hw5

December 26, 2019

0.1 STATS 507 HW5 numpy and matplotlib

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import scipy.stats
        %matplotlib inline
```

0.1.1 Problem 1 Warmup: Around the Semicircular Law

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

$$f(x) = \begin{cases} \frac{\sqrt{4-x^2}}{2\pi} & if -2x2 \quad 0 \quad otherwise \end{cases}$$

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 ZR_{nn} by generating $Z_{i,j}$ i.i.d. normal with mean 0 and variance $1/n$ for all $1ijn$, and set $Z_{j,i} = Z_{i,j}$ for $1jin$. Then the matrix ZR_{nn} 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.

```
In [2]: def wigner_density(x):
            if x >= 2 or x <= 2:
                return np.sqrt(4-x**2)/(2*np.pi)
            else:
                return 0
```

```
In [3]: wigner_density(1)
```

```
Out[3]: 0.27566444771089604
```

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 an integer or if it is not positive. The output of your function may be either a numpy matrix or simply a numpy array. I would slightly 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.

```
In [29]: def generate_wigner(n):
             if type(n) != int:
                 raise TypeError
             if n <= 0:
                 raise ValueError
             else:
                 x = np.random.normal(0, np.sqrt(1/n), (n,n))
                 lower = np.tril(x, k = -1)
                 diag = np.diag(np.diagonal(x))
                 upper = np.transpose(lower)
                 wigner = lower + diag + upper
                 wigner = np.matrix(wigner)
                 return wigner
```

```
In [31]: x = generate_wigner(500)
```

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. You do not need to perform any error checking for this function.

   You fill find the following documentation useful: https://docs.scipy.org/doc/numpy/reference/generated

```
In [32]: def get_spectrum(m):
             eigenvalue, eigenvector = np.linalg.eigh(m)
             return eigenvalue
```

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).

   Hint: depending on how you implemented wigner_density above, you may find the numpy.vectorize function helpful.

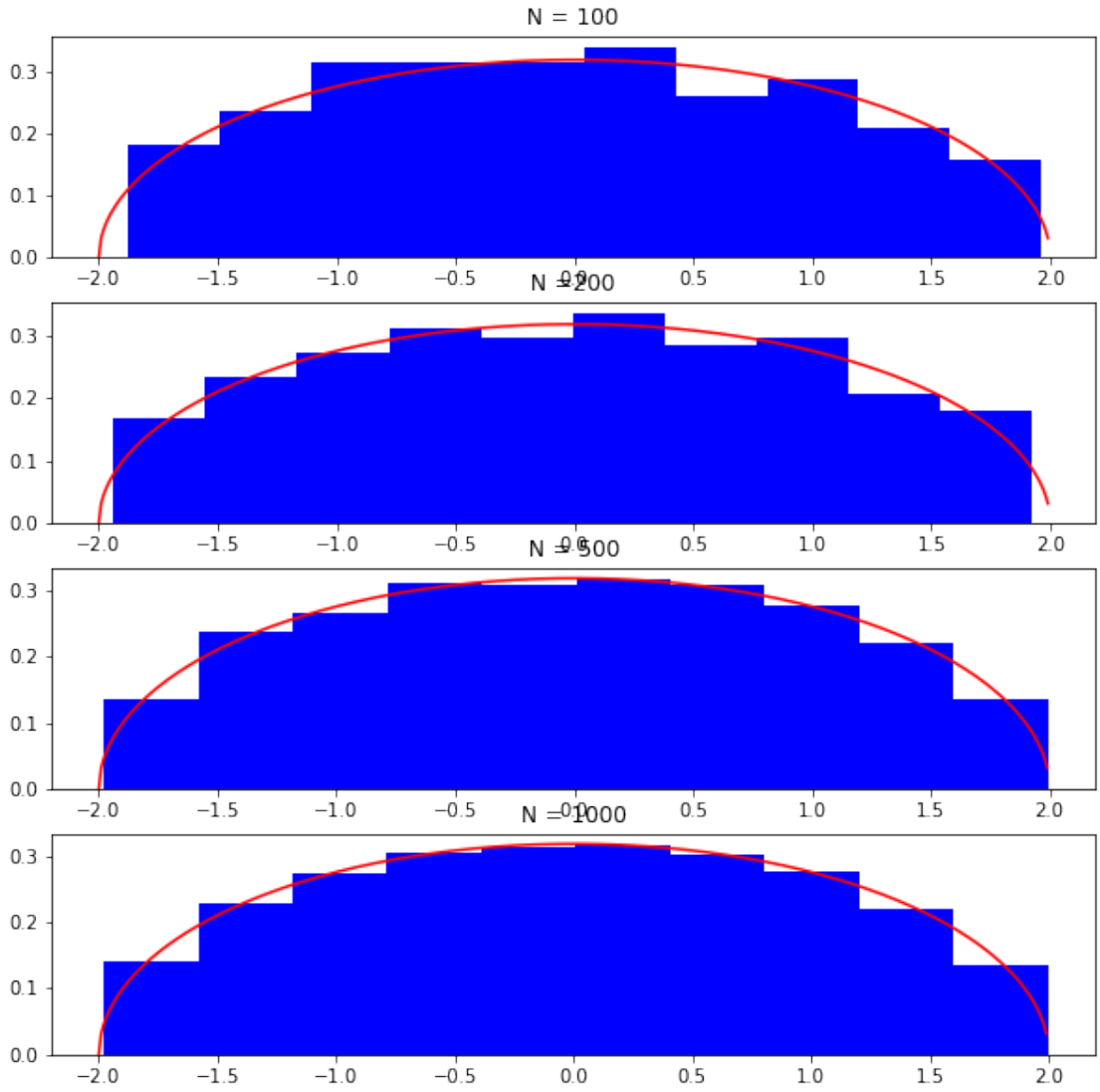
How big does n have to be before the semicircular law appears to be a good fit? Of course, in practice, we would answer this question more rigorously with, for example, a Kolmogorov-Smirnov test, which you can find in the scipy.stats module, but that is entirely optional.

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 a laptop.

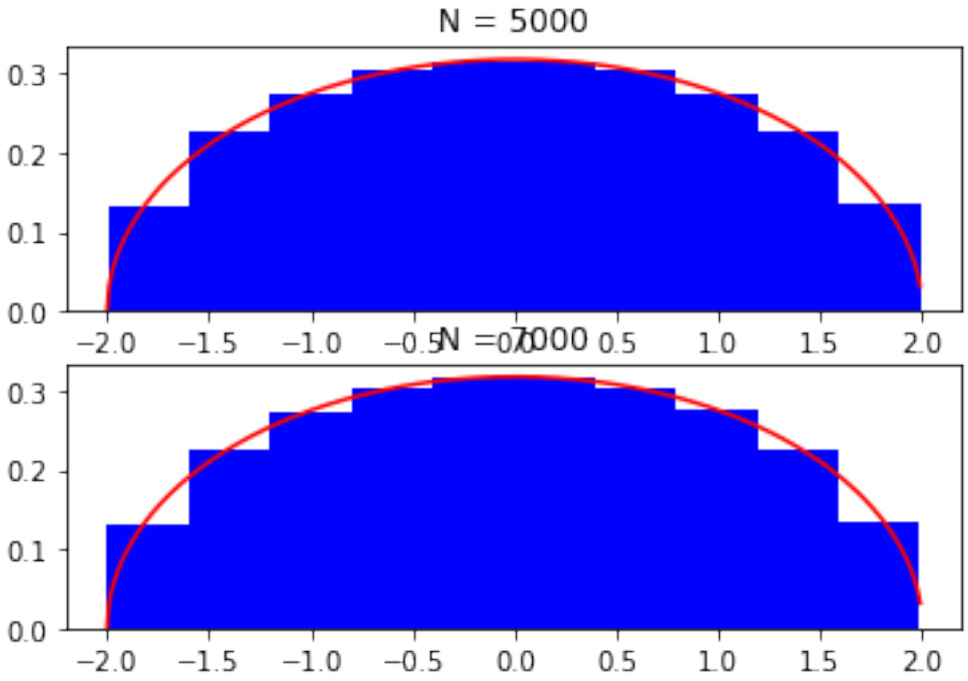
In [7]:
```
# density curve
x = np.arange(-2, 2, 0.01)
density = np.vectorize(wigner_density) # element-wise multiplication
y = density(x)
```

In [8]:
```
fig = plt.figure(figsize=(10,10))
fig.add_subplot(411)
plt.hist(get_spectrum(generate_wigner(100)), color = 'blue', density = True)
plt.plot(x, y, color = 'red')
plt.title('N = 100')

fig.add_subplot(412)
plt.hist(get_spectrum(generate_wigner(200)), color = 'blue', density = True)
plt.plot(x, y, color = 'red')
plt.title('N =200')

fig.add_subplot(413)
plt.hist(get_spectrum(generate_wigner(500)), color = 'blue', density = True)
plt.plot(x, y, color = 'red')
plt.title('N = 500')

fig.add_subplot(414)
plt.hist(get_spectrum(generate_wigner(1000)), color = 'blue', density = True)
plt.plot(x, y, color = 'red')
plt.title('N = 1000')
```

Out[8]:
```
Text(0.5, 1.0, 'N = 1000')
```




```
In [9]: # setting N = 5000
        fige = plt.figure()
        fige.add_subplot(211)
        plt.hist(get_spectrum(generate_wigner(5000)), color = 'blue', density = True)
        plt.plot(x, y, color = 'red')
        plt.title('N = 5000')

Out[9]: Text(0.5, 1.0, 'N = 7000')
```



N = 5000 is large enough for eigenvalues to form a semicircle.

0.1.2 Problem 2 Plotting a Mixture of Normals

The whole reason that we use plotting software is to visualize the data that we are work- ing with, so let's do that. The zip file located at www-personal.umich.edu/~klevin/teaching/Winter2019/STATS507/hw5_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.

The observations in my experiment 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.

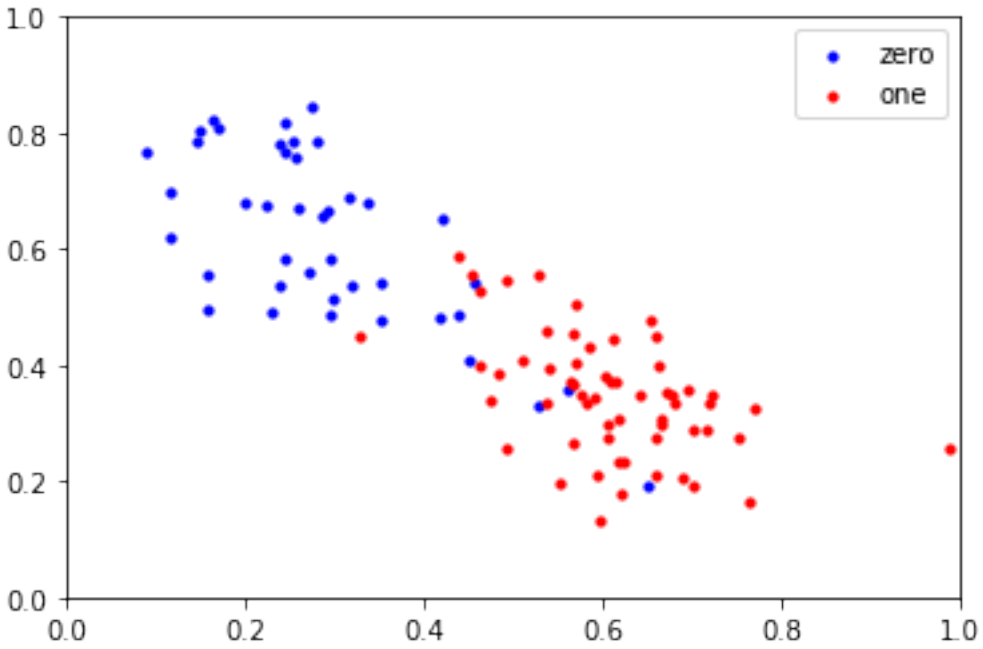
The i-th entry of the array in labels.npy corresponds to the cluster membership label of the i-th row of the matrix stored in points.dlm.

```
In [10]: label = np.load('labels.npy')  # 100x1  (0,1 entries)
         relabel = np.reshape(label, (100,1))
         point = np.loadtxt('points.dlm') # 100x2
```

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).

```
In [11]: # subsetting data to cluster 0 and 1
         full = np.append(point, relabel, axis=1)
         zero = full[full[:,2]==0]
         one = full[full[:,2]==1]
```

```
In [12]: fig = plt.figure()
         plt.xlim([0,1])
         plt.ylim([0,1])
         plt.scatter(zero[:,0], zero[:,1], color = 'blue', label = 'zero', s = 10)
         plt.scatter(one[:,0], one[:,1], color = 'red', label = 'one', s = 10)
         _ = plt.legend(loc = 'best')
```



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

$$(x - \mu)^T \Sigma^{-1} (x - \mu) \leqslant X_2^2(p)$$

where p is a probability and X_d^2 is the quantile function for the X^2 distribution with d degrees of freedom.

Hint: use the optional argument levels for the pyplot.contour function.

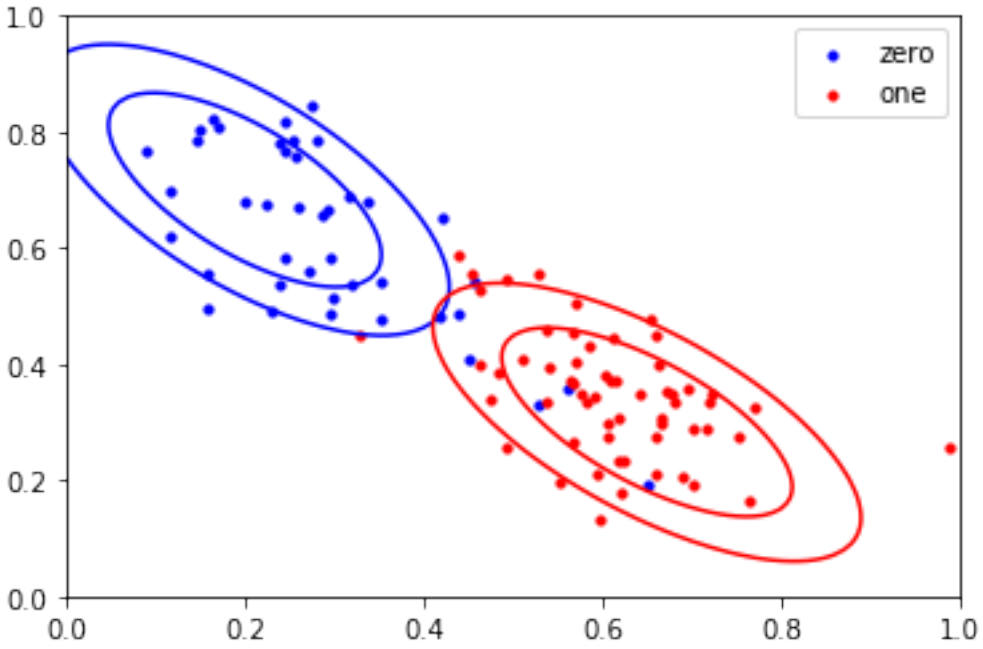
```
In [13]: # find chi-square score
         from scipy.stats import chi2
         chi2.ppf([0.68, 0.95], df = 2)
```

```
Out[13]: array([2.27886857, 5.99146455])
```

```
In [14]: from scipy import linalg
         mu0 = np.array([0.2, 0.7])
         mu1 = np.array([0.65, 0.3])
         sigma0 = np.array([[0.015, -0.011], [-0.011, 0.018]])
         sigma1 = np.array([[0.016, -0.011], [-0.011, 0.016]])
```

```
In [15]: x, y = np.mgrid[0:1:.01, 0:1:.01]
         pos = np.empty(x.shape + (2,))
         pos[:, :, 0] = x; pos[:, :, 1] = y
         mvn1 = scipy.stats.multivariate_normal(mu0,sigma0)
         mvn2 = scipy.stats.multivariate_normal(mu1,sigma1)

         fig3 = plt.figure()
         plt.xlim([0,1])
         plt.ylim([0,1])
         plt.scatter(zero[:,0], zero[:,1], color = 'blue', label = 'zero', s = 10)
         plt.scatter(one[:,0], one[:,1], color = 'red', label = 'one', s = 10)
         plt.legend(loc = 'best')
         plt.contour(x, y, mvn1.pdf(pos), levels = [2.27886857, 5.99146455], colors = 'blue')
         plt.contour(x, y, mvn2.pdf(pos), levels = [2.27886857, 5.99146455], colors = 'red')
```

Out[15]: <matplotlib.contour.QuadContourSet at 0x112631940>

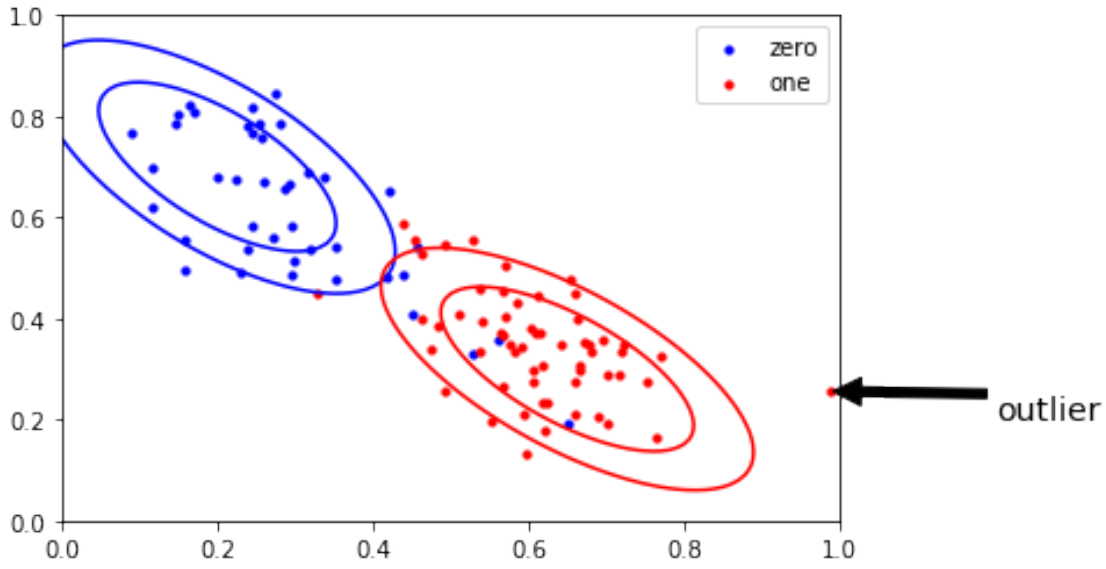


4. Do the data appear normal? There should be at least one obvious outlier. Add an annotation to your figure indicating one or more such outlier(s).

one outlier exist in the red zone.

```
In [16]: fig4 = plt.figure()
         plt.xlim([0,1])
         plt.ylim([0,1])
         plt.scatter(zero[:,0], zero[:,1], color = 'blue', label = 'zero', s = 10)
         plt.scatter(one[:,0], one[:,1], color = 'red', label = 'one', s = 10)
         plt.legend(loc = 'best')
         plt.contour(x, y, mvn1.pdf(pos), levels = [2.27886857, 5.99146455], colors = 'blue')
         plt.contour(x, y, mvn2.pdf(pos), levels = [2.27886857, 5.99146455], colors = 'red')
         plt.annotate('outlier', xy = (0.98788763,0.25612301), xytext = (1.2, 0.2), fontsize =
                      arrowprops = dict(facecolor = 'black', shrink = 0.02))
```

```
Out[16]: Text(1.2, 0.2, 'outlier')
```



0.1.3 Problem 3 Conway's Game of Life

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.

```
In [33]: def is_valid_board(a):
             # CHECK DIM, TYPE NDARRAY
```

```
            if type(a) != np.ndarray:
                return False
            if len(a.shape) != 2:
                return False
            for i in a:
                for j in i:
                    if not (j==1.0 or j==0.0):
                        return False
            return True
```

```
In [34]: a = np.array([[0,1,0,1], [1,0,0,1], [1,1,0,0]])
         is_valid_board(a)
```

```
Out[34]: True
```

```
In [19]: b = np.array([[0,1,1,1], [0,0,0,3], [1,1,0,0]])
         is_valid_board(b)
```

```
Out[19]: False
```

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.

```
In [20]: def gol_step(a):
             rows = np.shape(a)[0]
             cols = np.shape(a)[1]
             new_a = np.zeros(shape = (rows, cols))
             if is_valid_board(a) == False:
                 raise TypeError
             else:
                 for row in range(rows):       # iterate over row
                     for col in range(cols):   # iterate over column
                     # a[row][col] is the target cell
                     # iterate the neighbor surround center
                         total = 0
                         for i in range(-1, 2):
                             for j in range(-1, 2):

                                 if not (i == 0 and j == 0):
                                     total += a[((row + i) % rows)][((col + j) % cols)]
                         #print(total)
                         #print("({} {}): {}".format(row, col, total))
                         if total < 2 or total > 3:
                             new_a[row][col] = 0
                         elif total == 3 and a[row][col] == 0:
                             new_a[row][col] = 1
```

```
                    else:
                        new_a[row][col] = a[row][col]
        return new_a
```

```
In [21]: x = np.array([[0, 1, 0, 0, 0],
          [1, 0, 0, 1, 0],
          [1, 0, 0, 0, 1],
          [0, 1, 0, 0, 0],
          [0, 0, 0, 0, 0]])
         y = gol_step(x)
```

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.

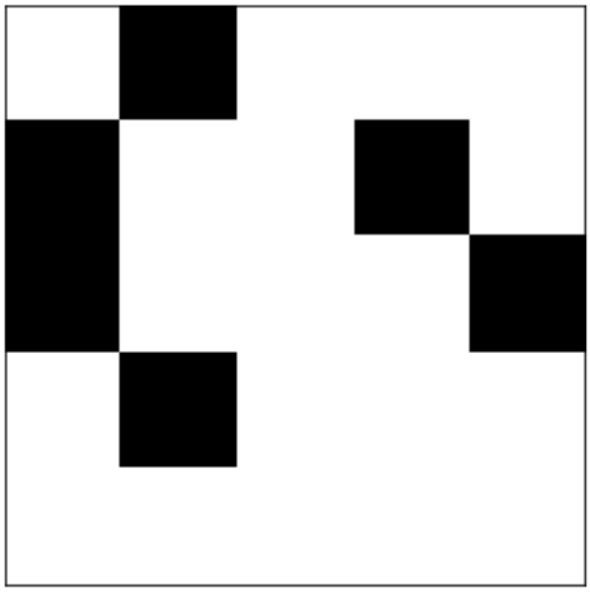
   Hint: you may find the function plt.get_cmap to be useful for working with the matplotlib Colormap objects.

```
In [22]: def draw_gol_board(a):
             plt.imshow(a, cmap='binary')
             plt.xticks([])
             plt.yticks([])
             return plt.show()
```

```
In [23]: x
```

```
Out[23]: array([[0, 1, 0, 0, 0],
                [1, 0, 0, 1, 0],
                [1, 0, 0, 0, 1],
                [0, 1, 0, 0, 0],
                [0, 0, 0, 0, 0]])
```

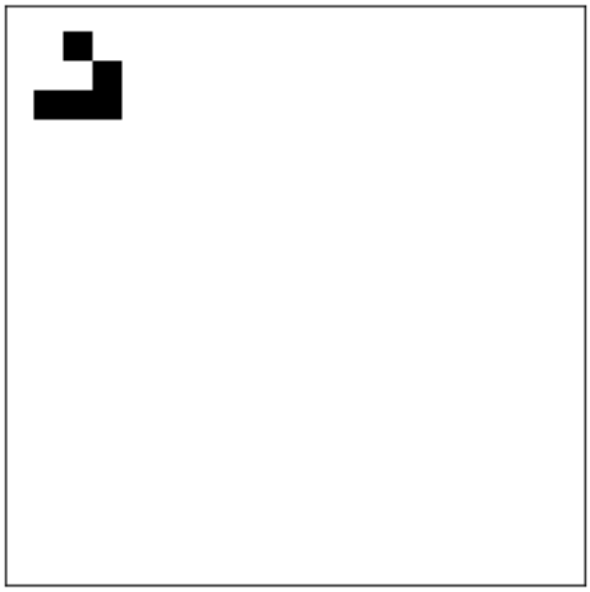
```
In [24]: board = draw_gol_board(x)
```



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:

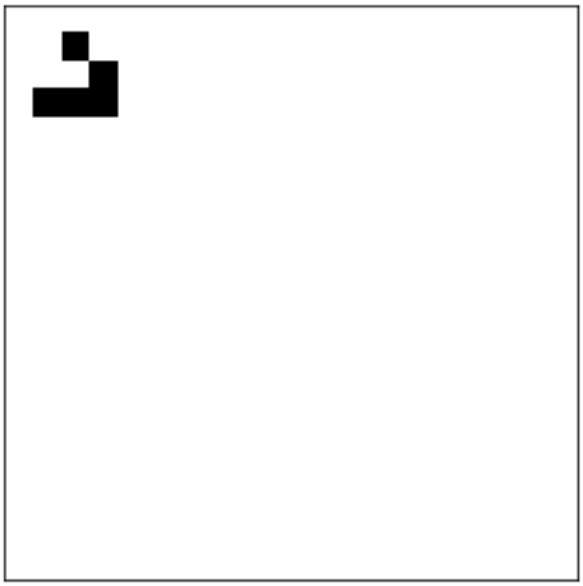
   Plot this 20-by-20 board using draw_gol_board.

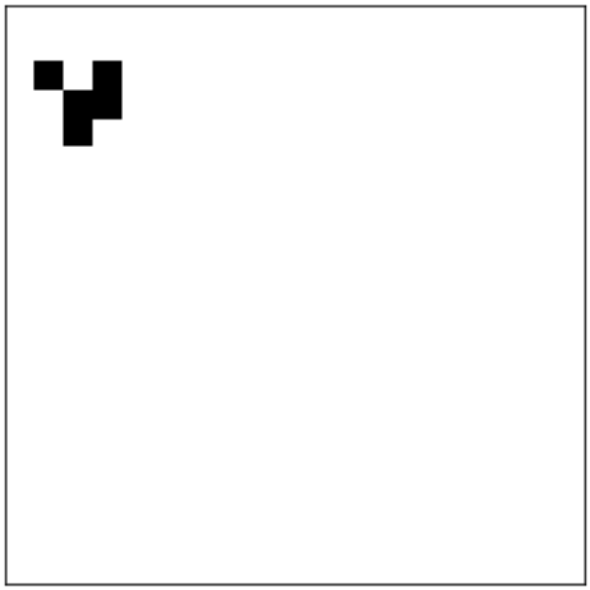
```
In [25]: a = np.zeros(shape = (20,20))
         a[1,2] = 1
         a[3,1] = 1
         a[3,2] = 1
         a[2,3] = 1
         a[3,3] = 1

In [26]: draw_gol_board(a)
```

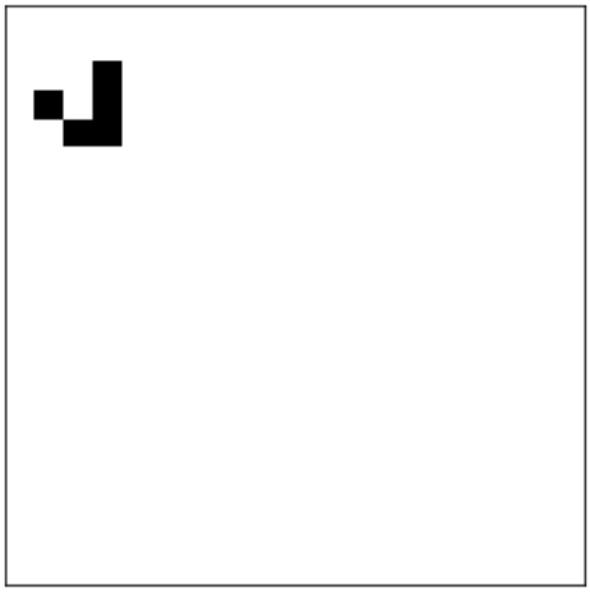


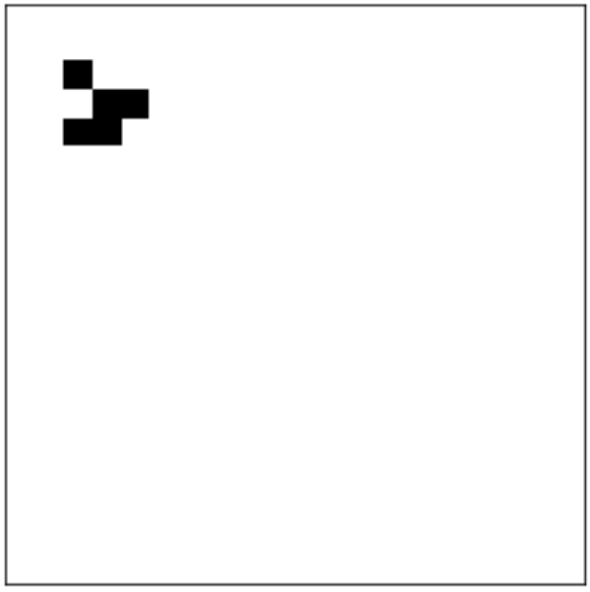
5. Generate a plot with 5 subplots, arranged in a 5-by-1 grid, showing the first five steps of the Game of Life when started with the board you just created, with the steps ordered from top to bottom, The figure in the 5-by-5 sub-board above is called a glider, and it is interesting in that, as you can see from your plot, it seems to move along the board as you run the game.

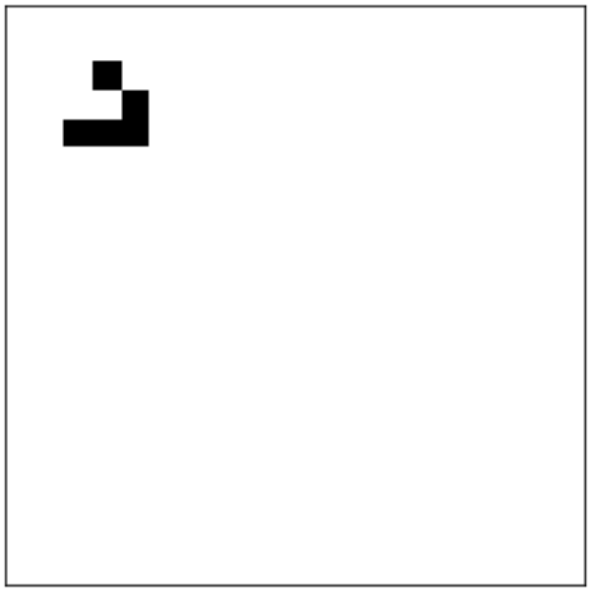
```
In [27]: b = gol_step(a)
         c = gol_step(b)
         d = gol_step(c)
         e = gol_step(d)

In [28]: fig = plt.figure(figsize=(23,23))

         fig.add_subplot(511)
         draw_gol_board(a)

         fig.add_subplot(512)
         draw_gol_board(b)

         fig.add_subplot(513)
         draw_gol_board(c)

         fig.add_subplot(514)
         draw_gol_board(d)

         fig.add_subplot(515)
         draw_gol_board(e)
```











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.