Homework 6: `numpy` and `matplotlib`

Due March 7, 11:59 pm

Worth 10 points

Read this first. A few things to bring to your attention:

1. **Important:** If you have not already done so, please request a Flux Hadoop account. Instructions for doing this can be found on Canvas.
2. Start early! If you run into trouble installing things or importing packages, it's best to find those problems well in advance, not the night before your assignment is due when we cannot help you!
3. **Make sure you back up your work!** I recommend, at a minimum, doing your work in a Dropbox folder or, better yet, using `git`, which is well worth your time and effort to learn.

Instructions on writing and submitting your homework.

Failure to follow these instructions will result in lost points. Your homework should be written in a jupyter notebook file. I have made a template available on Canvas, and on the course website at `http://www-personal.umich.edu/~klevin/teaching/Winter2018/STATS701/hw_template.ipynb`. You will submit, via Canvas, a `.zip` file called `yourUniqueName_hwX.zip`, where `X` is the homework number. So, if I were to hand in a file for homework 1, it would be called `klevin_hw1.zip`. Contact the instructor or your GSI if you have trouble creating such a file.

When I extract your compressed file, the result should be a directory, also called `yourUniqueName_hwX`. In that directory, at a minimum, should be a jupyter notebook file, called `yourUniqueName.hwX.ipynb`, where again `X` is the number of the current homework. You should feel free to define supplementary functions in other Python scripts, which you should include in your compressed directory. So, for example, if the code in your notebook file imports a function from a Python file called `supplementary.py`, then the file `supplementary.py` should be included in your submission. In short, I should be able to extract your archived file and run your notebook file on my own machine. Please include all of your code for all problems in the homework in a single Python notebook unless instructed otherwise, and please include in your notebook file a list of any and all people with whom you discussed this homework assignment. Please also include an estimate of how many hours you spent on each of the three sections of this homework assignment.

These instructions can also be found on the course webpage at `http://www-personal.umich.edu/~klevin/teaching/Winter2018/STATS701/hw_instructions.html`. Please direct any questions to either the instructor or your GSI.

1 Warmup: plotting CLTs (2 points)

Let random variables $X_1, X_2, \ldots, X_n$ be independently identically distributed with mean $\mu = \mathbb{E}X_1$ and finite variance $\sigma^2 = \mathbb{E}X_1^2 < \infty$, and denote their sample mean by $S_n = (X_1 + X_2 + \cdots + X_n)/n$. The classical central limit theorem states that as $n \to \infty$, $\sqrt{n}(S_n - \mu)$ converges in distribution (also called "in law") to a normal random variable with mean 0 and variance σ^2. A natural question to ask is whether the distribution of the X_i has an effect on how quickly $\sqrt{n}(S_n - \mu)$ starts to look like a normal. Let's explore this phenomenon.

Choose four different probability distributions with mean 0 and variance 1. For example, I might choose standard normal, $(\text{Bern}(p) - p)/\sqrt{p(1-p)}$, $\text{Poisson}(1) - 1$ and $\sqrt{12}(\text{Unif}(0,1) - 1/2)$ (you can check that all of these are mean 0 and variance 1), but feel free to choose more interesting or exotic distributions! See `https://docs.scipy.org/doc/scipy/reference/stats.html` for a list of good choices.

1. Pick a reasonably large value of n (say, $n = 20$). Make a plot with four subplots, one for each of the four distributions. For each subplot, use Numpy/Scipy to generate $m = 1000$ independent draws of $\sqrt{n}(S_n - \mu)$ from that subplot's distribution and show in that subplot a (normalized) histogram of the empirical distribution of $\sqrt{n}(S_n - \mu)$.

2. Title each of your subplots with the name of the distribution you used for that subplot. Make sure that all four of your plots have the same scales on their x- and y-axes.

3. Add to each subplot a solid line indicating the density of the standard normal, so that we can see how "normal" the empirical distribution looks (you will need to import the `scipy.stats` module for this: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html#scipy.stats.norm`).

4. Do all four of your plots look equally "normal"? Note that it may help to try a few different values of n to explore differences between the plots, depending on which four distributions you picked.

2 Plotting a Mixture of Normals (4 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 `www-personal.umich.edu/~klevin/teaching/Winter2018/STATS701/hw6_files.zip` contains two files, storing data from an 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`.

These observations were generated from a distribution that is *approximately* normal, 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 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. The i-th entry of the array in `labels.npy` corresponds to the cluster membership label of 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 will 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 *chi*^{2} distribution with d degrees of freedom. **Hint:** use the optional argument `levels` for the `pyplot.contour` function.

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).

3 Conway's Game of Life (4 points)

Conway's Game of Life (`https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`) 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 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 as a 1) or dead (which we will represent as a 0). 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.

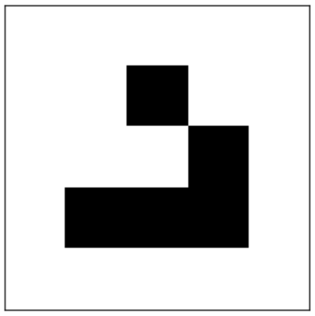
The neighbors of a cell are the 8 cells adjacent to it, i.e., left, right, above, below, upper-left, lower-left, upper-right and lower-right. We will follow the convention that the board is *toroidal*, so that using matrix-like notation (i.e., the cell $(0,0)$ is in the upper-left of the board and the 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". **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 an m-by-n `numpy` array (i.e., an `ndarray`) as its only 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` with all entries either 0.0 and 1.0.

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 100-by-100 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 100-by-100 board using `draw_gol_board`.

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 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.

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.