AA228 Project 1

Jenna Lee

Oct 18th, 2019

1) Approach.

Three sets of data were given to find the structure of data that generate highest Bayesian score. I implemented local search with randomized start.

1 Calculating Bayesian Score.

Bayesian score is calculated using

$$\ln P(G|D) = \ln P(G) + \sum_{i=1}^{n} \sum_{i=j}^{q_i} \ln \big(\frac{\Gamma(\alpha_{ij0})}{\Gamma(\alpha_{ij0} + m_{ij0})}\big) + \sum_{k=1}^{r_i} \ln \big(\frac{\Gamma(\alpha_{ijk} + m_{ijk})}{\Gamma(\alpha_{ijk})}.$$

Since our prior of each graph is uniform, we don't bother to calculate $\ln G$. Likewise, given the uniform distribution per each graph, we have $\alpha_i jk = 1 \forall i, j, k$. Therefore, this problem boils down to coutning m_{ijk} given the data structure.
A list of dictionary was implemented to easily calculate m_{ijk}. By maintaining one dictionary per variable, we quickly retrieve and update m_{ijk} values. Likewise, a dictionary of j to optimize runtime. Below is the pseudocode:

```
for each data (d1, d2, ... dn):
    for each variable d_i:
        #i indicates ith variable

        #find j
        find parents of d_i
        if there's no parents, j = 1
        else: assign an index corresponding to the current parents \\ configuration.

        cache j: lookup_i[parents configuration] = j

        #find k
        k = value of d_i, which conveniently can be used as an index.
```

```
        defaultdict M[i,j,k] +=1

#calculate score
#-- Just occuring to me at the last minute, but this can be improved so\\ much more
for all variables
    for all j values appeared in the dataset
        for all k values
            calculate Bayesian score
```

2 Graph search.

Local search was implemented with randomized initial graphs.
`nx.gn_graph(n)` was used to enable this. Based on the initial graph, stochastic local search was done. Figure 2c shows the general local search algorithm. What I did differently to expedite my search is, however, I update my graph as soon as I find the first neighbor with a better Bayesian score. While this stochastic approach does not guarantee the global optimum, it results in 'good enough' value with more civilized run-time. Below is the pseudocode:

```
repeat:
    repeat:
        generate a random neighbor
        if we can't generate any feasible neighbor:
            return G_best, current_score
        find bayesian score of the random neighbor

        until the new Bayesian score is better than the current score
    return highest bayesian score, and associated graph

    if the newly updated score is only marginally (epsilon=0.001) larger \\than our
        break

    if we have continued our search max_iters number of timse:
        break
```

And to randomly generate a neighbor:

```
repeat:
    randomly generate a number, r
    if r ==1:
        reverse an edge picked from a shuffled list of edges
```

```
        if the new graph is not cyclic:
            return the new graph
    if r ==2:
        remove an edge picked from a shuffled list of edges
        if the new graph is not cyclic:
            return the new graph
    if r==3:
        add an edge between two randomly picked nodes from an exhaustive list
        if the new graph is not cyclic:
            return the new graph
    until there isn't any more possible operations left
```

To make sure that the resulting graphs are acyclic, the following function was used:

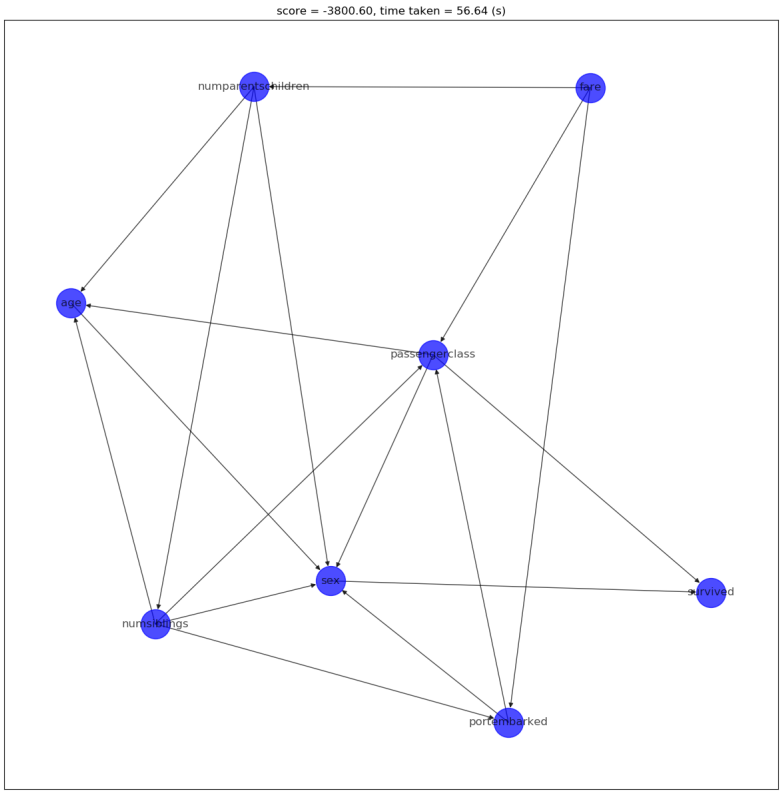
```
    def is_cyclic(self,G):
try:
    nx.find_cycle(G, orientation='original')
except:
    return False
return True
```

1: **function** LocalDirectedGraphSearch(f, G_0)
2: $G' \leftarrow G_0$
3: **repeat**
4: $G \leftarrow G'$
5: $\mathcal{G} \leftarrow$ neighbors of G
6: $G' \leftarrow \arg\max_{G' \in \mathcal{G}} f(G')$
7: **until** $f(G') \leq f(G)$
8: **return** G

Figure 1: General local search algorithm

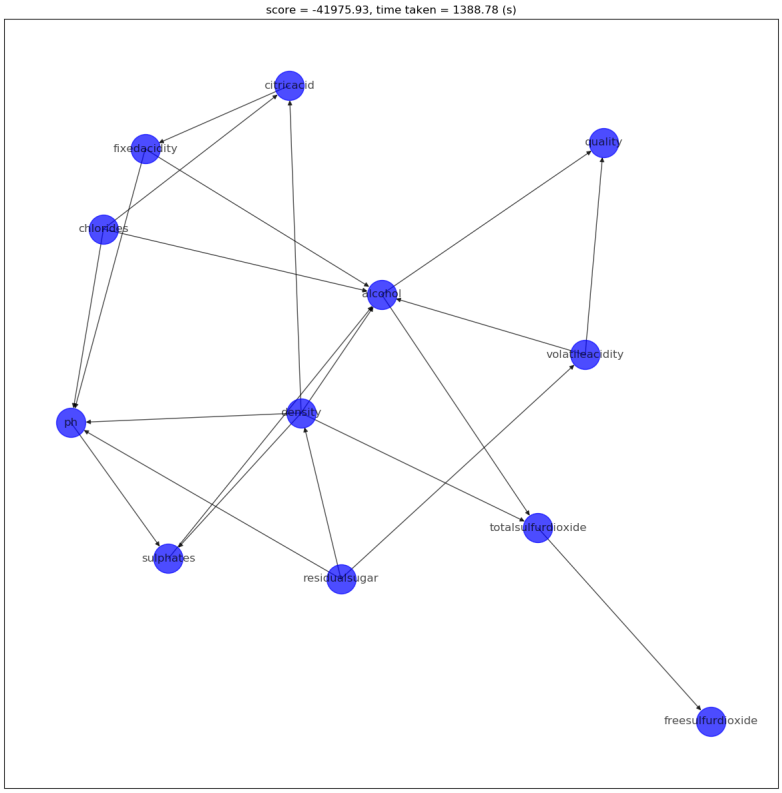
2) Results

(a) Small graph



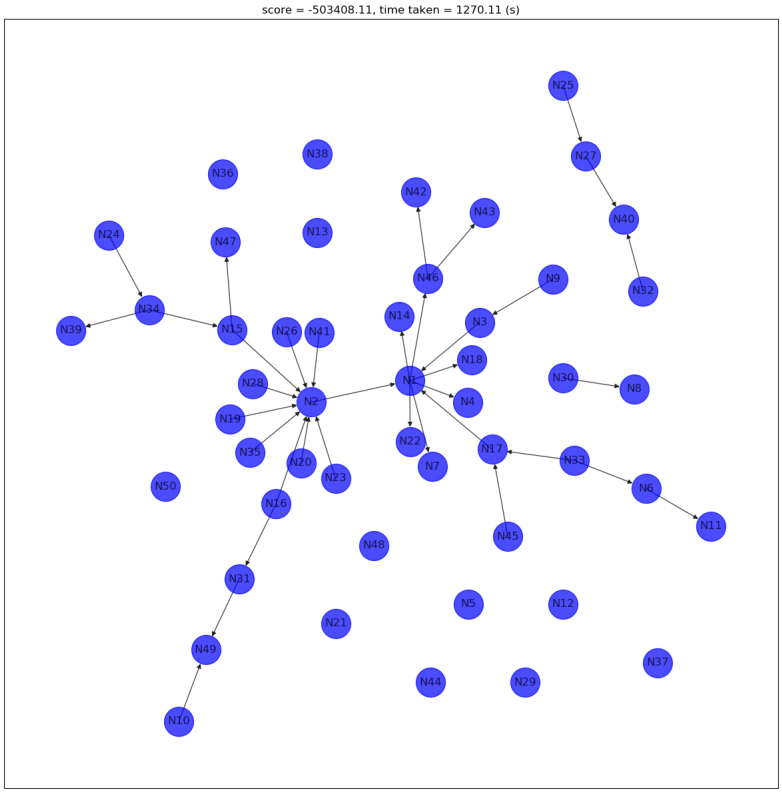
Execution time was at around 5 seconds.

(b) Medium graph



Execution time was at around 10 minutes.

(c) Large graph



Execution time was at around several hours (overnight).