CPBS 7711 - Module 4 Day 3

1. Motivating Problem from Domain
   1. There are 12 known genes that are associated with Fanconi anemia (FA). This disease is caused by genomic instability and is known to give rise to developmental abnormalities in major organ systems, early onset bone marrow failure and a high predisposition to cancer. Creating a functional network of all Fanconi anemia genes may assist in identifying new mechanisms and pathways that are affected by this disease and give rise to new mechanisms that can be used to develop new therapeutics.
2. Computational Problem
   1. Given a network of gene-gene connections (STRING 1.txt) and a list of loci associated with FA (Input.gmt.txt). Create and visualize an optimized co-functional network (CFN) that represents linkages between FA genes, where the linkages represent a potential shared biological function.
3. Specific Approach
   1. To create and visualize an optimized FA CFN: create a randomized population of FA subnetworks; subject that population of subnetworks to a genetic algorithm, resulting in a final, optimized population of FA subnetworks (the goal of this genetic optimization algorithm is to create a population of subnetworks that have the highest possible density score, which would yield a highly connected CFN); simulate a null case hypothesis (create a null case population of subnetworks and subject the population to the genetic algorithm) that gives rise to a statistical significance metric; simulate a permutation test, comparing the density (see subnetwork weight definition below) from the final optimized population of FA subnetworks with ~1000 instances of the null case hypothesis population of subnetworks. The resulting p value will represent the fraction of the trials where the null case hypothesis population density is equal to or greater than the density of the final optimized FA subnetwork population density.
4. Specific Implementation
   1. To create a randomized population of FA subnetworks; pull one gene (at random) from each of the 12 loci found in the Input.gmt.txt file and add the gene to a subnetwork, repeat this process 5000 times to get a population of randomized FA subnetworks.
   2. To implement the genetic algorithm that optimizes the randomized population of FA subnetworks: implement a two step routine (mutation and mating) which will mutate each gene in each subnetwork at a 5% probability; calculate the average density of the mutated population; calculate a selection and probability score for each subnetwork; using the probability score, select two parent subnetworks from the mutated population; for each of the 12 genes in a given subnetwork, create a child subnetwork by randomly choosing (50% chance) each of the 12 genes from either of the parent subnetworks (repeat 5000 times); calculated the average density of the new mated population; calculate the percent difference between the new mated populations average density and the previous populations average density; repeat the routine above until the percent difference is less than .5%.
   3. Create 1000 random non-fa subnetworks to use as the null case.
      1. This has been where most of my time over the last two weeks have been spent, please reference the Null case section in Discussion.
   4. Obtain the average gene score for every gene from the final optimized FA population. Implementation found in Module 3 Day 3.
   5. Calculate p-value.

**Pseudocode:**

Due to the amount of code in this project I will supply pseudo code for the main additions to the project. Note: I had to refactor most of the previous module’s code to implement the genetic algorithm.

**High level overview of the genetic algorithm:**

**Class Genetic Algorithm:**

1. Create 5000 random FA subnetworks 🡪*initialPopulation*
2. Subject the *initialPopulation* to the Genetic Algorithm:
3. While *densityImprovement* > .5: \*density improvement defined below
   1. Mutate:
      1. for each *subnetwork* in *initialPopulation*:
         1. create random number 1-100 🡪 *mutationProbability*
         2. empty list 🡪 *swappedSubnet*
         3. for each *gene* in *subnetwork*
            1. if *mutationProbability* is less than or equal to 5:

find *gene*’s locus 🡪 *locus*

*randomGeneIndexFromLocus* 🡪 random int (0 🡪 len(locus)-1)

*newGene* = *locus*[*randomGeneIndexFromLocus*]

*swappedSubnet*.append(*newGene*)

* + - * 1. else:

*swappedSubnet*.append(*gene*)

* + - 1. *generationXSubnets* 🡪
  1. Mate:
     1. empty list 🡪 *newGeneration*
     2. for each *subnetwork* in *generationXSubnets:*
        1. Calculate selection score:
           1. *subnetSelectionScore* = sum of weights in subnetwork
        2. Calculate sum of selection scores 🡪 *sumOfSelectionScores*
     3. for each subnetwork in *generationXSubnets*
        1. Calculate probability scores:
           1. *subnetProbabilityScore* = *subnetSelectionScore* / *sumOfSelectionScores*
     4. while iteration < 5000:
        1. *parentSubnets* = randomly choose two subnets using probabilityScores
        2. *child* = (
           1. while index < 12:

*whichParent* = randomInt (0 or 1)

if *whichParent* == 0:

*child*.append(*parent1[index]*)

elif *whichParent* == 1:

*child.*append(*parent2[index]*)

* + - 1. *newGeneration*.append(child)
    1. *averageDensity =* **calculate\_average\_density**(*newGeneration*)
    2. return *averageDensity, newGeneration*
  1. *densityPercentage*  = (current generations average density - previous generations average density) / previous generations average density \* 100

**function calculate\_average\_density**(subnets):

1. *weights* 🡪 empty list
2. utilize parallel processing to trigger **faUtilitiesInstance.count\_edges(**subnet, parentNetwork)
3. as the results from the executor come in 🡪 store in weights
4. return sum(*weights*)/len(*subnets*)

**Class: FA Utilities**

Note: The pseudocode below does not include all functions defined in this class. Other utility functions, pulled from the other modules, and utility functions created for this module are located in this class.

**function count\_edges(**subnet, parentNetwork): \*\*\*This is a huge bottleneck in the project, takes +/- 20 min to run on masterParentNetwork

1. \*Note: parentNetwork is a pandas dataframe, this argument is either the masterParentNetwork or parentFANetwork
2. create a mask = parentNetwork[“gene1”].isin(subnet) & parentNetwork[“gene2”].isin(subnet)
3. sort the mask by rows (use numpy)
4. drop any duplicates in the mask
5. get the weights column for the rows that 2. evaluated as true 🡪 *weights*
6. *weightSum* = sum(*weights*)
7. return *weightSum*

**function create\_master\_parent\_network(**faLoci):

1. read in the STRING 1.txt file and save as a pandas data frame, where the three columns from the STRING 1.txt file are (gene1, gene2, weight)
2. sort the data frame by rows and remove duplicate rows
3. extract all fa genes from Input.gmt.txt 🡪 *faGenes*
4. convert *faGenes* to a single column data frame 🡪 *faLociDF*
5. fill in the second column of *faLociDF* with “FAGENEROW” to indicate the gene does not exist in the STRING 1.txt file and fill in the weight column with 0
6. concatenate the two data frames 🡪masterParentNetwork
7. return *masterParentNetwork*

**function create\_parent\_network**():

Same steps as **create\_master\_parent\_network**, but read in faNetwork.txt (a filtered network file from STRING 1.txt, that only contains FA-FA genes) and do not concatenate with another data frame.

**Class: Create Random NonFA Subnetworks:**

Note: I include this class in the pseudocode because there was significant refactoring to this class.

**function create\_non\_fa\_subnetworks():**

1. **extract\_non\_fa\_genes()**
2. *geneDensity* 🡪 **calculate\_weights\_for\_bins(***masterParentNetworkDF***):**
3. *bins 🡪* **create\_bins(***geneDensity):*
4. *invertedBins* 🡪 **genes\_to\_bins(***bins***)**
5. use Thread Pool to **create\_individual\_non\_fa\_subnet(***finalPopulationSubnet***)**for every subnet in *finalPopulationSubnets*
6. *nfaSubnetworks* 🡪 {subnetworks returned from **create\_individual\_non\_fa\_subnet()**

**function create\_individual\_non\_fa\_subnet(**subnet**):**

1. for each gene in subnet:
   1. *genesBinKey*  = invertedBins[gene]
   2. *genesBin*  = bins[genesBinName]
   3. *randomIndex* = randomInt (0 🡪 len(*genesBin*) -1)
   4. *randomGene* = genesBin[randomIndex]
   5. check if *randomGene* is from Input.gmt.txt, if *randomGene*’s bin contains a non fa gene and if *randomGene* is a non fa gene.

**function genes\_to\_bins**(bins):

1. empty dictionary 🡪 *invertedBins*
2. for *binName, genes* in bins.items():
   1. for *gene* in *genes:*
      1. invertedBins[gene] = binName
3. return *invertedBins*

**function calculate\_weights\_for\_bins(**parentNetworkDF**):**

1. convert weight column from parentNetworkDF to a float for calculation
2. calculate average edge weights from parentNetworkDF using pandas functions 🡪 *geneDensity*
3. return *geneDensity*

**function create\_bins(**geneCounts):

1. *minGeneEdgeCount*  🡪 min(geneCounts)
2. *maxGeneEdgeCount* 🡪 max(geneCounts)
3. *edgeCountRange* 🡪 *maxGeneEdgeCount – minGeneEdgeCount*
4. *binSize* 🡪 *edgeCountRange* / 128 (\*as specified in paper)
5. for gene, weight in geneCounts:
   1. *binIndex* 🡪 (weight – minGeneEdgeCount)/ binSize
   2. bins[binIndex].append(gene)
6. return bins

**function extract\_non\_fa\_genes():**

1. *faGenes* 🡪 get faGenes from faLoci
2. empty set 🡪 *nfaGenes*
3. open globally defined *parentNetworkFile*:
   1. for each line in the *parentNetworkFile*:
      1. if either gene from the line is not in *faGenes*, add to *nfaGenes*
4. return *nfaGenes*

**function calculate\_average\_density(**subnets):

1. empty list. 🡪 *weights*
2. open ThreadPool to execute a **count\_edges()** instance for each of the subnetworks in the argument
3. return sum(weights) / len(subnets)

**High level pseudocode for statistical test (using strategy 3, as defined in the Null case section in Discussion):**

1. create ~1000 non fa subnetwork populations
2. calculate average density of the populations 🡪 store as null case test statistics
3. create initial FA subnetwork population
4. subject the FA subnetwork population to the genetic algorithm
5. calculate the average densities for each of the 5000 subnetworks in the final, optimized FA population 🡪 store as test statistics
6. for each of the *5000 fa test* statistics:
   1. calculate the number of times the 1000 null test statistics are greater than or equal to the *fa test statistic* 🡪 p-value

**Results:**

Unfortunately, due to my lack of ability to implement a more efficient null case I was only able to generate 8 null case, un-optimized, non fa subnetwork populations; therefore, my results are minimal. If I had more time/ understood and committed to one approach for the null case, I would expect to get 1000 null case populations and maybe even been able to finish more than one of the null case scenarios I outline below. The estimated runtime of the whole project before the last-minute changes made to the null case would have been ~42 days 24 hrs/day, which is very unreasonable. I would like to revisit this project after submission to see if I can improve the efficiency of the runtime and cleanliness/reproducibility of the project.

The current runtimes for each of the main components of the project are:

* Initial 5000 FA population generation 🡪 Genetic Algorithm (resulting in the final optimized FA population: ~15 minutes
* Score genes from final population: 25.5 minutes
* Create a single non fa subnetwork population without subjecting it to the modified Genetic Algorithm: 25.6 minutes.
  + To create 1000 populations: ~17.8 days (with the last-minute modifications)
  + Runtime of the generation of 8 populations: ~4 hours (starting evening of 12/4/23)
* Statistical test on 8 individual non fa populations with one optimized, final FA population: ~4.5 hours

Permutation Test: In result to only have 8 non fa populations to run a permutation test with, I am getting a p-value of 0 which is to be expected for the current situation. Therefore, I am changing the format of the output files to be the top ten densest subnetworks, just to have some significant results.

**Discussion:**

* **Subnetwork density**: To include a weighted edge calculation into the project, instead of the unweighted edge calculation (the total number of connected genes (edges) in the subnetwork), I utilized the third column from the parent network (derived from the STRING 1.txt file). The third column (**weight**) from the parent network signifies the **confidence** that STRING evaluates an edge to exist1. Including this ‘confidence’ score into the subnetwork density calculations adds a small amount of complexity into the significance of each subnetwork. The **density** calculation for each subnetwork is the sum of all weights in the subnetwork; for each of the genes in the subnetwork, if an edge exists (identified by the rows in which two genes from the subnetwork exist in one row of the parent network) the weight column from that row will be included in the summation of the weight density.
* **Average weight density:** To signify the total density of any given population of subnetworks, an average of the weight densities for each subnetwork was taken.
* **Null case:** I have gone down three different paths for the null case and I am not sure which one is correct. In retrospect I should’ve just chosen 1 approach and stuck with it. 1) Create 1000 versions of: a population of 5000 non fa subnetworks (this is done by first creating 128 bins that contain all genes (both from the STRING 1.txt file and the Input.gmt.txt file); using the initial fa population, iterate over each subnetwork and find the corresponding bin for each of the 12 genes in the subnetwork; replace each gene with a random non fa gene from the gene specific bin). Then subject each population to a modified version of the genetic algorithm (using bins in place of loci for mutation step). Calculate the average density for each population and use as test statistics in statistical test. 2) Create 1000 versions of: a population of fa subnetworks, similar logic to 1) but strictly using fa genes in the replacement process, instead of non fa genes. Subject each population to the genetic algorithm and calculate the density of the populations to use as test statistics. 3) Create 1000 populations of 5000 non fa subnetworks, same a 1) but do not subject to the genetic algorithm, calculate the average density of each population and use as test statistic in the statistical test.
* For the past week I have been working under the impression that we are supposed to subject the null case populations (population of non fa subnetworks) to the genetic algorithm. This would result in a set of 1000 average population densities that could be used as test statistics in the statistical test. Using these null case average population densities as test statistics, I believe, this would show us that the observed test statistic from the optimized FA population is equally as dense or denser in comparison to the optimized null case average densities. This situation would give us information about the probability of getting a dense population, it would not necessarily tell us any useful information about the effectiveness of the genetic algorithm. After discussing this with the TA’s (getting a final answer today 12/04/23), I changed my approach. This is personally advantageous because the expected runtime of the whole project (including the null case genetic algorithm) would take approximately 42 days to complete on my local machine. The new approach I am taking is to not subject the null case average population densities (1000) to the genetic algorithm and instead compare the average density of the un-optimized random non fa populations to the optimized FA subnetwork average densities. The results from this statistical test will tell us about the proficiency of the genetic algorithm, which from my understanding, is one of the main components of the assignment.

If I had another two weeks to work on this project, I would have done a fair amount of refactoring, including:

* Refactor the count\_edges() function and locations where this function is called.
* Refactor the use of pandas data frames, this is one of the largest bottlenecks in my project at the moment. Whenever I have to use the master parent network data frame for data manipulation or to check for genes in the data frame (count\_edges()), it takes ~20 minutes to complete.
* Bring calculate\_average\_density into FA Utilities.
* Bring invertedBins function into FA Utilities.
* Clean up all variable and function names to correspond to this modules terminology (use of count\_edges, instead of calculate\_weight).
* Clean up any unused variables and functions carried over from previous modules.

Additionally, I thought of refactoring the whole project to use a mysql database to store the STRING 1.txt file and Input.gmt.txt file. I should have thought of this much sooner as I have a fair amount of database administration/management experience and the likelihood that it would’ve cut down the runtime would be significant.

**Citations:**

1. https://string-db.org/cgi/info?sessionId=brF0q0dx3EFx&footer\_active\_subpage=scores