Homework 6: `numpy` and `matplotlib`

Due March 24, 11:59 pm

Worth 10 points

Instructions on writing and submitting your homework can be found on the course webpage at `http://pages.stat.wisc.edu/~kdlevin/teaching/Spring2023/STAT606/hw_instructions.html`. *Failure to follow these instructions will result in lost points.* Please direct any questions the instructor.

1 Warmup: Around the Semicircular Law (3 points)

The (comparatively) young field of random matrix theory (RMT) concerns the behavior of certain matrices with independent random entries. A landmark result in RMT concerns the behavior of the eigenvalues of a random symmetric matrix with normal entries. Under the proper scaling, the joint distribution of the eigenvalues of such a matrix follows the *Wigner semicircular distribution*, which has density

$$f(x) = \begin{cases} \frac{\sqrt{4-x^2}}{2\pi} & \text{if } -2 \le x \le 2 \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

That is, a symmetric matrix with random normal entries will have eigenvalues whose histogram looks more and more like a semicircle of radius 2 as n increases to ∞. In particular, define a matrix-valued random variable $Z \in \mathbb{R}^{n\times n}$ by generating $Z_{i,j}$ i.i.d. normal with mean 0 and variance $1/n$ for all $1 \le i \le j \le n$, and set $Z_{j,i} = Z_{i,j}$ for $1 \le j \le i \le n$. Then the matrix $Z \in \mathbb{R}^{n\times n}$ is called a *Wigner matrix.*

1. Define a function `wigner_density` that takes a single number (integer or float) as its input and returns a float as its output, given by the value of the semicircular density evaluated at the input. That is, for a number `x`, `wigner_density(x)` should return $f(x)$, where f is defined above in Equation (1). You do not need to perform any error checking in this function, but note that your function should operate equally well on Python ints/floats and on `numpy` ints/floats, and you should be able to accomplish this without checking the type of the input. Use the `numpy.sqrt` function for the square root, *not* the Python `math.sqrt` function.

2. Define a function `generate_wigner` that takes a single positive integer `n` as its argument and returns a random n-by-n Wigner matrix. Your function should raise an appropriate error in the event that the input is not a positive integer. The output of your function may be either a `numpy` matrix or simply a `numpy` array, though I would recommend the former, for ease of use in the next subproblem. You can cast a 2-dimensional `numpy` array `a` to a matrix by writing

np.matrix(a). **Hint:** depending on the solution you choose, you may find the numpy.triu and numpy.tril functions to be useful. A different solution makes use of the scipy.spatial.distance.squareform function.

3. The RMT result referenced above states that the joint distribution of the eigenvalues of a random Wigner matrix converges to the semicircular law. Write a function get_spectrum that takes a numpy matrix or 2-dimensional numpy array and returns a numpy array of its eigenvalues in non-decreasing order.[1] You do not need to perform any error checking for this function.

4. Create a plot with four subplots, arranged vertically, each showing a (normalized) histogram, in blue, of the eigenvalues of a random n-by-n Wigner matrix for $n = 100, 200, 500$ and 1000. In each subplot, overlay a red curve indicating the density of the semicircular law, as defined in (1). Save this plot in a .png file called wigner_plots.png and include it in your submission. **Hint:** depending on how you implemented wigner_density above, you may find the numpy.vectorize function helpful.

   At least based on looking at the plots, how big does n have to be before the semicircular law appears to be a good fit? In practice, we would answer this question more rigorously with, for example, a Kolmogorov-Smirnov test, which you can find in the scipy.stats module, but that is entirely optional. **Note:** this experiment involves some matrix eigenvalue computations, which are comparatively expensive. If you set n larger than about 5000, be prepared to wait a few minutes for your answer, especially if you are running on an older machine or on a VM.

2 Plotting a Mixture of Normals (3 points)

The whole reason that we use plotting software is to visualize the data that we are working with, so let's do that. The zip file located at http://pages.stat.wisc.edu/~kdlevin/teaching/Spring2023/STAT606/hw/hw6_files.zip contains two files, storing data from a simulation experiment in my own research. The file points.dlm is a tab-delimited file (.dlm stands for "delimited"). Such a format is common when writing reasonably small files, and is useful if you expect to use a data set across different programs or platforms. See the documentation for the command numpy.loadtxt to see how to read this file. The file labels.npy is a numpy binary file, representing a numpy object. The .npy file format is specific to numpy. Many languages (e.g., R and MATLAB) have their own such language-specific file formats for saving variables, workspaces, etc. These formats tend to be more space-efficient, typically at the cost of program-dependence. It is best to avoid such files if you expect to deal with the same data set in several different environments (e.g., you run experiments in MATLAB and do your statistical analysis in R). .npy files are opened using numpy.load.

The observations in my experiment were generated from a distribution that is *approximately* a mixture of normals, but not precisely so. Let's explore how well the normal approximation holds.

1. Download the .zip file, extract it, and read the two files into numpy. Please include both labels.npy and points.dlm in your final submission. The former of these

[1] You may find the following documentation useful: https://numpy.org/doc/stable/reference/generated/numpy.linalg.eigh.html

should yield a `numpy` array of 0s and 1s, and the latter should yield a 100-by-2 `numpy` array, in which each row corresponds to a 2-dimensional point. Each of these 2-dimensional points is assigned to one of two clusters. The i-th entry of the array in `labels.npy` encodes this the cluster membership label of the point in the i-th row of the matrix stored in `points.dlm`.

2. Generate a scatter plot of the data. Each data point should appear as an `x` (often called a *cross* in data visualization packages), colored according to its cluster membership as given by `points.npy`. The points with cluster label 0 should be colored blue, and those with cluster label 1 should be colored red. Set the x- and y- axes to both range from 0 to 1. Adjust the size of the point markers to what you believe to be reasonable (i.e., aesthetically pleasing, visible, etc.).

3. Theoretically, the data should approximate a mixture of normals with means and covariance matrices given by

$$\mu_0 = (0.2, 0.7)^T, \Sigma_0 = \begin{bmatrix} 0.015 & -0.011 \\ -0.011 & 0.018 \end{bmatrix},$$

$$\mu_1 = (0.65, 0.3)^T, \Sigma_1 = \begin{bmatrix} 0.016 & -0.011 \\ -0.011 & 0.016 \end{bmatrix}.$$

   For each of these two normal distributions, add two contour lines corresponding to 1 and 2 “standard deviations” of the distribution. We will take the 1-standard deviation contour to be the level set (which is an ellipse) of the normal distribution that encloses probability mass 0.68 of the distribution, and the 2-standard deviation contour to be the level set that encloses probability mass 0.95 of the distribution. The contour lines for cluster 0 should be colored blue, and the lines for cluster 1 should be colored red. The contour lines might go off the edge of the 1-by-1 square that we have plotted. Do not worry about that. **Hint:** these ellipses are really just confidence regions given by

$$(x - \mu)^T \Sigma^{-1} (x - \mu) \leq \chi_2^2(p),$$

   where p is a probability and χ_d^2 is the quantile function for the χ^2 distribution with d degrees of freedom. **Hint:** use the optional argument `levels` for the `pyplot.contour` function. The quantile function for the χ^2 distribution is available in `scipy.stats.chi2`.

4. Do the data appear normal? There should be at least one obvious outlier. Add an annotation to your figure indicating one or more such outlier(s). **Note:** as with many things in statistics, what does or does not constitute an outlier is ultimately a judgment call. There is not, strictly speaking, a right or wrong answer to this. Simply annotate your figure to indicate which points appear to you to be outliers.

3 Conway’s Game of Life (4 points)

Conway’s Game of Life [2] is a classic example of a *cellular automaton* devised by mathematician John Conway. The game is a classic example of how simple rules can give rise to complex behavior. The game is played on an m-by-n board, which we will represent

[2] `https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`

as an m-by-n matrix. The game proceeds in steps. At any given time, each cell of the board (i.e., entry of our matrix), is either alive (which we will represent by the Boolean `True`) or dead (which we will represent by the Boolean `False`). At each step, the board evolves according to a few simple rules:

- A live cell with fewer than two live neighbors becomes a dead cell.
- A live cell with more than three live neighbors becomes a dead cell.
- A live cell with two or three live neighbors remains alive.
- A dead cell with *exactly* three live neighbors becomes alive.
- All other dead cells remain dead.

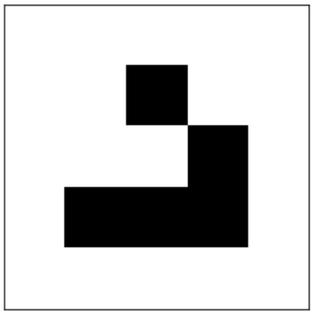
The neighbors of a cell are the 8 cells adjacent to it, i.e., left, right, above, below, upper-left, lower-left, upper-right and lower-right. We will follow the convention that the board is *toroidal*, so that using matrix-like notation (i.e., the cell $(0, 0)$ is in the upper-left of the board and the first coordinate specifies a row), the upper neighbor of the cell $(0, 0)$ is $(m - 1, 0)$, the right neighbor of the cell $(m - 1, n - 1)$ is $(m - 1, 0)$, etc. That is, the board "wraps around". This definition gets a bit odd when the board is smaller then three-by-three. You may simply ignore that case in what follows, if you wish. It will not be included when testing your code. **Note:** you are not required to use this matrix-like indexing. It's just what I chose to use to explain the toroidal property.

1. Write a function `is_valid_board` that takes a single argument and returns a Python Boolean that is `True` if and only if the argument is a valid representation of a Game of Life board. A valid board is any two-dimensional numpy `ndarray` whose entries are all numpy Booleans (i.e., `numpy.bool_`). Note that your function should accept, e.g., a two-by-two board as valid, even if the definition for generating the next step in such a board is a bit odd.

2. Write a function called `gol_step` that takes an m-by-n `numpy` array as its argument and returns another `numpy` array of the same size (i.e., also m-by-n), corresponding to the board at the next step of the game. Your function should perform error checking to ensure that the provided argument is a valid Game of Life board.

3. Write a function called `draw_gol_board` that takes an m-by-n `numpy` array (i.e., an `ndarray`) as its only argument and draws the board as an m-by-n set of tiles, colored black or white correspond to whether the corresponding cell is alive or dead, respectively. Your plot should *not* have any grid lines, nor should it have any axis labels or axis ticks. **Hint:** see the functions `plt.xticks()` and `plt.yticks()` for changing axis ticks. **Hint:** you may find the function `plt.get_cmap` to be useful for working with the `matplotlib Colormap` objects.

4. Create a 20-by-20 numpy array corresponding to a Game of Life board in which all cells are dead, with the exception that the top-left 5-by-5 section of the board looks like this:



Plot this 20-by-20 board using `draw_gol_board`, save this plot in `.png` file called `glider_board.png`, and include it in your submission.

5. Generate a plot with 5 subplots, arranged in a 5-by-1 grid, showing the first five steps of the Game of Life when started with the board you just created, with the steps ordered from top to bottom. The first plot should be the starting board just described, so that the starting board is the first "step". The figure in the 5-by-5 sub-board above is called a *glider*, and it is interesting in that, as you can see from your plot, it seems to move along the board as you run the game. Save the resulting plot in a `.png` file called `glider_5steps.png`, and include it in your submission.

Optional additional exercise: create a function that takes two arguments, a Game of Life board and a number of steps, and generates an animation of the game as it runs for the given number of steps. **Note:** this is an optional exercise. It is not worth any points, and does not count for any bonus points.