ISTA 421 / INFO 521 - Homework 1

Due: Friday, September 8, 5pm

30 points total (same for undergraduates and graduates)

STUDENT NAME

Undergraduate / Graduate

Instructions

Always start assignments as soon as possible, do *NOT* wait until the last couple of days to get started.

On this (and other "regular" – i.e., non-midterm/final assignments) you may *discuss* exercises with others in the class, BUT your code and written solutions **MUST BE YOUR OWN**. Do not copy code or written solutions by others. Copying is cheating.

If you work with others, you must include in your submission a list of people you talked with. If you reference other material (fine to do!), you must cite those resources. Please review the course submissions instructions webpage (see below) for instructions on using ChatGPT, large language models and other generative AI.

Include in your final submission the written answers as a PDF format file called `hw1-assignment.pdf`, along with separate files for any python scripts that you write in support of answering the questions. Refer to the course submission instructions at:

`https://ml4ai-2023-fall-ml.github.io/submissions.html`

In general, keep files in the directories that they are initially found in the homework release. For example, in this homework, keep the `hw1.py` in the `code/` directory. Place the `hw1-assignment.pdf` that you will create in the top level (root) directory of the repository.

In future exercise sets, there will be additional or different exercises that graduate students must complete, but for this homework, everyone must do the same exercises for the same points.

Mathematical content: This is a mathematical subject and there is a wide variance in backgrounds of students who take this course. For example, there may be exercises in the assignments which seem more difficult than they really are simply because you are not (yet) used to the kind of problem.

The purpose of this assignment is to learn (or remember) how to write short programs involving numeric computation and some basic input/output (I/O) and plotting in Python to explore mathematical ideas.

Most of the exercises below have multiple parts. In most cases, the correspondence between what is being asked for and the points that can be earned is clearly indicated by the explicit question(s) or request(s) (e.g., exercise 5 clearly asks you to show that two expressions, one in scalar notation and one in vector notation are equivalent, and showing that earns you full credit). Exercises 6 through 9 ask you to add code to the `code/hw1.py` script, and the exercises have the flavor of a "tutorial". For these exercises, there is a ($) for each part of the exercise that requires an explicit answer that contributes to the points for the exercise. Each ($) is worth an equal proportion of the total points for that exercise, and in some cases, some parts of the exercise will earn a multiple of $'s (such as (3$), meaning it's worth 3 $'s). As an example, in exercise 7, there are three individual ($)'s and there is one (3$), so altogether there are 6 $'s; because exercise 7 is worth a total of 3 points and there are 6 $s, then each $ is worth 0.5 points.

In the following, FCML refers to the course text: Simon Rogers and Mark Girolami (2016), *A First Course in Machine Learning*, second edition. (All questions from the book are reproduced here in full, so you do not need the book to proceed.)

For general notes on using latex to typeset math, see: `http://en.wikibooks.org/wiki/LaTeX/Mathematics`.

Exercises

1. [0 points] If you haven't already, setup your Python programming environment!
   `https://ml4ai-2023-fall-ml.github.io/python_setup.html`

   If python is new to you, see the official python tutorial:
   `https://docs.python.org/3/tutorial/`

   See the Appendix at the end of this document for additional information about numpy arrays and matrices.

2. [1 point] **Exercise 1.1** from FCMA p.35

   Figure 1: Reproduction of figure 1.1, Olympic men's 100m data

   By examining Figure 1.1 [from p. 2 of FCMA, reproduced here], *estimate* (i.e., by hand / in your head – not by rigorous calculation) the kind of values we should expect for w_0 (y-intercept) and w_1 (slope) as parameters of a line fit to the data: Small (between 0 and 1), Medium (between 1 and 15) or Large (greater than 15)? Also, Positive or Negative? (No computer or calculator calculation is needed here – just estimate!)

   Solution. The slope will certainly be negative as the time is decreasing with respect to year. The slope (w_1) looks to be around -1/70 because the time decreased about 1 second over 70 years. w_o is little harder to predict but I would say around 20

NOTE: The following three exercises (3, 4 and 5) review basic linear algebra concepts.

Notation conventions:

- Script variables, such as x_{n2} and w_1 represent scalar values
- Lowercase bold-face variables, such as $\mathbf{x}_n$ and $\mathbf{w}$, represent vectors
- Uppercase bold-face variables, such as $\mathbf{X}$, represent m (rows) $\times$ n (columns) matrices
- Note that because all indexes in the following are either single digit integers (0, 1, ..., 9), or a single letter representing an integer index, e.g., n, I am representing multiple dimension indexes without a comma, as it is unambiguous; e.g., x_{32} is the element scalar value of $\mathbf{X}$ at row 3, column 2. When we have to refer to specific index values greater than 9, we'll use commas, such as $x_{32,3}$ is the scalar value in the 32nd row and 3rd column.
- '$\top$' in expressions like $\mathbf{w}^\top$ indicates the *transpose* operator.
- Unless stated otherwise, we will assume that all vectors *without* the transpose, $\top$, are *column* vectors, so $\mathbf{w}^\top$ is a *row vector*.
- It is sometimes convenient to express an example vector as a bracketed list of elements (e.g., in a sentence): $[x_1, x_2, ..., x_n]$. In general I am going to try to be careful about the orientation of vectors, so the previous example would be a *row* vector. To make it a column vector, I'll add a transpose: $[x_1, x_2, ..., x_n]^\top$.

3. [3 points] **Exercise 1.3** from FCMA p.35

Show that:

$$\mathbf{w}^\top \mathbf{X}^\top \mathbf{X} \mathbf{w} = w_0^2 \left(\sum_{n=1}^{N} x_{n1}^2 \right) + 2 w_0 w_1 \left(\sum_{n=1}^{N} x_{n1} x_{n2} \right) + w_1^2 \left(\sum_{n=1}^{N} x_{n2}^2 \right),$$

where

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}, \mathbf{X} = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \\ \vdots & \vdots \\ x_{N1} & x_{N2} \end{bmatrix}.$$

(Hint – it's probably easiest to do the $\mathbf{X}^\top \mathbf{X}$ first!)

Solution.

dimensions of $\mathbf{X}^\top$: 2,N
dimensions of $\mathbf{X}$: N,2
dimensions of $\mathbf{X}^\top \mathbf{X}$: 2,2

$$\mathbf{X}^\top \mathbf{X} = \begin{bmatrix} x_{11}^2 + x_{21}^2 + ... + x_{N1}^2 & x_{11}x_{12} + x_{21}x_{22} + ... + x_{N1}x_{N2} \\ x_{12}x_{11} + x_{22}x_{21} + ... + x_{N2}x_{N1} & x_{12}^2 + x_{22}^2 + ... + x_{N2}^2 \end{bmatrix}$$

dimensions of $\mathbf{w}^\top$: 1,2
dimensions of $\mathbf{w}$: 2,1

$\mathbf{w}^\top \mathbf{X}^\top \mathbf{X} = \begin{bmatrix} a & b \end{bmatrix}$

Where:
a = $w_0(x_{11}^2 + x_{21}^2 + ... + x_{N1}^2) + w_1(x_{12}x_{11} + x_{22}x_{21} + ... + x_{N2}x_{N1})$
b = $w_0(x_{11}x_{12} + x_{21}x_{22} + ... + x_{N1}x_{N2}) + w_1(x_{12}^2 + x_{22}^2 + ... + x_{N2}^2)$

$x_{11}^2 + x_{21}^2 + ... + x_{N1}^2 = \sum_{n=1}^N x_{n1}^2$
$x_{12}^2 + x_{22}^2 + ... + x_{N2}^2 = \sum_{n=1}^N x_{n2}^2$
$x_{12}x_{11} + x_{22}x_{21} + ... + x_{N2}x_{N1} = x_{11}x_{12} + x_{21}x_{22} + ... + x_{N1}x_{N2} = \sum_{n=1}^N x_{n1}x_{n2}$

Thus:
a = $w_0 \left(\sum_{n=1}^N x_{n1}^2\right) + w_1 \left(\sum_{n=1}^N x_{n1}x_{n2}\right)$
$b = w_0 \left(\sum_{n=1}^N x_{n1}x_{n2}\right) + w_1 \left(\sum_{n=1}^N x_{n2}^2\right)$

and $\mathrm{w}^\top \mathbf{X}^\top \mathbf{X}\mathbf{w} = aw_0 + bw_1$
$= w_0(w_0 \left(\sum_{n=1}^N x_{n1}^2\right) + w_1 \left(\sum_{n=1}^N x_{n1}x_{n2}\right)) +$
$w_1(w_0 \left(\sum_{n=1}^N x_{n1}x_{n2}\right) + w_1 \left(\sum_{n=1}^N x_{n2}^2\right))$

$=\mathrm{w}_0^2 \left(\sum_{n=1}^N x_{n1}^2\right) + 2w_0w_1 \left(\sum_{n=1}^N x_{n1}x_{n2}\right) + w_1^2 \left(\sum_{n=1}^N x_{n2}^2\right)$

4. [2 points] **Exercise 1.4** from FCMA p.35

Using $\mathbf{w}$ and $\mathbf{X}$ as defined in the previous exercise, show that $(\mathbf{X}\mathbf{w})^\top = \mathbf{w}^\top \mathbf{X}^\top$ by multiplying out both sides.

Solution. $\mathbf{X}\mathbf{w} = \begin{bmatrix} x_{11}w_0 + x_{12}w_1 \\ \vdots \\ x_{N1}w_0 + x_{N2}w_1 \end{bmatrix}$

$(\mathbf{X}\mathbf{w})^\top = \begin{bmatrix} x_{11}w_0 + x_{12}w_1 ... x_{N1}w_0 + x_{N2}w_1 \end{bmatrix}$

$\mathbf{w}^\top = \begin{bmatrix} w_0 & w_1 \end{bmatrix}$ $\mathbf{X}^\top = \begin{bmatrix} x_{11} & x_{21} & ... & x_{N1} \\ x_{12} & x_{22} & ... & x_{N2} \end{bmatrix}$

$\mathbf{w}^\top \mathbf{X}^\top = \begin{bmatrix} x_{11}w_0 + x_{12}w_1 ... x_{N1}w_0 + x_{N2}w_1 \end{bmatrix}$ thus $(\mathrm{Xw})^\top = \mathbf{w}^\top \mathbf{X}^\top$

5. [3 points] **Exercise 1.5** from FCMA p.35

When multiplying a scalar by a vector (or matrix), we multiply each element of the vector (or matrix) by that scalar. For $\mathbf{x}_n = [x_{n1}, x_{n2}]^\top$, $\mathbf{t} = [t_1, ..., t_N]^\top$, $\mathbf{w} = [w_0, w_1]^\top$, and

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1{}^\top \\ \mathbf{x}_2{}^\top \\ \vdots \\ \mathbf{x}_N{}^\top \end{bmatrix}$$

show that

$$\sum_n \mathbf{x}_n t_n = \mathbf{X}^\top \mathbf{t}$$

and

$$\sum_n \mathbf{x}_n \mathbf{x}_n^\top \mathbf{w} = \mathbf{X}^\top \mathbf{X}\mathbf{w}$$

Solution.

$$\mathbf{X}^\top \mathbf{t} = \begin{bmatrix} x_{11} & x_{21} & ... & x_{N1} \\ x_{12} & x_{22} & ... & x_{N2} \end{bmatrix} \begin{bmatrix} t_1 \\ \vdots \\ t_N \end{bmatrix} = \begin{bmatrix} x_{11}t_1 + x_{21}t_2 + ... + x_{N1}t_n \\ x_{12}t_1 + x_{22}t_2 + ... + x_{N2}t_n \end{bmatrix}$$

$$\textstyle\sum_n \mathbf{x}_n t_n = \begin{bmatrix} x_{11}t_1 \\ x_{12}t_1 \end{bmatrix} + \begin{bmatrix} x_{21}t_2 \\ x_{22}t_2 \end{bmatrix} + ... + \begin{bmatrix} x_{n1}t_n \\ x_{n2}t_n \end{bmatrix} = \begin{bmatrix} x_{11}t_1 + x_{21}t_2 + ... + x_{N1}t_n \\ x_{12}t_1 + x_{22}t_2 + ... + x_{N2}t_n \end{bmatrix}$$

Thus,

$$\sum_n \mathbf{x}_n t_n = \mathbf{X}^\top \mathbf{t}$$

$$\mathbf{X}^\top \mathbf{X} = \begin{bmatrix} \mathbf{x}_1{}^\top & \mathbf{x}_2{}^\top & ... & \mathbf{x}_N{}^\top \end{bmatrix} \begin{bmatrix} \mathbf{x}_1{}^\top \\ \mathbf{x}_2{}^\top \\ \vdots \\ \mathbf{x}_N{}^\top \end{bmatrix} = \begin{bmatrix} (\mathbf{x}_1{}^\top)^2 + (\mathbf{x}_2^\top)^2 + ... + (\mathbf{x}_N^\top)^2 \end{bmatrix}$$

$$\mathrm{X}^\top \mathbf{X}\mathbf{w} = \begin{bmatrix} (\mathbf{x}_1{}^\top)^2 + (\mathbf{x}_2^\top)^2 + ... + (\mathbf{x}_N^\top)^2 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$$

Thus,

$$\sum_n \mathbf{x}_n \mathbf{x}_n^\top \mathbf{w} = \mathbf{X}^\top \mathbf{X}\mathbf{w}$$

NOTE: In the following four exercises (6, 7, 8 and 9) you will implement your solution in the function-stubs provided in the python script `hw1.py` found in the `code/` subdirectory in the hw1_release GitHub repository. Each exercise below will specify which function you are to edit (their names correspond to the number of the exercise), and you will be prompted to fill in your solution code in the locations in the `hw1.py` indicted by the `#### YOUR CODE HERE ####` comments.

At the top of the `hw1.py` file are four global variables named `RUN_EXERCISE_<#>`, where the `<#>` number corresponds to the exercise/function number. They are set to `False` by default. When you go to implement a function, you need to set the corresponding variable to `True` for that function to execute. There is no harm in setting all of them to `True`, just be aware that that means all of the exercise functions will run, some of which may not be relevant to the exercise you are currently working on. It does not matter what states these variables are in for running the tests – the tests will call the exercise functions independently – nor does it matter what state they're set to for your final submission. These are simply provided as a convenience for you while you are working on implementing the functions.

You *can*...

- Add additional functions to the `hw1.py` file that are then called within the exercise functions.
- Import additional packages for use in the file using the `import` command. In at least one exercise below, you will be explicitly asked to include an additioanl package. However, if you want to include other packages, **please ask the instructors first**. In general, these exercises are intended to give you experience implementing aspects of the calculations and data manipulation without relying on already-existing solutions provided by other packages. We will always try to be explicit about which packages are to be used.

You *cannot* edit exercise function arguments or the exercise function return values – those must remain as they are defined in order for the automated `pytest` tests to work. The tests are for your benefit!

The `code/` directory also contains test scripts corresponding to each of the following exercises. Feel free to look at the test contents, but **do not** alter them in any way. From the tests you can see what behavior is expected for the testable parts of the exercises (based on the values returned by the exercise functions). However, do not adjust your code simply to return the values that pass the test. Instead, you must implement the processes that generated the expected behavior described in the exercises below. Do not simply make the functions return the passing value while bypassing implementation of the functionality described in the exercises; **doing this just to pass the tests will be considered cheating**. If you have any questions about this, please contact us (the instructors)!

To run the tests, follow the instructions for running `pytest` provided in the course website submission instructions:
`https://ml4ai-2023-fall-ml.github.io/submissions.html`

6. [7 points] Reading, Plotting and Scaling Data.

   In this exercise you will implement your solution in the function `exercise_6` in the provided python script `code/hw1.py`

   This exercise requires you to load a 1-d array of data from the file `walk.txt` found in the `data/` directory of the `hw1_release` repository. You will store the data in the variable `walk_arr`. Use the numpy function `loadtxt` to load the data. Note that since the data is comma-separated, you will need to specify the delimiter to the function (\$).

   The `walk` data represents a "random walk", where from one element to the next (a "step") in the array, the data moves up or down by some amount. This random walk was created by sampling a new point (the next

"step") from a Gaussian (or Normal) distribution centered at the current step.

Use the `matplotlib.pyplot` package to create a figure that plots the data ($). You will need to add an `import` statement at the top of the `hw1.py` file to import the `matplotlib.pyplot` package. Here is the recipe to create the figure with the plot (look up the matplotlib documentation for the instructions on how to use the following commands if they are not familiar):

- Create a new figure using the pyplot `figure` command
- Plot the data using `plot`
- Be sure to label your axes (using `xlabel` and `ylabel`) and provide a title (using `title`). The y-axis should be labeled "Location", the x-label should be labeled "Step", and the title of the figure should be "Random Walk".
- Finally, save your figure as a PNG file, using the function argument variable `path_to_figure` as specifying the figure name and path (where the figure will be stored: `../figures/walk.png`). Use the pyplot function `savefig` to save the figure; you will need to specify the format (`fmt`) as `'png'`.

At this point you should be able to run the `hw1.py` script (assuming you have set `RUN_EXERCISE_6` to `True`) and the figure should be generated and saved. The figure will be plotted in a window, but at this point the window will disappear immediately. In order to get the figure window to stay open until you close it, go to the very bottom of the `hw1.py` script where the line

```
if __name__ == '__main__':
```

appears, and in the first conditional that executes `exercise_6` you will see a `#### YOUR CODE HERE ####` that instructs you to add a call to the `matplotlib.pyplot show()` function. This will make it so that the figure window stays open (and the script will remain paused) until you close the window.

In the written part of the homework submission (`hw1-assignment.pdf`) you must include the figure along with a caption that explains what the figure represents ($). You will *always* need to provide descriptive captions for ANY figures you include in any assignment in this class. If you are using latex to format your written assignment, I have provided commented latex code after the "<Solution goes here.>" line below that will display the figure; however, you must still provide your own descriptive caption (replacing the place-holder text for the caption in the latex code).

Next, you will use the numpy `min` and `max` functions to find the lowest and highest value points in the random walk ($).

Hint: take a look at the related `amin` *and* `amax` *functions for full documentation.*

Now, suppose you need to scale and shift all of these points to fall between the values of -2 and 3, while preserving the relative "shape" of the walk. This is a very common task in general data manipulation, called "linear scaling" (or a "linear transformation"). In a linear scaling, you can "scale" the values by multiplying them by some value, and also "shift" the values by adding (or subtracting) some value. Your task in this part of the exercise is to write a function that takes three arguments – the `walk_arr` array, a minimum value and a maximum value – and returns a new array of the same length but the new array values are linearly scaled so that they now have a minimum value -2 and maximum value 3 (still preserving the relative "shape" of the random walk). Call this function to set the value of the variable `walk_arr_scaled` (2$).

Finally: **You must always document your code!** Documenting your code is required for this class – you will loose points if you do not document your code. Python in-line comments can be added using the `#` character – anything following will be ignored by the Python interpreter. Python also uses a special idiom for documenting functions: Right after the function signature line, add documentation within a triple-quote body, e.g.:

```
def scale01(arr):
    """
    Linearly scale the values of an array in the range [0,1]
    :param arr: input ndarray
    :return: scaled ndarray
    """
    <function_body>...
```

Beyond making your source code easier to understand and maintain, you also get the benefit of making documentation available within the python console, once functions are defined within the python instance. For example, once I've executed the above function definition within the python console, I can execute `help(scale01)` as follows:

```
>>> help(scale01)
Help on function scale01:

scale01(arr)
    Linearly scale the values of an array in the range [0,1]
    :param arr: input ndarray
    :return: scaled ndarray
```

Document your code (with inline comments) and provide function docstrings for each function you write in this homework ($). This includes the new function you just created and `exercise_6`. Do the same for the other exercise functions as well (explaining what they compute) as you implement them. NOTE: The unit tests will test whether you have added docstrings to each of the exercise functions.

Solution. see code

7. [3 points] Working with a Random Number Generator.

   For this exercise you will fill in the function `exercise_7` in `hw1.py`.

   The first task is to set the numpy random seed (look up how to set this!) to the value 7 ($). On a computer, there is no such thing as a *truly random* random number generator. Instead, there is some function that starts at some point and then generates a *deterministic* sequence of values. When the random number generator is called (by calling some random number function), the next value in this sequence is selected. If you start from the same starting point (as specified by the generator "seed"), you will get the *exact same* sequence of "random" numbers. In this exercise, we will demonstrate this by first setting numpy's the random number seed to a particular value (was 7), then run an experiment that involves drawing "random" numbers, followed by again setting the random seed to the same value (7), and repeat the experiment. What we will see is that the we end up with the identical behavior out of the two experiments, as a result of both experiments getting the same sequence of "random" numbers.

   After first setting the numpy random seed, the next task is to finish implementing the `run_experiment` function ($). (Yes, it is possible to have a function *within* another function in Python! Many languages allow this, but it might be new to you.) This function will simulate repeatedly throwing two "fair", six-sided dice. The "rolling" of a six-sided dice can be simulated as a call to the numpy random function `randint`, where the argument to `randint` is 6. Look up the function to see what it does.

   We're not just interested in simulating the "rolling" of the two dice, but in particular we're interested in estimating the probability that the two dice end up with "doubles", meaning that both dice have the same value. The experiment will estimate this probability by "rolling" the two dice $10,000$ times (this is specified by the parameter `num_dice_throws`). The probability of the "doubles" event can be estimated by calculating the ratio of the number of times the event occurred relative to the total number throws. You must implement this calculation within the loop that iterates over the `trials`.

The estimate of the probability of doubles will be saved in the list `trial_outcomes`, and this will be repeated `num_trials` times (where `num_trials` is set above to be 10). We're saving these trial outcomes so that we can run the overall experiment multiples times and see whether we get the same behavior.

After finishing implementation of `run_experiment` (3$), the `exercise_7` function will run the experiment two times, saving the results in `experiment_outcomes_1` and `experiment_outcomes_2`, and displaying them. You should see that the values within these sequences of 10 estimates of the probability of getting doubles vary within the sequences, and the two sequences are different.

Next, you'll again reset the seed to 7 ($), and run the experiment again. The resulting sequence, `experiment_outcomes_3`, should now be the same as `experiment_outcomes_1`.

In your written answers (in `hw1-assignment.pdf`), explain why it is often important to have random number sequences that can be controlled ($).

Solution. as seen in the homework by seeing our random number generator we can be sure of what random numbers will arise, this is helpful in cases when we want a "seemingly" random data set but perhaps we always want it to look the same for each test done. This allows us to create "random" data but access that same data again.

8. [5 points] Random vectors & matrices, and some linear algebra operations

   For this exercise you will fill in the function `exercise_8` in `hw1.py`.

   The purpose of this exercise is to use numpy's random number generation to create arrays of random numbers (that could represent vectors or matrices), and then get experience applying linear algebra operators on these, making the connection between linear algebra math notation and computational linear algebra.

   The first step is to again set the random number generator seed to 7.

   Use the numpy random number function `rand` to create a 2-d array with three rows and one column and set the value to $\mathbf{x}$. Again, use `rand` to create another 2-d array with three rows and one column, and set the value to $\mathbf{y}$ ($).

   Set the next four variables to the following values:

   - `v1` $= \mathbf{x} + \mathbf{y}$ (element-wise addition) ($)
   - `v2` $= \mathbf{x} \circ \mathbf{y}$ (element-wise multiply; Note: the notation $\mathbf{x} \circ \mathbf{y}$ is also known as the Hadamard product, the entrywise product, or the Schur product.) ($)
   - `xT` $= \mathbf{x}^\top$ (transpose x) ($)
   - `v3` $= \mathbf{x}^\top \mathbf{y}$ (also called the dot-product) ($)

   Now use numpy `rand` to create a 2-d array with three rows and three columns, and set the value to $\mathbf{A}$ ($).

   Set the next three variables to the following values:

   - `v4` $= \mathbf{x}^\top \mathbf{A}$ ($)
   - `v5` $= \mathbf{x}^\top \mathbf{A} \mathbf{y}$ ($)
   - `v6` $= \mathbf{A}^{-1}$ ($)
   - `v7` $= \mathbf{A}^{-1} \mathbf{A}$ ($)

   Mathematically, the last expression evaluates to the identity, $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. Observe that the computation is very close to the identity, but not exactly: the off diagonals are *very small* numbers, but not exactly zero. Numerical and computational linear algebra continues to be a very active research area that studies the challenges of the numerical representation of linear algebra in discrete computers.

9. [6 points] Implementing Scalar Versus Vector Math

   In this exercise, you will fill in the function `exercise_9` in `hw1.py`.

   The goal of this exercise is to directly compare the difference between implementing the same calculation using scalar and then vector math in code.

   First, load $\mathbf{X}$ and $\mathbf{w}$ from the provided files in `data/`: `X.txt` and `w.txt`, respectively ($).

   Extract the first and second column of $\mathbf{X}$ to store in `x_n1` and `x_n2`, respectively ($). There are a couple of ways to do this, but I recommend using a numpy slice operator!:
   `https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html`

   Now, using `x_n1`, `x_n2`, and `w`, use scalar arithmetic to compute the right-hand side of the equation in Exercise 3. By scalar arithmetic, we mean: iterating over a vector and squaring the values while summing them up, as opposed to using numpy to do a dot-product of the vector with itself (2$).

   In the remaining of the exercise, compute the same quantity, but this time using linear algebra operations (i.e., transpose and dot product), as shown in the left-hand side of the equation in Exercise 3. In the code, you're asked to first compute the product of the inner terms: $\mathbf{X}^\top\mathbf{X}$. After that, then apply the remaining operations by left- and right-multiplying by the $\mathbf{w}$ vector (2$).

Appendix: Numpy Arrays and Linear Algebra

Here are some notes on the relation of numpy arrays to the linear algebra math of "vectors" and "matrices" we have been using. The bottom line is that vectors are, in an important sense, just limit cases of matrices that are two-dimensional, but one of the dimensions is size 1. In all code examples I will use `numpy arrays` as our base data structure to represent vectors and matrices.

Numpy does provide matrix objects; these build on top of the numpy.ndarray data structure. While they are nice for reducing some of the verbosity of code, we can do *everything* with plain old ndarrays. (numpy.matrix objects introduce (at least when you are first getting used to them) potential confusion about what operators are being used, due to operator overloading.) In this course, for pedagogical reasons I'll stick with numpy arrays for all representation of vectors and matrices.

There are a variety of ways to create arrays. Here is a helpful overview:
`http://docs.scipy.org/doc/numpy/user/basics.creation.html`
Numpy arrays can be n-dimensional. The dimensionality of an array can be accessed by the attribute `shape`, which is represented as a tuple where each position in the tuple represents the number of indices in the corresponding dimension. Here are three arrays and their shape:

```
>>> a = numpy.array([1, 2, 3])  # This creates a 1-dimensional array
                                # with elements 1, 2, and 3
>>> a
array([1, 2, 3])
>>> a.shape
(3,)  # this shows the array is 1-d with three elements in the first dimension
>>> b = numpy.array([[1, 2, 3], [4, 5, 6]])  # This creates a 2-dimensional array
>>> b.shape                  # We see this array is two-dimensional with 2 indices
(2, 3)                       # in the first dimension and 3 in the second.
>>> b                        # We generally interpret the first dimension as row
array([[1, 2, 3],            # indices, the second as column indices.
       [4, 5, 6]])
>>> b[1, 2]       # We can use the bracket notation to index into the array;
6                 # keep in mind that python indices are 0-based, so b[1, 2] is
                  # picking out the second row, third column
>>> c = numpy.array([[1], [2], [3]])  # this defines a 2-d array
>>> c.shape
(3, 1)
>>> c
array([[1],
       [2],
       [3]])
```

We can create higher dimensional arrays by nesting more lists specifying elements, but for representing vectors and matrices, we'll stick to 1 and 2-dimensional arrays.

Now for the connection with the linear algebra. It turns out that 1-d arrays are equivalent to linear algebra column vectors (as I refer to them in the course lectures): as noted above in the comments, the convention is that the first dimension (even for 1-d arrays) is interpreted as indexing the *rows* of an array – so all 1-d array indices can be interpreted as spread across rows – i.e., their natural interpretation is as a column vector. One thing that is a little odd is that taking the transpose of a 1-d numpy array has no effect: there is only one dimension so the transpose cannot swap dimensions – this is where the programming with numpy is a little different from the mathematical concept of vector with an "orientation" (row or column). For 2-d arrays, however, we can explicitly transpose, and therefore we can make an explicit distinction between column and row vectors (as in c vs. c1, below):

```
>>> a1 = a.T  # This "transposes" the 1-d array, which does nothing
>>> a1
array([1, 2, 3])  # we can see a1 is the same as the original a
>>> a1.shape
(3,)  # this shows the transpose of a is still a 1-d array,
      # safely interpreted as a vector with three rows: a column vector
>>> b1 = b.T  # This transposes the b 2-d array
>>> b1.shape                # We now see this array is two-dimensional with 3 rows
(3, 2)                      # and 2 columns (i.e., 2 elements per row)
>>> b1
array([[1, 4],      # and this shows the elements have indeed been transposed
       [2, 5],
       [3, 6]])
>>> c1 = c.T  # now we transpose c, which was a 2-d array
>>> c1.shape
(3, 1)   # c1 now has 1 row with 3 columns.
>>> c1   # This is most naturally interpreted as an explicit row vector
array([[1, 2, 3]])
```

You will see in the provided code that I generally follow the (popular) convention that as long as I am only working with a column vector that does not need to be transposed into a row vector, I will use a 1-d array. IF, however, I know that I will need to switch between column and row vector forms, then I will use the explicit 2-d array as in c and c1 (where one of the dimensions has only 1 index).

Here is a handy trick for converting a 1-d array into an explicit 2-d array as a column vector:

```
>>> a.shape    # recall that a is a 1-d array (with 3 row indices)
(3, )
>>> d = numpy.array(a[:, numpy.newaxis])  # The ':' is the slice operator
>>> d.shape     # The numpy.newaxis allows us to add a new axis
(3, 1)          # (1 more dimension) to our array object ... which just
>>> d           # makes explicit that our previously "implicit" column
array([[1],     # vector is now an explicit column vector, which can then
       [2],     # be transposed...
       [3]])
```

The creation of the array d is built from indexing into the vector a, using a `slice` operator to refer to all elements along the first dimension, and then effectively adding a second dimension (of size 1) using the numpy.newaxis as the specifier to the second dimension index.

See here for more information about indexing:
`http://docs.scipy.org/doc/numpy/user/basics.indexing.html`
and
`http://docs.scipy.org/doc/numpy/reference/arrays.indexing.html`

You will also want to look at the numpy linalg package for linear algebra operators (esp. for the `dot` and `inv` operators):
`http://docs.scipy.org/doc/numpy/reference/routines.linalg.html`