Describing Code for the Subsequent Projects

Description

The purpose of this document is to explain all of the code found throughout the repository that isn't covered by the docstrings. Namely, much of the `.py` functions lack explanations as they exist to streamline code throughout the different notebooks. As such, each subsequent section here will represent a different `.py` file, including its unit testing.

Code

generate_network.py

The only item here, the class, is here to store all of the network properties and to include its basic methods, mostly that of rewiring. Of course, the imports come first:

```
from numpy import append, argwhere, arange, array, diag, delete, eigh, ones, random, sum, triu
from matplotlib import pyplot as plt
import networkx as nx
```

The purpose of importing `numpy` in portions like these is to reduce the time it takes to begin the notebooks. Namely, importing the entirety of the library takes time and space, so reducing it down to the only essential functions works wonderfully. `numpy` is used primarily for its vector operations and data structures.

Further, `matplotlib` has the same treatment, by only importing a subsection in `matplotlib.pyplot`. There are many plotting functions, so trying to reduce this down to only a few would pose its own challenge. The same can be said for `networkx`, so importing them in bulk takes time, but is a necessary sacrifice. Of course, both of these are used for their plotting capabilities.

Now, examining the actual class:

```
class Erdos_Renyi_GNP:
```

For our purposes, the graphs that we are using can be represented by an Erdos-Renyi graph, through some type of manipulation. As such, this class will act as the super class to any extended classes. So, it takes in no other classes since it is the parent.

init

Erdos-Renyi

Inside of this, once all of the initializations have been executed, there exists this portion

```
if A is None:
            self.A = triu(array(random.rand(N, N) < p, dtype = int))
            if self_edges == False:

                self.A = self.A + self.A.T - 2*diag(diag(self.A))

            else:
                self.A = self.A + self.A.T - diag(diag(self.A))

else:  self.A = A
```

which sets the adjacency matrix. Since the class allows for an adjacency matrix to be passed beforehand, namely useful if copying the methods, we check to see if this flag has been changed from its default `None` to some value. If it is, we assign the set value to the adjacency matrix. If not, we form our own.

Effectively, `random.rand(N, N)` creates a matrix of size $N \times N$, with values $A_{ij} \in [0, 1]$. If $A_{ij} < p$, where p is the edge probability, then that entry is replaced with a 1. Otherwise, it's set to 0. This is the result of `array(random.rand(N, N) < p, dtype = int)`. We take the `triu`, or upper triangular, of this result so

that we can recombine it on the basis of self edges. Of course, when looking at something like the number of edges this introduces a form of error, but by doing this it forces symmetry.

Now, `self_edges` is a flag determining whether or not a node can be connected to itself. In our case, it never will. So, we add `self.A` with its transpose `self.A.T` and substract `2*diag(diag(self.A))`. Note that `diag(self.A)` grabs the diagonal elements of `self.A` and `diag(diag(self.A))` places those in the diagonal of a matrix of zeros with size $N \times N$. Since `triu` include the 1's along the diagonal, they're added twice and as such we subtract twice the diagonal.

Otherwise, we only subtract once due to the symmetry allowing for the diagonal to be a multiple of two, at least for an undirected and unweighted graph. To get the actual diagonal matrix, we execute

```
self.D = diag(sum(self.A, axis = 1))
self.L = self.D - self.A
```

Here, `sum(self.A, axis = 1))` sums the value of `self.A` across the second axis (since everything starts at 0). This could be done on the first axis as well, but for convention we use the second. This produces a vector of size N, which represents the diagonal values of `self.A`. We then use `diag` to force this vector into the diagonal of a zero matrix, like previously. Subtracting `self.D` and `self.A` is just the Laplacian `self.L`, as defined in the Documentation.

Last of the `init` function is declaring the edges and their locations:

```
self.M = sum(self.A)/2

self.edges = argwhere(triu(self.A) != 0)
self.potential_edges = argwhere((triu(1 - self.A) - diag(ones(self.N)) != 0))

eigenvalues, eigenvectors = eigh(self.L)

self.eigenvalues = eigenvalues
self.eigenvectors = eigenvectors
```

Note that `self.M` is just its definition as defined in the Documentation. The other variables, however, are a bit more complex. Note that `self.edges` represents all of the places where there's an edge. Namely, `triu(self.A) != 0` gives a Boolean array of all the locations in the upper triangular where the values are not zero. This is because we define the lack of an edge as 0, whereas the existence of an edge can be any value except for 0. So, `argwhere` gives us the location of these.

As for `self.potential_edges`, this applies only for unweighted and undirected edges. Basically, if we invert `self.A`, that is to say `1 - self.A`, this will show us all the locations where edges can be placed. We subtract a diagonal of ones again, since `triu(1 - self.A)` will have ones in its diagonal. Then, we apply the same rhetoric as finding `self.edges`.

We also define the eigenvalues and eigenvectors so that we can use it for spectral analysis.

As such, all of the `init` variables have been explained for a random graph. We can repeat the process for an SBM:

SBM

```
class SBM(Erdos_Renyi_GNP):
```

Basically, the SBM class extends the original class; it acts as the parent to the SBM class. We can set a few new properties:

```
self.p_in = p_in
self.p_out = p_out
self.N = N
self.k = k
```

```
self.n = N//k
```

where `self.k` is the number of communities, `self.n` is the nodes per community, and `self.p_in` and `self.p_out` are the inside and outside probabilities, respectively. Since this is a community-based structure, we have to define a one hot such that

```
if one_hot is None:
    self.one_hot = zeros((self.N, k))

    for dim in range(k):
        self.one_hot[(self.n*dim):(self.n*(dim + 1)), dim] = 1

else: self.one_hot = one_hot
```

Basically, the one hot is a 1 is the node belongs to the community and a 0 if it doesn't. This just creates an $N \times k$ array of ones and zeros that we can use to identify community members. In a random graph, it would all be ones. We define the mask for the inside community and outside communities such that

```
self.in_mask = self.one_hot @ self.one_hot.T
self.out_mask = 1 - self.in_mask
```

Basically, it's an $N \times N$ array where the edges are 1 if it's inside the community and 0 if outside the community. This is for the `self.in_mask`. The `self.out_mask` is really just the additive inverse of this. Now, to actually generate `A`:

```
if A is None:
    self.A_in = Erdos_Renyi_GNP(self.N, self.p_in).A * self.in_mask
    self.A_out = Erdos_Renyi_GNP(self.N, self.p_out).A * self.out_mask

    self.inside_edges = argwhere(triu(self.A_in) != 0)
    self.outside_edges = argwhere(triu(self.A_out) != 0)

    self.potential_inside_edges = argwhere((triu(1 - self.A_in) * self.in_mask - diag(ones(self.N)) != 0
    self.potential_outside_edges = argwhere((triu(1 - self.A_out) * self.out_mask != 0))

    self.A = self.A_in + self.A_out

else: self.A = A
```

Here, we effectively create two separate adjacency matrices `self.A_in` and `self.A_out`. The first is the in-communities and the second is the out-connections. We multiply them by their masks so that we can linearly add them and the edges don't overlap. Then, we derive the potential edges as normally, but include the different edge types for rewiring purposes. Mostly, if someone wishes to add community-directed rewiring.

Once this is done, we declare the rest of the structure-independent properties by calling the init of the Erdos-Renyi class:

```
Erdos_Renyi_GNP.__init__(self, N, 0, A = self.A)
```

We set `N` and `0` because we don't actually care what these properties produce.

plot_graph

The purpose of this function is to make a spy plot of the adjacency matrix. Since all of these are weighted the same, 1 represents a black dot and 0 represents a white dot, on the spy plot. We first create a figure object

```
fig, ax = plt.subplots(1, 1, figsize = figsize)
```

which allows us to modify the figure and axes separately. The `1, 1` means that there is only 1 subplot. Of course, `figsize` is just the figure size. Then we can plot the actual spy via

```
ax.spy(self.A)

ax.set_ylabel('node ID, $y$', fontsize = 12)
ax.set_xlabel('node ID, $x$', fontsize = 12)
ax.xaxis.set_label_coords(0.5, 1.175)

fig.tight_layout()
```

The `spy` function generates the image of the graph, the `set_ylabel` adds text to the y-axis, and `set_xlabel` to that of the x-axis. The first parameter there is the text to be added and `fontsize` is its font size. We have to adjust the x-axis label coordinates slightly by `ax.axis.set_label_coords` so that they appear above the x-axis numbering, as opposed to below. This isn't specified for any graph, just for a `figsize = (4, 4)`, unfortunately. This will be adjusted later.

Note that `fig.tight_layout()` optimizes the shape of the image, especially for subplots with more than 1. We return `fig` and `ax` so that they can be modified if needed.

plot_networkx

Now, we can create a `networkx` plot, which is really just a fancier `matplotlib` plot. We begin the same

```
fig, ax = plt.subplots(1, 1, figsize = figsize)

G = nx.Graph()
```

However, `nx.Graph()` creates a graph object so that we can add our edges and nodes. This is done by

```
G.add_nodes_from(arange(self.N))
G.add_edges_from(self.edges)
```

such that we can graph it by

```
pos = nx.spring_layout(G)

nx.draw_networkx_nodes(G, pos = pos, node_color = node_color, node_size = node_size, alpha = node_alpha
nx.draw_networkx_edges(G, pos = pos, edge_color = edge_color, alpha = edge_alpha, ax = ax)
```

where `pos` is the position of the graph according to the `spring_layout` function. This basically optimizes the placement of the nodes and edges. We then draw the edges and nodes separately, passing in their specific parameters. This is elaborated upon more in the docstrings. However, we pass `ax = ax`, so it knows that the axis by which we're working under is the one we initially declared.

We finish off by setting

```
plt.axis('off')
```

which removes the lines that form the axes.

copy_graph()

For the two different graphs we've defined, there are two definitions for `copy_graph()`. The first:

```
Erdos_Renyi_GNP(self.N, self.p, A = self.A.copy())
```

which just returns a new instance of the current graph. For the SBM:

```
SBM(self.N, self.p_in, self.p_out, self.k, A = self.A.copy())
```

Nothing special. This just helps us in spectral analysis for comparing the base state to the original state.

edge_removal()

Now, we introduce notion to remove edges from the graph. Removal in particular is explained in the Documentation, but it's basically removing edges of the network without creating any new ones. We begin by taking in an index to remove from:

```
r_ij, r_ji = self.edges[idx]
A_ij, A_ji = self.A[r_ij, r_ji].copy(), self.A[r_ji, r_ij].copy()
```

where `r_ij` is the edge to remove on the upper triangular and `r_ji` for the lower triangular. We then insert this into the array to retrieve the value. Normally, we can just set this to be one, but we've generalized it for weighted edges. Of course, we then set the removed edge to be zero:

```
self.A[r_ij, r_ji] = 0
self.A[r_ji, r_ij] = 0
```

Now, since this is an area where a potential edge exists, we must add this to that list and remove the edge from the actual edges:

```
self.potential_edges = append(self.potential_edges, self.edges[idx].reshape(1, 2), axis = 0)
self.edges = delete(self.edges, idx, axis = 0)
```

The `append` function creates a new array, so we just assign that to the original array we're appending on, since it takes in `self.potential_edges` as the array to be expanded on. We just have to `reshape` that particular edge because indexing it like that messes with the array structure.

Regardless, `delete` removes the value at that `idx` in `self.edges` and returns a separate array. Similar to `append`, we just place this into the original `self.edges`. We return `A_ij` and `A_ji` in case the graph is weighted.

edge_addition()

The process to add an edge is similar, but different:

```
potential_idx = random.randint(self.potential_edges.shape[0])

a_ij, a_ji = self.potential_edges[potential_idx]
```

We again get an index from the potential edges P on the range $[0, P-1]$ and then retrieve that location. Instead of saving that value, since we know that it will be zero always, we just reassign it such that

```
self.A[a_ij, a_ji] = A_ij
self.A[a_ji, a_ij] = A_ji
```

This way, if the edge is weighted, we preserve that status. Otherwise, the process of adding the new edge to the edges list and removing it from the potential edges is fairly similar:

```
self.edges = append(self.edges, self.potential_edges[potential_idx].reshape(1, 2), axis = 0)
self.potential_edges = delete(self.potential_edges, potential_idx, axis = 0)
```

rewire_graph()

To rewire the graph, we combine addition and removal such that we begin by picking an edge to remove at random:

```
idx = random.randint(self.edges.shape[0])
```

The `random.randint` picks an integer at random on the domain $[0, M-1]$. We can then grab the value of that edge before removing it such that

```
A_ij, A_ji = self.edge_removal(self.edges[idx], idx)

potential_idx = random.randint(self.potential_edges.shape[0])
```

```
self.edge_addition(self.potential_edges[potential_idx], potential_idx, (A_ij, A_ji))
```

This also executes the addition.

Since the rewiring has finished, we have to update the Laplacian

```
self.update_laplacian()
```

Note that this function returns nothing, since it really only modifies the state of the graph. If desired, the modified indices could be removed!

update_laplacian()

Updating the Laplacian properties is rather simple. We're really just updating the eigenvalues and eigenvectors, as well as the edges while we're there:

```
self.D = diag(sum(self.A, axis = 1))
self.L = self.D - self.A

eigenvalues, eigenvectors = eigh(self.L)

self.eigenvalues = eigenvalues
self.eigenvectors = eigenvectors

self.M = self.edges.shape[0]
```

Nothing special here; this is really similar to the init.

get_edge_colors()

One special feature of the SBM is that we can distinguish between community and connecting edges. That's what this function does. We:

```
colors = array([])

for edge in self.edges:
    i, j = edge
    if self.in_mask[i, j] == 1:
        colors = append(colors, in_color)
    else: colors = append(colors, out_color)

locs = colors == in_color
```

This generates a color array such that we can distinguish the edges. It also returns the location of these edges in a Boolean array so that we can use this with edge ranking. Effectively, if the `self.in_mask` is 1 at a location, then it must be a community edge. Otherwise, it's a connecting edge.

generate_network.py unit testing

To start, the imports:

```
import sys
sys.path.insert(1, '../')

from generate_network import *
```

Of course, since we're testing network generation we need that file, but `sys` is included so that we can reach it. We begin by making sure that the network size is correct:

test_network_size()

```
N = 100
p = 0.25

G = Erdos_Renyi_GNP(N, p)

assert(G.A.shape == (N, N))
```

A very simple, almost too simple test, but it at least makes sure that nothing was messed up in the many moving parts of this process. We want to also make sure that our edges are within the anticipated range:

test_edge_numbers()

```
bar_m = lambda N, p: N*(N - 1)/2 * p
std_m = lambda N, p: 0.10 * bar_m(N, p)

assert((G.M < bar_m(N, p) + std_m(N, p)) and (G.M > bar_m(N, p) - std_m(N, p)))

for _ in range(100):
    G1 = Erdos_Renyi_GNP(N, p)
    assert((G1.M < bar_m(N, p) + std_m(N, p)) and (G1.M > bar_m(N, p) - std_m(N, p)))
```

Since there's not exactly a closed form for the standard deviation that I could find, I just set it to be 10% of the expected number of edges. Then, we check to see that this works for our initial graph.

Just to show that we weren't lucky, I also show this for 100 more randomly generated graphs.

We can also make sure that there is no overlap in the potential edges and actual edges:

test_total_edges()

```
assert(G.edges.shape[0] == G.M)
assert(G.edges.shape[0] + G.potential_edges.shape[0] == G.N*(G.N - 1)/2)
```

We first make sure that there are no extra edges picked up anywhere, then test to make sure that the sum of the actual edges and potential edges is the total possible edges. That way, we know that there's no overlap at the surface.

test_rewiring()

Perhaps the most important test is to make sure that rewiring works - that is, it doesn't add one edge too many or remove one edge too many. This way, we know that each addition and removal is successful. We can do this by the following:

```
M = G.M
for _ in range(100):
    G.rewire_graph()
    assert(G.M == M)
```

Basically, every iteration we make sure that they're the same. This means that the addition and removal have succeeded in some way *and* no self edges were added in the process.

Note that none of these tests reflect the SBM - this is OK, they would be the exact same, anyway. The real test of the SBM comes from testing entropy, since that's more of what it indicates.

calculate_entropy.py

As always, imports come at the beginning of the file:

```
from numpy import argsort, argwhere, exp, isnan, log, zeros
from numpy.linalg import eigh
from scipy.special import xlogy
```

The new special library here is the `scipy.special` import. We use this over the `numpy` log function because it doesn't throw an error message for $0 \log_2 0$, which is what we want. Regardless, we begin with the classic model.

m_entropy()

The code for this is as follows:

```
M = G.M
eigenvalues = G.eigenvalues

f = eigenvalues / (2*M)

H = -xlogy(f, f)/log(2)
H[isnan(H)] = 0
```

This is really just following the formula defined in Documentation.md. The main difference here is that we remove `NaN` values with the line `H[isnan(H)] = 0` - basically, wherever the arrays are undefined we just set the value to be 0. We then return this array so that we can choose to sum it ourselves, or look at the entropy for each eigenvalue. We do something similar for the other entropy model.

b_entropy()

```
eigenvalues = G.eigenvalues

f = exp(-beta*eigenvalues)
f /= f.sum()

H = -xlogy(f, f)/log(2)
H[isnan(H)] = 0
```

Same idea here - we follow the formula then get rid of the `NaN` values. This section really isn't complicated. Let's move onto something more complicated!

edge_rankings()

This is the bread and butter of the experiment. Basically, we remove an edge, calculate the entropy, then restore the edge. Or, in code:

```
Hs = zeros((G.edges.shape[0]))

for idx, edge in enumerate(G.edges):

    jdx = argwhere(edge == G.edges)

    A = G.edge_removal(edge, jdx)

    G.update_laplacian()
    Hs[idx] = b_entropy(G, beta = beta).sum()
    G.edge_addition(edge, jdx, A)

edge_sort = argsort(-Hs)

edge_ranks = argsort(edge_sort)
```

We start by initializing an entropy array for each edge. Then, we iterate over all of the edges and find out where that edge is. We use this to remove the edge, calculate the entropy, and then store that entropy. After that, we restore it. This is somewhat like rewiring, except the edge isn't moved.

Once all of the entropies have been collected, we sort them with `argsort` in descending order (i.e., the highest entropy appears at top). To get the actual numerical ranks, we sort the `edge_sort`. Both of these last variables are returned.

top_ranked()

Now, we want to find out the proportion of connecting edges that are top ranked, to support our theory in Documentation.md:

```
cutoff_idx = int(cutoff*sorts.shape[0])
if cutoff_idx == sorts.shape[0]:
    is_connecting = ~locs[sorts]
else:
    is_connecting = ~locs[sorts][:cutoff_idx]

is_connecting = is_connecting.sum()/is_connecting.shape[0]
```

Note that `cutoff_idx` is just how many first element we choose, since the ranks are already sorted such that the top are at the front and the bottom are at the back. Then, if the `cutoff_idx` is really just the entire array, then we take the entire array as our initial `is_connecting`. Otherwise, we take it up until this index.

Since `locs` is a Boolean array, taking the negated sum of it will produce the number of connecting edges in that range. Then, we just divide it by the total possible elements in that range, giving us the proportion of top ranked edges that are connecting in that percentile.

calculate_entropy.py unit testing

The unit testing for `calculate_entropy.py` is a bit more complicated than the other tests. Namely, since entropy is a bit more random in nature, not all things are certain. We can begin with import statements:

```
import sys
sys.path.insert(1, '../')

from generate_network import *
from calculate_entropy import *
from numpy import log2
```

Nothing new here; we have to use `numpy` for something later. Otherwise, both utility functions are needed, which is to be expected. We can start with something basic:

test_minimum_entropy()

All entropy must be greater than zero. We can test this:

```
N = 100
p = 0.25

G = Erdos_Renyi_GNP(N, p)

assert(m_entropy(G).sum() >= 0)
```

This is true for both entropy models. Although individual parts can be less than zero, the aggregate cannot. We can also test maximum entropy:

test_maximum_entropy()

```
assert(m_entropy(G).sum() <= log2(N))
```

Note that this comes from the idea that the most dense a matrix can be is at $N \cdot p$ for $p = 1$. This is the ceiling of our entropy. We can also express that entropy scales upward with N:

test_scaling_entropy()

```
Ns = range(10, 100, 5)
for idx, N in enumerate(Ns[1:]):
    G0 = Erdos_Renyi_GNP(Ns[idx], p)
    G1 = Erdos_Renyi_GNP(N, p)
    assert(m_entropy(G1).sum() >= m_entropy(G0).sum())
```

This is actually expressed in our other notebooks. However, the idea is that if p is fixed, then adding new nodes increases the density of the network, since now there are more edges. Since our entropy model measures density, the entropy will increase with N. Now, we can move onto the other entropy model!

test_community_entropy()

We start by showing that the entropy increases with the number of communities, as defined in Documentation.md:

```
N = 100
ks = range(1, 10, 2)
p_in = 1
p_out = 0

for idx, k in enumerate(ks[1:]):
    G0 = SBM(N, p_in, p_out, k = ks[idx])
    G1 = SBM(N, p_in, p_out, k = k)

    assert(b_entropy(G0).sum() <= b_entropy(G1).sum())
```

Basically, as k increases, so does entropy. We fix `p_in` and `p_out` so that the entropy is as rigid as possible. But basically, the entropy logarithmically increases, since `ks[idx]` $<$ `k`.

Lastly, we can check that the entropy increases with random rewiring from a strongly structured system:

test_rewiring_entropy()

```
N = 100
k = 2
p_in = 1
p_out = 0

G = SBM(N, p_in, p_out, k = k)
H0 = b_entropy(G).sum()
for t in range(1000):
    G.rewire_graph()
    if t % 25 == 0 and t != 0:
        H1 = b_entropy(G).sum()
        assert(H1 < H0)
        H0 = H1
```

The main thing here is that we check every 25 iterations because rewiring here is a random process. So, potentially, the next rewire after this could produce something counterintuitive to the if statement. As such,

we check after a large enough interval to guarantee it. We also have to make sure to not rewire too much so that it doesn't converge to a purely random graph.

Conclusion

The unit tests here are kind of sparse. Since this is a random process at the core, it's kind of hard to test this stuff without just looking at the average each time. I really just wanted to show that the basic logic was solid, so that we can have that assumption moving forward. If I had the time, I would make far more dedicated unit tests, but the scope of this project is something that needs to be reigned in at least for a final submission. It's definitely something that can be done in the future.