

CSC338. Assignment 2

Due Date: March 11, 11:59pm on Markus

What to Hand In

Please hand in 2 files:

- Python File containing all your code, named `a2.py`.
- PDF file named `a2_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 15% penalty if you need a remark due to small issues that renders your code untestable. (Please note the penalty is higher than in A1!)

Make sure to remove or comment out all matplotlib or other expensive code before submitting your code! Expensive code can render your code untestable, and you will incur the 15% penalty for remark.

Submit the assignment on **MarkUs** by 10pm 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.

For this question, we will work in the floating-point system $F(\beta = 10, p = 3, L = -10, U = 10)$.

Consider the system $A\mathbf{x} = \mathbf{b}$ where

$$A = \begin{bmatrix} 0.03 & 21.2 \\ 2.18 & -0.30 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 21.5 \\ 21.5 \end{bmatrix}$$

Part (a) – 2pt

Solve the system by hand, exactly, using infinite precision arithmetic. That is, the addition, multiplication, division, and other operations that you do will be exact (rather than the floating-point version). Do not use pivoting.

Include your step-by-step solution in your PDF writeup. Your *final* result can be rounded to 5 decimal places, but do not round any intermediate computations since we are using infinite precision arithmetic.

Part (b) – 2pt

Solve the system **using floating-point operations and the rounding rule is chopping**. In other words, replace every operation you used in part (a) with the floating-point operation in the system $F(\beta = 10, p = 3, L = -10, U = 10)$. Do not use pivoting.

Include your step-by-step solution in your PDF writeup.

Part (c) – 3pt

Solve this similar system using floating-point operations, but **with partial pivoting**.

Which result is closer to the true result that you obtained in Part (a)?

Include your step-by-step solution in your PDF writeup.

Question 2.

Part (a) [2 pt]

Prove that $\|A\|_\infty$ can be computed by computing the maximum row sum:

$$\|A\|_\infty = \max_i \sum_{j=1}^n |a_{ij}|$$

Part (b) [2 pt]

Suppose that A and B are two $n \times n$ matrices, and both are well-conditioned. Is $A(B^{-1})$ also well-conditioned? Why or why not? Include your answer and justifications in your pdf write-up. Be specific.

Part (c) [4 pt]

Describe an *efficient* algorithm to compute $d^T B^T A^{-1} B d$

Where:

- A is an invertible $n \times n$ matrix,
- B is an $n \times n$ matrix, and
- d is an $n \times 1$ vectors

Be clear and specific. Include your strategy in your pdf write-up.

Question 3

Part (a) – 2 pt

Suppose that A is a symmetric positive definite matrix. Show that the function

$$\|x\|_A = (x^T A x)^{\frac{1}{2}}$$

on a vector x satisfies the three properties of a vector norm.

Include your solution in your pdf writeup.

Part (b) – 6 pt

Complete the function `cholesky_factorize` that returns the Cholesky factorization of a matrix, according to its docstring.

```
def cholesky_factorize(A):
    """Return the Cholesky Factorization L of A, where
    * A is an nxn symmetric, positive definite matrix
    * L is lower triangular, with positive diagonal entries
    * $A = LL^T$

    >>> M = np.array([[8., 3., 2.],
                      [3., 5., 1.],
                      [2., 1., 3.]])
    >>> L = cholesky_factorize(M)
    >>> np.matmul(L, L.T)
    array([[8., 3., 2.],
```

```

        [3., 5., 1.],
        [2., 1., 3.]])
"""

```

Question 4

Part (a) – 4 pts

Complete the function `solve_rank_one_update` that solves the system $(A - \mathbf{u}\mathbf{v}^T)\mathbf{x} = \mathbf{b}$, assuming that the factorization $A = LU$ has already been done for you. You are welcome to add any helper functions that you wish, including functions that you wrote in homeworks 3 and 4. Just make sure that you include the helper functions in your python script submission.

```

def solve_rank_one_update(L, U, b, u, v):
    """Return the solution x to the system (A - u v^T)x = b, where
    A = LU, using the approach we derived in class using
    the Sherman Morrison formula. You may assume that
    the LU factorization of A has already been computed for you, and
    that the parameters of the function have:
        * L is an invertible nxn lower triangular matrix
        * U is an invertible nxn upper triangular matrix
        * b is a vector of size n
        * u and v are also vectors of size n

    >>> A = np.array([[2., 0., 1.],
                      [1., 1., 0.],
                      [2., 1., 2.]])
    >>> L, U = lu_factorize(A) # from homework 3
    >>> L
    array([[1. , 0. , 0. ],
           [0.5, 1. , 0. ],
           [1. , 1. , 1. ]])
    >>> U
    array([[ 2. ,  0. ,  1. ],
           [ 0. ,  1. , -0.5],
           [ 0. ,  0. ,  1.5]])
    >>> b = np.array([1., 1., 0.])
    >>> u = np.array([1., 0., 0.])
    >>> v = np.array([0., 2., 0.])
    >>> x = solve_rank_one_update(L, U, b, u, v)
    >>> x
    array([1. , 0. , -1.])
    >>> np.matmul((A - np.outer(u, v)), x)
    array([1. , 1. , 0.])
    """

```

Part (b) – 2 pt

Explain why using `solve_rank_one_update` does not give us accurate results in the below example:

```

def run_example():
    A = np.array([[2., 0., 1.],
                  [1., 1., 0.],
                  [1., 1., 1.]])
    L = np.array([[1., 0., 0.],
                  [0.5, 1., 0.]])

```

```

        [0.5, 1., 1.])
U = np.array([[2., 0., 1.],
              [0., 1., -0.5],
              [0., 0., 1.]])
b = np.array([1, 1, -1])
u = np.array([0, 0, 0.9999999999999999])
v = np.array([0, 0, 0.9999999999999999])
x = solve_rank_one_update(L, U, b, u, v)
print(np.matmul((A - np.outer(u, v)), x) - b)

```

Include your solution in your PDF writeup.

Part (c) – 4 pt

A rank 1 matrix has the form xy^T where x and y are column vectors. Suppose A and B are non-singular matrices. Show that $A - B$ is rank 1 if and only if $A^{-1} - B^{-1}$ is also rank 1.

Include your solution in your pdf writeup.

Hint: Use the Sherman-Morrison formula. This question is not supposed to be easy, so leave aside time to think!

Question 5.

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
         $H = I - 2 \text{np.matmul}(v, v.T) / \text{np.matmul}(v.T, v)$ 
        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  $m > n$ ,  $\text{rank}(A) = n$ , and
    b is a vector of size (m)
    """

```

Question 6

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 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

```

Remember to comment out any plotting related code before you submit your assignment!

```
# plot_mnist()
```

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:

$$x_1p_1 + x_2p_2 + \dots + x_{400}p_{400} = y$$

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 the 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 variable `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 CSC311.)

```
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 the functions you wrote in Q3 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)

```