Project 1: Bayesian Structure Learning

Erich Trieschman etriesch@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

In this project I use an iterative combination of the K2 search algorithm and the local search algorithm to find Bayesian network structures that best fit three given datasets. My algorithm proceeds as follows.

I first implement the K2 search algorithm with a fixed topological sorting of my nodes. The K2 search algorithm iteratively builds a graph from a set of independent nodes by starting with the last descendent node and adding all parent nodes that improve the Bayesian score for the graph. I evaluate the impact of capping the maximum number of parents that a given child node can have, but find that I can achieve a Bayesian score that is as good or better each time I increase this maximum. I therefore decide not to limit the maximum number of parents in my final implementation. See Figure 1 for this assessment on the "small" dataset.

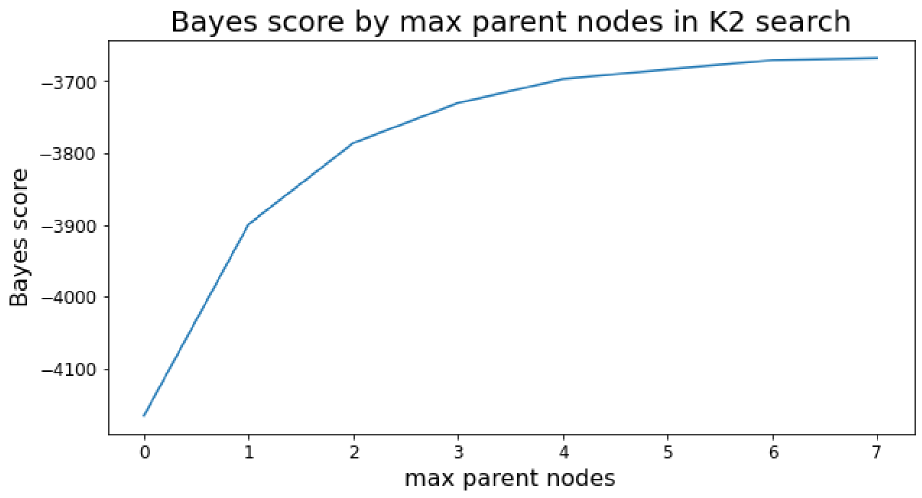

Figure 1: Bayes score by the number of maximum parents allowed in K2 search

I next implement a local search algorithm on the output graph from my K2 search algorithm. This algorithm explores the graph space by iteratively moving one step in graph space towards a graph with a higher Bayesian score. One step in graph space is defined as either adding a node, removing a node, or flipping a node, all while ensuring that the resulting graph is acyclic. I implement my local search algorithm until it takes **20** random steps in graph space without finding a graph that improves the Bayesian score.

Lastly, I repeat this two-step process **5** times with random topological sortings. While I imagine there is a more effective way to explore graph space, I thought this low-lift approach

	Small	Medium	Large
Bayes score	-3820.777	-42036.246	-421088.345
Graph structure (N, E)	(8, 12)	(12, 17)	(50, 110)
Total time (sec)	1.952	30.824	5535.849
Mean iteration time (sec, 5 iters)	0.000	6.000	1107.000

Table 1: Results from graph search algorithms for each dataset

would allow me to evaluate Bayesian scores for graphs across the graph space. My process concludes by selecting the best graph among the **5** iterations of my two-step process.

Table 1 provides the results of my algorithm for each of the three considered datasets, including the overall Bayes score, graph structure, and time taken to generate the graph (both total time and mean time per random-start iteration)

2. Graphs

Below I provide visual representations of the Directed Acyclic Graphs generated through my algorithm that seeks to find graphs with low Bayesian scores

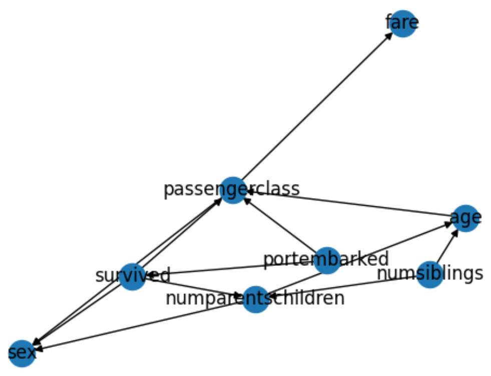

Figure 2: Path with largest Bayes score for the 'Small' dataset

3. Code

The first code block implements my search algorithm. I provide additional code blocks beneath this one, organized in separate sections with additional utility functions to support this code

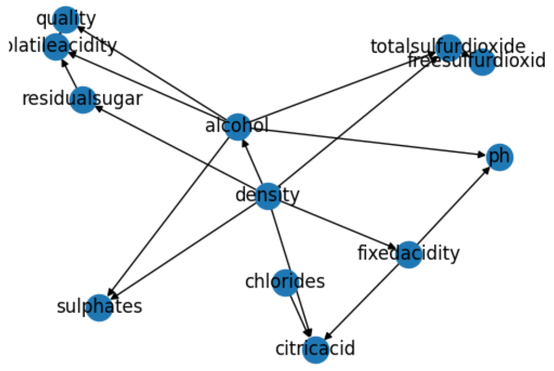

Figure 3: Path with largest Bayes score for the ‘Medium’ dataset

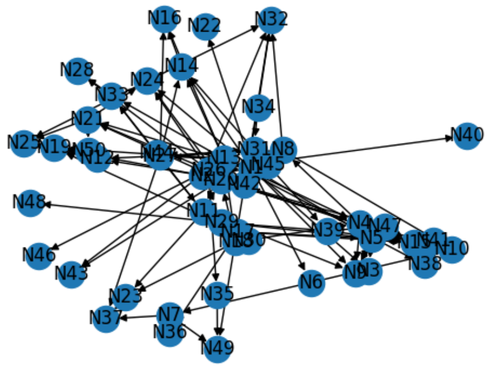

Figure 4: Path with largest Bayes score for the ‘Large’ dataset

```python
import numpy as np
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import time

from k2_search import k2_search # see blocks below
from local_search import local_search # see blocks below
```

```
def read_data(filename):
    # read in data
    raw_data = pd.read_csv(f'./data/{filename}.csv')
    df, nodes = raw_data.values, list(raw_data.columns)

    return nodes, df

def save_results(G, filename):
    # save graph image
    plt.plot(figsize=(15, 15))
    nx.draw(G, with_labels=True)
    plt.savefig(f'./writeup/results/{filename}_path.png')

    # save graph file
    with open(f'{filename}.gph', 'w') as f:
        for edge in G.edges():
            f.write(f'{edge[0]}, {edge[1]}\n')
    return

# graph search algorithm
def random_start_graphsearch(nodes, data, iterations=3):
    time_total = -time.time()
    time_mean_iter = 0
    bscore_best, G_best = -np.inf, np.nan

    for i in range(iterations):
        time_mean_iter -= time.time()

        # shuffle nodes
        rs_idx = np.random.choice(
            np.arange(len(nodes)), size=len(nodes), replace=False)
        rs_nodes = [nodes[r] for r in rs_idx]
        rs_data = data[:,rs_idx]

        # run k2 search
        G_cand, bscore_cand, __ = k2_search(
            rs_nodes, rs_data, max_par_nodes=None)
        # run local search
        G_cand, bscore_cand, __ = local_search(
            G_cand, rs_nodes, rs_data, max_iter=20)

        if bscore_cand > bscore_best:
            bscore_best = bscore_cand
            G_best = G_cand

        time_mean_iter += time.time()

    time_total += time.time()
    time_mean_iter /= iterations
    return G_best, bscore_best, (time_total, time_mean_iter)


def main():
    if len(sys.argv) != 3:
        raise Exception("usage: python project1.py <filename prefix>
    n_iterations")

    filename = sys.argv[1]
    n_iterations = int(sys.argv[2])
    results = {}

    # read data
    print('reading data...')
    nodes, df = read_data(filename)

    # get graph with best bayescore
    print('getting best graph...')
    G, bayescore, total_times = random_start_graphsearch(
        nodes, df, iterations=n_iterations, local=True)

    # save results
    print('saving results...')
    save_results(G, filename)

    # print table output
    results['Bayes score'] = [np.round(bayescore, 3)]
    results['Graph structure (N, E)'] = [f'({len(G.nodes)}, {len(G.edges)})']
    results['Total time (sec)'] = [np.round(total_times[0], 3)]
    results[f'Mean iteration time (sec, {n_iterations} iters)'] = [
        np.round(total_times[1])]
    print(pd.DataFrame(results).T.to_latex())

if __name__ == '__main__':
    main()
```

3.1 K2 search

```
import numpy as np
import networkx as nx
import scipy as sp
import time
import itertools

def get_M(G, data, nodes, query):
    parents = list(G.predecessors(query))

    # get all parent instances
    max_par_insts = np.zeros(len(parents))
    for i, pn in enumerate(parents):
        max_par_insts[i] = np.max(data[:,nodes.index(pn)])
    par_inst = np.array(list(
        itertools.product(*[np.arange(start=1, stop=mp+1) for mp in
    max_par_insts])), dtype=int)

    # make M matrix and alpha matrix
    max_query_inst = np.max(data[:, nodes.index(query)])
    M = np.zeros((len(par_inst), max_query_inst), dtype=int)
    alpha = np.ones_like(M)

    # for each parent instant find data rows that exist and populate M
    par_node_idx = [nodes.index(n) for n in parents]
    query_node_idx = nodes.index(query)
    for i, pi in enumerate(par_inst):
        # get rows in data with the parent instance
        d_idx = (data[:, par_node_idx] == pi).all(axis=1)
        # get query value for those rows
        query_vals = data[d_idx, query_node_idx]
        # increment query values
        for j in query_vals:
            M[i,j-1] += 1

    return M, alpha

def get_bscore(G, data, nodes):
    score = 0
    for query in nodes:
        M, alpha = get_M(G, data, nodes, query)
        score += np.sum(sp.special.loggamma(M+alpha))
        score -= np.sum(sp.special.loggamma(alpha))
        score += np.sum(sp.special.loggamma(np.sum(alpha, axis=1)))
        score -= np.sum(sp.special.loggamma(np.sum(alpha, axis=1) + np.sum(M,
     axis=1)))

    return score

def k2_search(nodes, data, max_par_nodes=None):
    time_total = -time.time()

    if max_par_nodes is None:
        max_par_nodes = len(nodes)

    # initialize graph
    G = nx.DiGraph()
    G.add_nodes_from(nodes)
    bscore_best = get_bscore(G, data, nodes)

    # first node is oldest anscestor, loop through all potential children
    for i, chi_node in enumerate(nodes[1:]):
            bscore_cand, par_node_cand = -np.inf, np.nan
            par_node_count = 0

            # loop through all potential parents of a given child
            # add new edges that improve bscore until either
            # max nodes are hit or there aren't any more parents
            while par_node_count < np.min((max_par_nodes, len(nodes[0:i+1]))):
                for par_node in nodes[0:i+1]:
                    if not G.has_edge(par_node, chi_node):
                        G.add_edge(par_node, chi_node)
                        bscore_cand_trial = get_bscore(G, data, nodes)
                        if bscore_cand_trial > bscore_cand:
                            bscore_cand = bscore_cand_trial
                            par_node_cand = par_node
                        G.remove_edge(par_node, chi_node)

                # add the best parent
                if bscore_cand > bscore_best:
                    bscore_best = bscore_cand
                    par_node_count += 1
                    G.add_edge(par_node_cand, chi_node)
                else:
                    break

        time_total += time.time()

        return G, bscore_best, time_total
```

3.2 Local search

```
    import numpy as np
    import networkx as nx
    import time

    from k2_search import get_bscore

    def move_to_rand_neighbor(G):
        n = len(G.nodes)
        i, j = np.random.choice(G.nodes(), size=2, replace=False)
        G_new = G.copy()

        # remove edge if there is one
        if G_new.has_edge(i, j):
            G_new.remove_edge(i, j)
            # with probability 0.5, add edge in opposite direction
            if np.random.random() > 0.5:
                G_new.add_edge(j, i)
        # add edge if there is not one
        else:
```

```
        G_new.add_edge(i, j)
    return G_new

def is_cyclic(G):
    n_cycles = len(sorted(nx.simple_cycles(G)))
    return n_cycles != 0

def local_search(G, nodes, data, max_iter=25):
    time_total = -time.time()
    bscore_best = get_bscore(G, data, nodes)
    iters = 0

    while iters < max_iter:
        G_new = move_to_rand_neighbor(G)

        if is_cyclic(G_new):
            bscore_cand = -np.inf
        else:
            bscore_cand = get_bscore(G_new, data, nodes)

        if bscore_cand > bscore_best:
            bscore_best = bscore_cand
            G = G_new
            iters = 0
        else:
            iters += 1

    time_total += time.time()

    return G, bscore_best, time_total
```