Russell Martin

AA228 Winter 2023

Project 1: Bayesian Structure Learning

Problem Statement

This project implements an algorithm to find a (somewhat) bayesian-optimal directed acyclic graph given some data.

Algorithm Description

I chose to use *opportunistic local search* for my graph search algorithm. This algorithm works by randomly choosing an operation - add, remove, or flip and edge - and then applying it to the current graph to generate a new graph. This new graph is kept and iterated upon if its Bayesian score is better than the original graph's score. The algorithm concludes when the overall best score has not been achieved in `maxAttempts` consecutive graphs.

This method of hill climbing is susceptible to getting stuck in local minima - to prevent this, there are a few methods used to introduce random resets into the algorithm. First, after a new graph is generated, even if its Bayesian score is worse than the original graph, I assign a 5% chance to keeping the new (worse) graph. Second, if the algorithm seems to get stuck and fails to find any improvement after `maxAttempts/20` graphs, I revert the most previous change made to the graph. Last, if there has not been an improvement on the overall best score `maxAttempts/10` new graphs, I reset the graph back to its state where the best overall score was achieved. For optimizing the graphs, I set `maxAttempts` with a goal of finishing optimization in approximately 10 to 20 minutes.

Pseudocode for the algorithm is shown in **Appendix 1**.

The runtime of these algorithms is in **Figure 1**. The runtime of these algorithms is summarized in **Table 1**.

Table 1. Summary of algorithm performance with various data dimensionality. Note that the 20 minute time limit was reached for the 50-variable dataset.

Number of variables	maxAttempts	Optimized bayesian score	Runtime (mins)	Graphs generated and scored per minute
8	1000	-3796.86	11.63	223
12	300	-42051.16	19.36	33
50	300	-483860.92	20	10

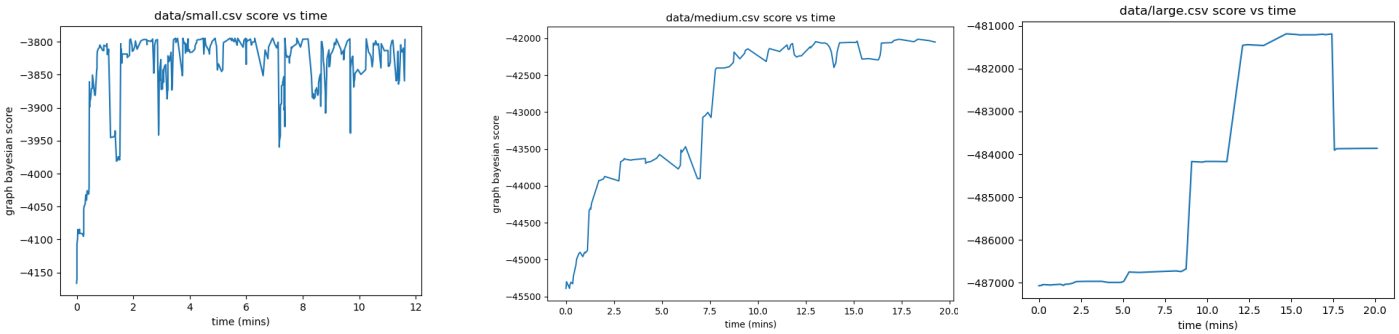

Figure 1. Plot of score versus runtime for the small (left), medium, and large (right) datasets. The algorithm struggled with the large dataset, as seen by the long duration in between data points in the rightmost plot.

Graph Plots

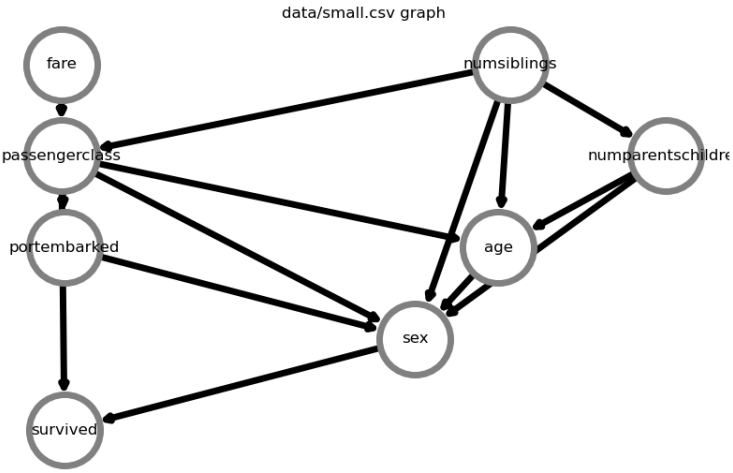

Figure 2. Graph from “small” dataset.

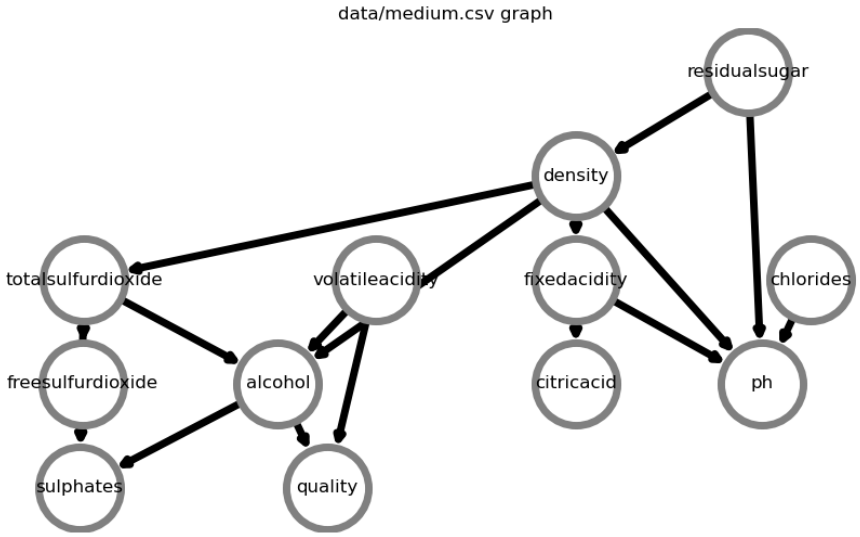

Figure 3. Graph from “medium” dataset. Note that “totalsulfurdioxide” and “freesulfurdioxide” are both parents of “sulphates” (this seems to have gotten overlapped in the plot).

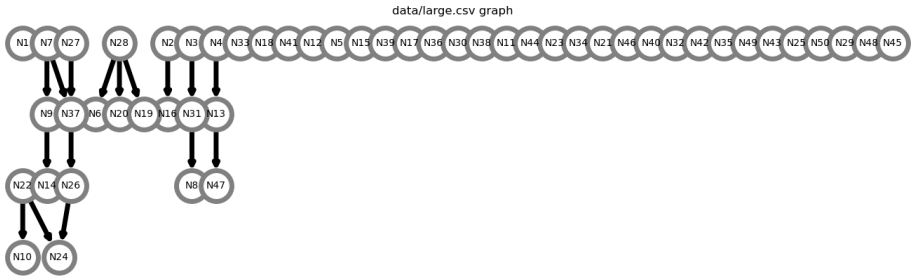

Figure 4. Graph from the “large” dataset. It seems like, even with 20 minutes of optimization, more time would have made this graph more representative of the data.

Appendix 1: Algorithm Pseudocode

```
Generate graph G
Set allTimeBest_graph to G
Set allTimeBest_score and score to bayesScore(G)

While True:
   Randomly add an edge, remove an edge, or flip an edge to generate G_new
   Determine score_new of G_new
   If score_new > allTimeBest_score:
      Update allTimeBest_score to score_new
      Update allTimeBest_graph to G_new
      Set countWithoutAllTimeImprovement to 0
   Else:
      Increment countWithoutAllTimeImprovement
   If score_new > score or randInt(0,20) == 0:
      Update score to score_new
      Update G to G_new
      Set countWithoutImprovement to 0
   Else:
      Increment countWithoutImprovement

   If countWithoutImprovement > maxAttempts/10:
      Undo last change to G
   If countWithoutAllTimeImprovement > maxAttempts/5:
      Reset G to allTimeBest_graph
   If countWithoutAllTimeImprovement > maxAttempts:
      Break

Save allTimeBest_G and allTimeBest_score
Plot
```

```python
import sys
import numpy as np
import networkx as nx
import pandas
import matplotlib.pyplot as plt
import scipy.special
import copy
from datetime import datetime
from networkx.drawing.nx_pydot import graphviz_layout

def write_gph(dag, idx2names, filename):
    with open(filename, 'w') as f:
        for edge in dag.edges():
            f.write("{}, {}\n".format(idx2names[edge[0]], idx2names[edge[1]]))

def lg(x):
    return scipy.special.loggamma(x)

def bayes_score_component(M, alpha):

    p = np.sum(lg(alpha + M))
    p = p - np.sum(lg(alpha))
    p = p + np.sum(lg(np.sum(alpha, axis=1)))
    p = p - np.sum(lg(np.sum(alpha, axis=1) + np.sum(M, axis=1)))

    return p

def getIdxFromParentState(r, idxOfParents, stateOfParents):
    # r = num instantiations of each variable
    returnIdx = 0

    nStatesForEachParent = [r[p] for p in idxOfParents]
    q = np.prod(nStatesForEachParent) # total number of parent states
    idxOfAllParentStates = range(q)
    idxOfAllParentStates = np.reshape(idxOfAllParentStates, (nStatesForEachParent))
    return idxOfAllParentStates[tuple(stateOfParents)]

def counts(vars, G, D, graphInfo):
    # D = data of shape(n,m) where n is num variables and m is num datapts
    # G = bayes struct
    # returns array M of shape(n,)

    # n = num variables
    n = np.shape(D)[0]
    m = np.shape(D)[1]

    # r = num instatiations of var[i]
    r = graphInfo['r']
    # q = num instantations of var[i]'s parents
    q = graphInfo['q']
    # m = list of size n, each element containing a shape(q[i], r[i]) containing counts
    of each state
    M = [np.zeros((q[i], r[i])) for i in range(n)]

    for datapt in range(m):
        for datapt_item in range(n):
            state_of_self = D[datapt_item, datapt]
            parents = list(G.predecessors(datapt_item))
            state_of_parents_idx = 0
            if len(parents) != 0:
                state_of_each_parent = [D[parent, datapt] for parent in parents]
                state_of_parents_idx = getIdxFromParentState(r, parents,
                state_of_each_parent)

            M[datapt_item][state_of_parents_idx, state_of_self] += 1
    return M

```

```
66
67  def prior(vars, graphInfo):
68      n = len(vars)
69      r = graphInfo['r']
70      q = graphInfo['q']
71
72      # m = list of size n, each element containing a shape(q[i], r[i]) containing counts
        of each state
73      M = [np.ones((q[i], r[i])) for i in range(n)]
74      return M
75
76
77
78  def bayes_score(vars, G, D, graphInfo):
79      n = np.shape(vars)[0]
80      M = counts(vars, G, D, graphInfo)
81      alpha = prior(vars, graphInfo)
82      return np.sum([bayes_score_component(M[i], alpha[i]) for i in range(n)])
83
84  def getGraphInfo(n, G, D):
85      # r = num instatiations of var[i]
86      r = [np.max(D[i])+1 for i in range(n)]
87      # q = num instantations of var[i]'s parents
88      q = [int(np.prod([r[j] for j in list(G.predecessors(i))])) for i in range(n)]
89
90      returnVal = {'r': r, 'q': q}
91      return returnVal
92
93  def compute(infile, outfile):
94      startt=datetime.now()
95
96      # (1) read in data
97      D = pandas.read_csv(infile)
98      vars = D.columns
99      D = np.array(D).T - 1 # subtract 1 for 0-indexing
100     n = len(vars)
101
102     # (2) get a graph, set nodes to 0, 1, ..., n
103     G = nx.DiGraph()
104     G.add_nodes_from(range(n))
105     graphInfo = getGraphInfo(n, G, D)
106
107     # (3) score the graph
108     score = bayes_score(vars, G, D, graphInfo)
109     allTimeBest = {'G': G, 'score': score}
110
111     # (4) optimize the graph using local directed graph search
112     maxTime = 20 # mins
113     maxAttempts = 1000
114     undoRate = np.floor(maxAttempts/10)
115     resetRate = np.floor(maxAttempts/5)
116     consecutiveNewGraphsWithoutImprovement = 0
117     consecutiveNewGraphsWithoutAllTimeImprovement = 0
118     totalGraphsGenerated = 0
119     scores = np.array([[0, score]])
120     mostRecentChange = {'type': None, 'nodes': None}
121     while True:
122
123         runtime_mins = np.round((datetime.now() - startt).total_seconds()/60, 2)
124         G_new = copy.deepcopy(G)
125         existingEdges = list(G_new.edges)
126         # (4.1) add an edge
127         decision = np.random.randint(0,3)
128         if decision == 0 or len(existingEdges) == 0:
129
130             for i in range(maxAttempts):
131                 # pick start and end node, make sure they're not the same and don't
```

```
                already exist
                for j in range(maxAttempts):
                    startNode = np.random.randint(0, n)
                    endNode = np.random.randint(0, n)
                    if startNode != endNode and (startNode, endNode) not in list(G.edges)
                     and (endNode, startNode) not in list(G.edges):
                        break

                # add to graph. If we still have a DAG, we can go get the score
                G_new.add_edge(startNode, endNode)

                if nx.is_directed_acyclic_graph(G_new):
                    break

        # (4.2) alternatively, remove or flip and existing edge
        else:

            for i in range(maxAttempts):
                existingEdges = list(G_new.edges)
                edgeToChange = existingEdges[np.random.randint(0, len(existingEdges))]
                startNode = edgeToChange[0]
                endNode = edgeToChange[1]
                if decision == 1: # remove
                    G_new.remove_edge(startNode, endNode)
                else: # flip
                    G_new.remove_edge(startNode, endNode)
                    G_new.add_edge(endNode, startNode)
                if nx.is_directed_acyclic_graph(G_new):
                    break

        # (4.3) get the score of G_new, and update G if the score is an improvement
        if nx.is_directed_acyclic_graph(G_new):
            graphInfo_new = getGraphInfo(n, G_new, D)
            score_new = bayes_score(vars, G_new, D, graphInfo_new)
            totalGraphsGenerated += 1

            if score_new > score or np.random.randint(0, 20) == 0:
                if score_new > score:
                    consecutiveNewGraphsWithoutImprovement = 0
                if score_new > allTimeBest['score']:
                    improvement = round(score_new-allTimeBest['score'],2)
                    print(f'New all time best score! {round(score_new,2)} (change: {
                    improvement})')
                    allTimeBest['score'] = score_new
                    allTimeBest['G'] = G_new
                    consecutiveNewGraphsWithoutAllTimeImprovement = 0

                G = copy.deepcopy(G_new)
                score = copy.deepcopy(score_new)
                scores = np.append(scores, np.array([[runtime_mins,score]]), axis=0)
                mostRecentChange['type'] = decision
                mostRecentChange['nodes'] = [startNode, endNode]
                print(f'Improved score {round(score,2)}. Runtime = {runtime_mins}')
            else:
                consecutiveNewGraphsWithoutImprovement += 1
                consecutiveNewGraphsWithoutAllTimeImprovement += 1
        else:
            consecutiveNewGraphsWithoutImprovement += 1
            consecutiveNewGraphsWithoutAllTimeImprovement += 1

        # (4.4) print status
        if np.mod(consecutiveNewGraphsWithoutAllTimeImprovement, 100) == 0 and
        consecutiveNewGraphsWithoutAllTimeImprovement > 99:
            print(f'consective graphs without all-time score improvement  = {
            consecutiveNewGraphsWithoutAllTimeImprovement}' +
            f' runtime = {runtime_mins}')

```

```
194             # (4.5) if no changes seem to help, undo the last change
195             if np.mod(consecutiveNewGraphsWithoutImprovement, undoRate) == undoRate-1:
196                 print(f'No changes are helping - removing last change')
197                 if mostRecentChange['type'] == 0: # undo an add
198                     try:
199                         G.remove_edge(mostRecentChange['nodes'][0], mostRecentChange['nodes'
                            ][1])
200                     except:
201                         print(f'error, cannot remove nodes from graph')
202                 elif mostRecentChange['type'] == 1: # undo a remove
203                     try:
204                         G.add_edge(mostRecentChange['nodes'][0], mostRecentChange['nodes'][1
                            ])
205                     except:
206                         print(f'error, cannot add nodes to graph')
207                 elif mostRecentChange['type'] == 2: # undo a flip
208                     try:
209                         G.remove_edge(mostRecentChange['nodes'][1], mostRecentChange['nodes'
                            ][0])
210                         G.add_edge(mostRecentChange['nodes'][0], mostRecentChange['nodes'][1
                            ])
211                     except:
212                         print(f'error, cannot flip nodes')
213 
214 
215             if np.mod(consecutiveNewGraphsWithoutAllTimeImprovement, resetRate) == resetRate-
                1:
216                 print(f'{resetRate} consecutive graphs w/o all-time improvement - resetting
                    back to all-time G')
217                 G = copy.deepcopy(allTimeBest['G'])
218 
219 
220             # (4.6) if we've had too many consecutive duds, we're done
221             if consecutiveNewGraphsWithoutImprovement > maxAttempts/10 or
                consecutiveNewGraphsWithoutAllTimeImprovement > maxAttempts or runtime_mins >
                maxTime:
222                 break
223 
224         # (5) return the graph when done
225         network_str = ''
226         G = copy.deepcopy(allTimeBest['G'])
227 
228         for i in range(len(vars)):
229             parentName = vars[i]
230             kidsIdxs = list(G.successors(i))
231             if len(kidsIdxs) != 0:
232                 for k in kidsIdxs:
233                     strToAdd = f'{parentName}, {vars[k]} \n'
234                     network_str += strToAdd
235             else:
236                 network_str += f'{parentName}, \n'
237         graphsPerMin = round(totalGraphsGenerated/runtime_mins, 2)
238         print(f'generated {totalGraphsGenerated} graphs in {runtime_mins} mins ({graphsPerMin
            } graphs/min), best score = {score}')
239         print(network_str)
240 
241         if 1:
242             plt.figure()
243             plt.plot(scores[:,0], scores[:,1])
244             plt.xlabel('time (mins)')
245             plt.ylabel('graph bayesian score')
246             plt.title(f'{infile} score vs time')
247 
248             nodeMapping = dict(zip(list(range(len(vars))), list(vars)))
249             G = nx.relabel_nodes(G, nodeMapping)
250             options = {
251                 "font_size": 10,
```

```
            "node_size": 1000,
            "node_color": "white",
            "edgecolors": "gray",
            "linewidths": 5,
            "width": 5,
        }
        plt.figure()
        # pos = nx.shell_layout(G)
        pos = graphviz_layout(G, prog='dot')
        nx.draw_networkx(G, pos=pos, arrows=True, arrowstyle='->', **options)
        plt.title(f'{infile} graph')
        plt.axis("off")
        plt.show()
    print('done')

def main():

    if len(sys.argv) != 3:
        raise Exception("usage: python project1.py <infile>.csv <outfile>.gph")

    inputfilename = sys.argv[1]
    outputfilename = sys.argv[2]
    compute(inputfilename, outputfilename)


if __name__ == '__main__':
    if 0:
        main()
    else:
        inputfilename = "data/large.csv"
        outputfilename = "graphs/large.gph"
        compute(inputfilename, outputfilename)

```