, 3.6, 3.9, 3. , 2.6, 3.2, 2.8, 2.7, 2.2, 3.2, 3.1, 3.1,
                    3.8, 2.2, 2.7, 3.1, 3. , 3.5, 2.5, 4. , 3.5, 3.4, 3. ],
                   [1.6, 4.9, 4.2, 1.5, 6.7, 4.5, 1.5, 1.4, 4.3, 4.3, 5.6, 1.3, 5. ,
                    1.6, 5.1, 1.3, 1.5, 6.1, 5.6, 6. , 3. , 1.5, 5.8, 4.4, 6. , 5.7,
                    5.5, 4.5, 1.4, 1.3, 4.1, 5.6, 4.8, 4.9, 5.1, 5. , 1.3, 4.9, 4.4,
                    1.5, 4. , 4.9, 1.5, 5.2, 1.3, 5.8, 1.2, 1.4, 1.4, 4.9]])
irisb = np.array([0.2, 1.5, 1.2, 0.4, 2.2, 1.5, 0.1, 0.2, 1.3, 1.3, 2.4, 0.3, 1.9,
                  0.2, 1.9, 0.2, 0.2, 1.9, 2.1, 2.5, 1.1, 0.4, 2.2, 1.2, 1.8, 2.3,
                  2.1, 1.5, 0.2, 0.4, 1.3, 1.4, 1.8, 2. , 1.6, 1.5, 0.2, 1.5, 1.4,
                  0.3, 1. , 1.8, 0.2, 2.3, 0.3, 1.8, 0.2, 0.2, 0.2, 1.8])
A = irisA.T # transpose
b = irisb

iris_x = None
iris_residual = None
```

Question 2

Part (a) – 3 pt

Write a function `householder_v` that returns the vector v that defines the Householder transform

$$H = I - 2\frac{vv^T}{v^T v}$$

that eliminates all but the first element of a vector a. You may use the function `np.linalg.norm`.

```
def householder_v(a):
    """Return the vector $v$ that defines the Householder Transform
        H = I - 2 np.matmul(v, v.T) / np.matmul(v.T, v)
    that will eliminate all but the first element of the
    input vector a. Choose the $v$ that does not result in
    cancellation.

    Do not modify the vector `a`.

    Example:
        >>> a = np.array([2., 1., 2.])
        >>> householder_v(a)
        array([5., 1., 2.])
        >>> a
        array([2., 1., 2.])
    """
```

Part (b) – 2 pt

Show that a Householder Transformation H is orthogonal. Include your solution in your PDF writeup.

Part (c) – 2 pt

Write a function `apply_householder` that applies the Householder transform defined by a vector v to a vector u. You should **not** compute the Householder transform matrix H. You should only need to compute vector-vector dot

products and vector-scalar multiplications.

```
def apply_householder(v, u):
    """Return the result of the Householder transformation defined
    by the vector $v$ applied to the vector $u$. You should not
    compute the Householder matrix H directly.

    Example:

    >>> apply_householder(np.array([5., 1., 2.]), np.array([2., 1., 2.]))
    array([-3.,  0.,  0.])
    >>> apply_householder(np.array([5., 1., 2.]), np.array([2., 3., 4.]))
    array([-5. ,  1.6,  1.2])
    """
```

Part (d) – 3 pt

Write a function `apply_householder_matrix` that applies the Householder transform defined by a vector v to all the columns of a matrix U. You should **not** compute the Householder transform matrix H.

Do not use for loops. Instead, you may find the numpy function `np.outer` useful.

```
def apply_householder_matrix(v, U):
    """Return the result of the Householder transformation defined
    by the vector $v$ applied to all the vectors in the matrix U.
    You should not compute the Householder matrix H directly.

    Example:

    >>> v = np.array([5., 1., 2.])
    >>> U = np.array([[2., 2.],
                      [1., 3.],
                      [2., 4.]])
    >>> apply_householder_matrix(v, U)
    array([[-3. , -5. ],
           [ 0. ,  1.6],
           [ 0. ,  1.2]])
    """
```

Part (e) – 3 pt

Write a function `solve_qr_householder` that takes an $m \times n$ matrix A and a vector b, and solves the linear least squares problem $Ax \approx b$ using Householder QR Factorization. You may use `np.linalg.solve` to solve any square system of the form $Ax = b$ that you produce.

You should use the helper function `qr_householder` that takes a matrix A and a vector b and performs the Householder QR Factorization using the functions you wrote in parts (b-d).

```
def qr_householder(A, b):
    """Return the matrix [R 0]^T, and vector [c1 c2]^T equivalent
    to the system $Ax \approx b$. This algorithm is similar to
    Algorithm 3.1 in the textbook.
    """
    for k in range(A.shape[1]):
        v = householder_v(A[k:, k])
        if np.linalg.norm(v) != 0:
            A[k:, k:] = apply_householder_matrix(v, A[k:, k:])
            b[k:] = apply_householder(v, b[k:])
```

```
    # now, A is upper-triangular
    return A, b

def solve_qr_householder(A, b):
    """
    Return the solution x to the linear least squares problem
        $$Ax \approx b$$ using Householder QR decomposition.
    Where A is an (m x n) matrix, with m > n, rank(A) = n, and
          b is a vector of size (m)
    """
```

Question 3

For the next few questions, we will use the MNIST dataset for digit recognition Download the files `mnist_images.npy` and `mnist_labels.npy` from the course website, and place them into the same folder as your ipynb file.

The code below loads the data, splits it into "train" and "test" sets, and plots the a subset of the data. We will use `train_images` and `train_labels` to set up Linear Least Squares problems. We will use `test_images` and `test_labels` to test the models that we build.

```
mnist_images = np.load("mnist_images.npy")
test_images = mnist_images[4500:]  #  500 images
train_images = mnist_images[:4500] # 4500 images
mnist_labels = np.load("mnist_labels.npy")
test_labels = mnist_labels[4500:]
train_labels = mnist_labels[:4500]

def plot_mnist(remove_border=False):
    """ Plot the first 40 data points in the MNIST train_images. """
    import matplotlib.pyplot as plt
    plt.figure(figsize=(10, 5))
    for i in range(4 * 10):
        plt.subplot(4, 10, i+1)
        if remove_border:
            plt.imshow(train_images[i,4:24,4:24])
        else:
            plt.imshow(train_images[i])# plot_mnist() # please comment out this line before submission
```

Part (a) – 1 pt

How many examples of each digit are in `train_images`? You should use the information in `train_labels`.

Some of the code in the later part of this question might be helpful. You might also find the `sum` function helpful.

Save your result in the array `mnist_digits`, where `mnist_digits[0]` should contain the number of digit `0` in `train_images`, `mnist_digits[1]` should contain the number of digit `1` in `train_images`, etc.

```
mnist_digits = []
```

Part (b) – 2 pt

We will build a rudimentary model to predict whether a digit is the digit `0`. Our features will be the intensity at each pixel. There are 28 * 28 = 784 pixels in each image. However, in order to obtain a matrix A that is of full rank, we will ignore the pixels along the border. That is, we will only use 400 pixels in the center of the image.

Look at a few of the MNIST images using the `plot_mnist` function written for you. Why would our matrix A not be full rank if we use all 784 pixels in our model?

If this question doesn't make sense yet, you might want to come back to it after completing the rest of this question.

Include your solution in your PDF writeup.

```
# Plot the MNIST images, with only the center pixels that we will use
# in our digit classification model.
#plot_mnist(True) # please comment out this line before submission
```

Part (c) – 1 pt

We will now build a rudimentary model to predict whether a digit is the digit `0`. To obtain a matrix of full rank, we will use the 400 pixels at the center of each image as features. Our target will be whether our digit is `0`.

In short, the model we will build looks like this:

$$x_1 p_1 + x_2 p_2 + ... + x_{400} p_{400} = y$$

Where p_i is the pixel intensity at pixel i (the ordering of the pixel's doesn't actually matter), and the value of y determines whether our digit is a `0` or not.

We will solve for the coefficients x_i by solving the linear least squares problem $Ax \approx b$, where A is constructed using the pixel intensities of the images in `train_images`, and y is constructed using the labels for those images. For convenience, we will set $y = 1$ for images of the digit `0`, and $y = 0$ for other digits.

We should stress that in real machine learning courses, you will learn that this is not the proper way to build a digit detector. However, digit detection is quite fun, so we might as well use to tools that we have to try and solve the problem.

The code below obtains the matrices A and the vector b of our least squares problem, where A is a $m \times n$ matrix and b is a vector of length m.

What is the value of m and n? Save the values in the variables `mnist_m` and `mnist_n`.

```
A = train_images[:, 4:24, 4:24].reshape([-1, 20*20])
b = (train_labels == 0).astype(np.float32)

mnist_m = None
mnist_n = None
```

Part (d) – 1 pt

Use the Householder QR decomposition method to solve the system. Save the result in the variable `mnist_x`. Save the norm of the residuals of this solution in the variable `mnist_r`.

```
mnist_x = None
mnist_r = None
```

Part (e) – 1 pt

Consider `test_images[0]`. Is this image of the digit 0? Set the value of `test_image_0` to either `True` or `False` depending on your result.

Let p be the vector containing the values of the 400 center pixels in `test_image[0]`. The features are extracted for you in the variabel `p`. Use the solution `mnist_x` to estimate the target y for the vector p. Save the (float) value of the predicted value of y in the variable `test_image_0_y`.

```
# import matplotlib.pyplot as plt # Please comment before submitting
# plt.imshow(test_images[0]) # Please comment before submitting
p = test_images[0, 4:24, 4:24].reshape([-1, 20*20])
```

```
test_image_0 = None
test_image_0_y = None
```

Part (f) – 2 pt

Write code to predict the value of y for **every** image in `test_images`. Save your result in `test_image_y`.

Do not use a loop.

```
test_image_y = None
```

Part (g) – 1 pt

We will want to turn the continuous estimates of y into discrete predictions about whether an image is of the digit 0.

We will do this by selecting a **cutoff**. That is, we will predict that a test image is of the digit 0 if the prediction y for that digit is at least 0.5.

Create a numpy array `test_image_pred` with `test_image_pred[i] == 1` if `test_image[i]` is of the digit 0, and `test_image_pred[i] == 0` otherwise. Then, run the code to compute the `test_accuracy`, or the portion of the times that our prediction matches the actual label.

HINT: You might find the code in Part(c) helpful.

(This is somewhat of an arbitrary cutoff. You will learn the proper way to do this prediction problem in a machine learning course like CSC411 or CSC321.)

```
test_image_pred = None
# test_accuracy = sum(test_image_pred == (test_labels == 0).astype(float)) / len(test_labels)
# print(test_accuracy)
```

Part (h) – 4 pt

So far, we built a linear least squares model that determines whether an image is of the digit 0. Let's go a step further, and build such a model for **every** digit!

Complete the function `mnist_classifiers` that uses `train_images` and `train_labels`, and uses the function `solve_normal_equation` to build a linear least squares model for every digit. The function should return a matrix xs of shape 10×400, with `xs[0] == mnist_x1` from earlier.

Make sure to comment out any code you use to test `mnist_classifier`, or your code might not be testable.

This part of the code will be graded by your TA.

```
def mnist_classifiers():
    """Return the coefficients for linear least squares models for every digit.

    Example:
        >>> xs = mnist_classifiers()
        >>> np.all(xs[0] == mnist_x1)
        True
    """
    # you can use train_images and train_labels here, and make
    # whatever edits to this function as you wish.
    A = train_images[:, 4:24, 4:24].reshape([-1, 20*20])
    xs = []
    # ...
    return np.stack(xs)
```

Just for fun...

The code below makes predictions based on the result of your `mnist_classifier`. That is, for every test image, the code runs all 10 models to see whether the test image contains each of the 10 digits. We make a discrete prediction about which digit the image contains by looking at which model yields the **largest** value of y for the image.

The code then compares the result against the actual labels, computes the accuracy measure: the fraction of predictions that is correct. Just for fun, look at the prediction accuracy of our model, but please comment any code you write before submitting your assignment.

Again, in machine learning and statistics courses you will learn ways to classifying digits that are better and more principled. You'll achieve a much better test accuracy than what we have here.

```
def prediction_accuracy(xs):
    """Return the prediction
    """
    testA = test_images[:, 4:24, 4:24].reshape([-1, 20*20])
    ys = np.matmul(testA, xs.T)
    pred = np.argmax(ys, axis=1)
    return sum(pred == test_labels) / len(test_labels)
```