Project 1: Bayesian Structure Learning

Justin Wang JJWANG01@STANFORD.EDU
AA228/CS238, Stanford University

1. Algorithm Description

For my algorithm I simply conducted a greedy local search. I started off by creating a random DAG, and then stepping to a random graph neighbor. If the graph neighbor's Bayesian score was higher than the current DAG's score, then I saved the neighbor as the current graph along with its Bayesian score. For the graph initialization, and each subsequent graph, I checked if it was cyclic before continuing. I experimented with multiple k_{max}, I ended up sticking with 20 so that I would not have to wait so long. While writing the code, I realized that I didn't have to traverse the entire dataset for each variable each time I wanted to update the graph–I only needed to update the counts/parental instantiations of the child of whatever edge I added or removed.

Runtime:

- Small Graph: $<$ 1 minute
- Medium Graph: 2 minutes
- Large Graph: $<$ 10 minutes

2. Graphs

3. Code

```
"""
project1.py
"""

import sys
import pandas as pd
import numpy as np
from scipy.special import gamma, gammaln
import math
from collections import defaultdict
import itertools

import networkx as nx

counts = None
variables = None
variable_values = dict()
D = None
```

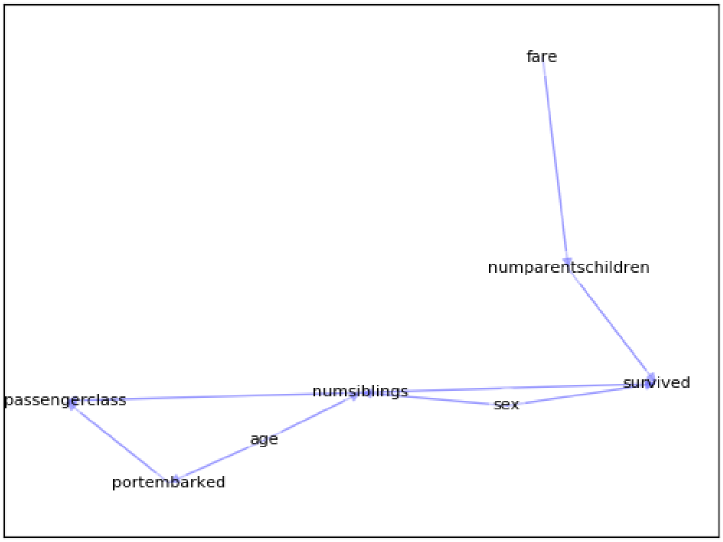

Figure 1: Small graph.

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


def idx2dto1d(row, var_name, parents):
    if len(parents) == 0:
        return 0

    j = []
    shape = []
    for var_name_ in variables:
        if var_name_ != var_name and var_name_ in parents:
            j.append(row[var_name_]-1)
            shape.append(variable_values[var_name_])
    return np.ravel_multi_index(j, tuple(shape))


def populate_counts(G):
    # because gamma(1)/gamma(1) = 1, and log 1 = 0,
    # we don’t need to care about the "missing" instantiations because they
    amount to 0 in the bayesian score
    global counts
```

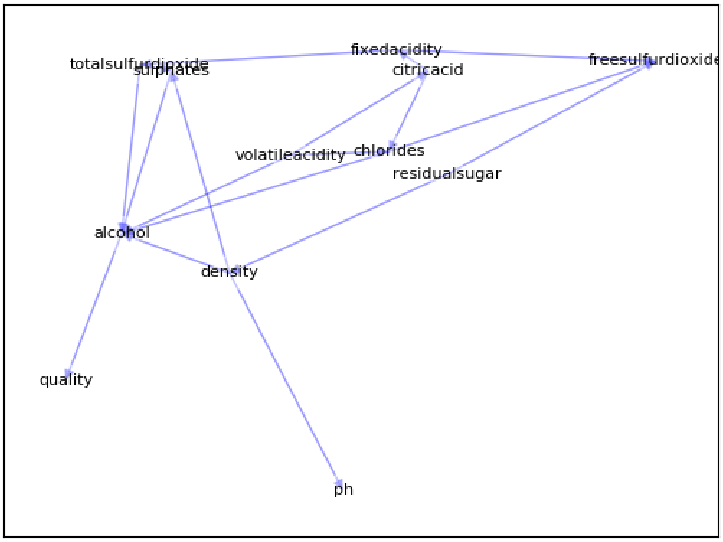

Figure 2: Medium graph.

```
    n = len(variables)
    counts = dict()
    for index, row in D.iterrows():
        for i in range(n):
            i = i
            var_name = variables[i]
            parents = [var_name_ for var_name_ in G.predecessors(var_name)]
            j = idx2dto1d(row, var_name, parents)
            k = row[var_name]
            if i not in counts:
                counts[i] = dict()
            if j not in counts[i]:
                counts[i][j] = defaultdict(int)
            counts[i][j][k] += 1
    #return counts


def update_counts(G, var_name):
    # do dynamic programming to propagate the change in parents?
    global counts

    n = len(variables)
    parents = G.predecessors(var_name)
```

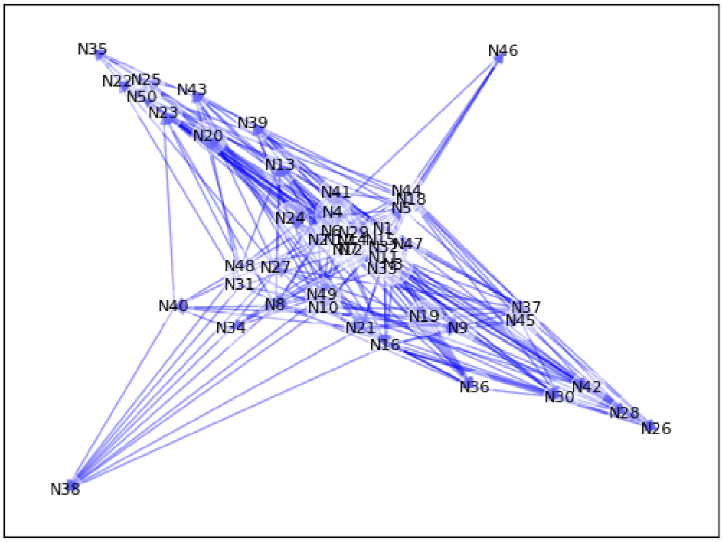

Figure 3: Large graph.

```
    i = variables.index(var_name)
    counts[i] = dict()
    for index, row in D.iterrows():
        parents = [var_name_ for var_name_ in G.predecessors(var_name)]
        j = idx2dto1d(row, var_name, parents)
        k = row[var_name]
        if j not in counts[i]:
            counts[i][j] = defaultdict(int)
        counts[i][j][k] += 1


def bayesian_score(G):
    # drop log P(G)
    n = len(variables)
    res = 0
    for i in range(n):
        component = 0
        for j in counts[i]:
            # all pseudo counts are 1
            alpha_sum = variable_values[variables[i]]
            m_sum = np.sum([counts[i][j][k] for k in counts[i][j]])
            component += gammaln(alpha_sum) - gammaln(alpha_sum + m_sum)
            for k in counts[i][j]:
                alpha_ijk = 1
                m_ijk = counts[i][j][k]
                component += gammaln(alpha_ijk + m_ijk) - gammaln(alpha_ijk)
        res += component
    return res


def random_directed_graph(p=0.2):
    # generate arbitrary ordering of nodes
    G = nx.DiGraph()
    G.add_nodes_from(variables)
    for i in range(len(variables)):
        for j in range(i+1, len(variables)):
            if np.random.uniform() < p:
                G.add_edge(variables[i], variables[j])
    return G

    """
    # limit number of parents for a given node
    G = nx.DiGraph()
    G.add_nodes_from(variables)
    edges = itertools.permutations(variables, 2)
    for e in edges:
        if np.random.uniform() < p:
            G.add_edge(*e)
    return G
    """


def rand_graph_neighbor(G):
    nodes = list(G.nodes)
    edges = list(G.edges)
    i = np.random.randint(len(nodes))
    j = (i + np.random.randint(1, len(nodes))) % len(nodes)
    G_ = G.copy()
    if (nodes[i], nodes[j]) in edges:
        G_.remove_edge(nodes[i], nodes[j])
        update_counts(G_, nodes[j])
    else:
        G_.add_edge(nodes[i], nodes[j])
        update_counts(G_, nodes[j])
    return G_


def is_cyclic(G):
    try:
        nx.find_cycle(G, orientation=’original’)
    except nx.exception.NetworkXNoCycle:
        return False

    return True
```

```
def hill_climbing(D, outfile, k_max=20):
    """
    Returns a graph instantiated by greedy local search algorithm.
    """
    G = random_directed_graph()
    while is_cyclic(G):
        G = random_directed_graph()
    print("writing graph")
    write_gph(G, outfile)
    populate_counts(G)
    y = bayesian_score(G)
    for k in range(k_max):
        G_ = rand_graph_neighbor(G)
        if is_cyclic(G_):
            y_ = float(’-inf’)
            print(’cyclic’)
        else:
            y_ = bayesian_score(G_)
        if y_ > y:
            y, G = y_, G_
            print("writing improved graph")
            write_gph(G, outfile)

    return G


def compute(infile, outfile):
    # WRITE YOUR CODE HERE
    # FEEL FREE TO CHANGE ANYTHING ANYWHERE IN THE CODE
    # THIS INCLUDES CHANGING THE FUNCTION NAMES, MAKING THE CODE MODULAR,
    BASICALLY ANYTHING
    global D, variables, variable_values
    D = pd.read_csv(infile)
    variables = list(D.columns)
    for var in variables:
        num_values = D[var].max()
        variable_values[var] = num_values

    # implement simple algorithm
    G = hill_climbing(D, outfile)

    # convert to file
    write_gph(G, outfile)


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__’:
    main()

# ---------------------------------------------------------------------------
"""
draw_graph.py
"""

import networkx as nx
import pandas as pd
import sys
import matplotlib.pyplot as plt

if len(sys.argv) != 4:
    raise Exception("usage: python draw_graph.py <data_file>.csv <edge_file>.
    gph <out_file>.png")

data_file = sys.argv[1]
edge_file = sys.argv[2]
out_file = sys.argv[3]

D = pd.read_csv(data_file)
variables = list(D.columns)
G = nx.DiGraph()
G.add_nodes_from(variables)

with open(edge_file, ’r’) as f:
    for line in f:
        e = line.strip().split(",")
        G.add_edge(*e)


pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos, node_color="w", alpha=0.4)
nx.draw_networkx_edges(G, pos, alpha=0.4, node_size=0, width=1, edge_color="b
    ")
nx.draw_networkx_labels(G, pos, font_size=8)
plt.savefig(out_file)
```