Project 1: Bayesian Structure Learning

Tae Yang taeyang@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

1.1 Bayesian Score Computation

The Bayesian score quantifies how well a graph G (representing the network structure) models a dataset D. The score is based on the posterior probability of the graph given the data, using the following equation:

$$\log P(G \mid D) = \log P(G) + \sum_{i=1}^{n} \sum_{j=1}^{q_i} \left(\log \frac{\Gamma(\alpha_{ij0})}{\Gamma(\alpha_{ij0} + m_{ij0})} + \sum_{k=1}^{r_i} \log \frac{\Gamma(\alpha_{ijk} + m_{ijk})}{\Gamma(\alpha_{ijk})} \right) \tag{1}$$

This computation is implemented in the `bayesian_score` method of the code. The accuracy of the calculation is verified by confirming that the resulting score matches the score provided in the example. Additionally, after running the structure learning for all datasets, the resulting scores are the same as those on the leaderboard.

However, the computation involves counting the occurrences of various parent-child configurations and calculating log probabilities, which can be computationally expensive for large datasets. As the number of variables increases in the training dataset, the compute time increases significantly.

1.2 Optimization Techniques in Bayesian Score

To efficiently compute the Bayesian score, vectorized operations using `NumPy` are employed. Vectorization allows for the simultaneous processing of array data, significantly speeding up operations such as counting occurrences and filtering rows based on parent configurations. Instead of using explicit loops, operations like counting the occurrences of each state of a child node are handled using `np.bincount()`.

For example, the count of each state k of a node i without parents is computed as:

$$\text{counts}[k] = \sum_{j=1}^{n} \mathbb{I}(X_{ij} = k) \tag{2}$$

where $\mathbb{I}$ is the indicator function and X_{ij} is the observed value of node i for sample j. The marginal score is computed as:

$$\text{score} = \sum_{k=1}^{r_i} (\Gamma(1 + \text{counts}[k]) - \Gamma(1)) + \Gamma(r_i) - \Gamma(r_i + n) \tag{3}$$

where n is the total number of samples in the dataset.

To further optimize performance, the algorithm uses memoization to cache previously computed scores for specific node-parent configurations. Since the graph is modified iteratively, only small parts of the graph change between steps, so caching avoids unnecessary recomputation of scores for unchanged substructures. This significantly reduces the computation time for large datasets.

1.3 Local Search Algorithm

The local search method implemented in `local_search` is a greedy optimization technique used to iteratively improve the Bayesian network structure by modifying the graph. Starting with an initial graph containing all nodes but no edges, the algorithm explores possible improvements by adding, removing, or reversing edges between nodes.

For each pair of nodes (u, v), the algorithm attempts to:

- Add an edge if it improves the Bayesian score,
- Remove an existing edge if doing so increases the score,
- Reverse the direction of an edge and recalculate the score.

If any of these modifications improves the score, the change is accepted; otherwise, the graph is reverted to its previous state. The graph must remain a *Directed Acyclic Graph (DAG)* throughout the search, so any changes that introduce cycles are reverted immediately. The local search terminates when no further improvement in the score can be found, indicating that a local optimum has been reached.

The greedy approach ensures that only changes that improve the Bayesian score are accepted, but it systematically explores all potential edge modifications. This thorough exploration helps to find an optimal or near-optimal structure for the Bayesian network.

1.4 Greedy Edge Modifications and DAG Validation

After each edge modification, the structure is checked to ensure that the graph remains a DAG. This validation is essential since Bayesian networks require acyclic structures. The score computation is then repeated, and if the new graph provides a better fit to the data, the modification is accepted. Otherwise, the change is reverted, and the search proceeds to the next possible modification.

2. Graphs

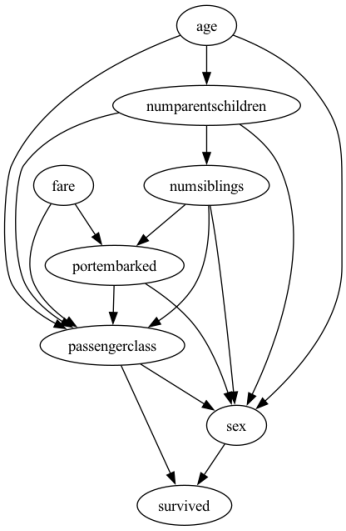

Figure 1: Graph learned from `small.csv`; Running time: 0.07 s; Bayesian score: -3821.4893586463472

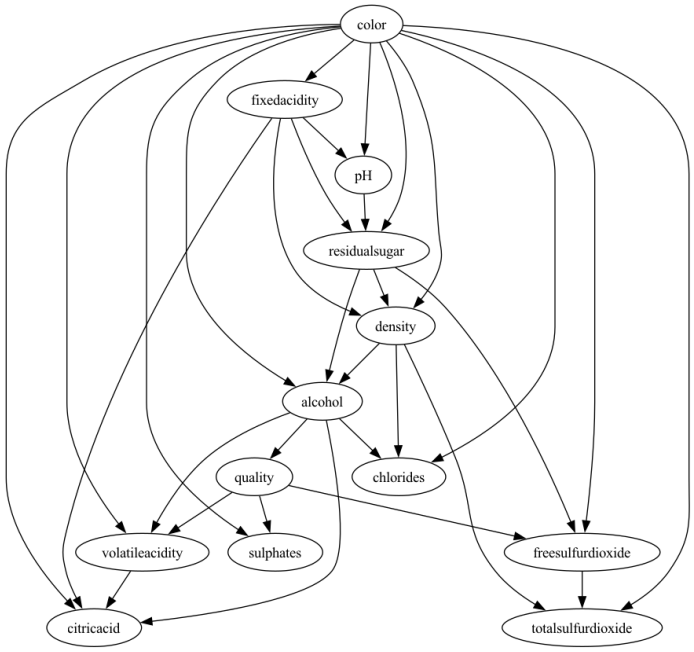

Figure 2: Graph learned from `medium.csv`; Running time: 4.60 s; Bayesian score: -96911.23843553875

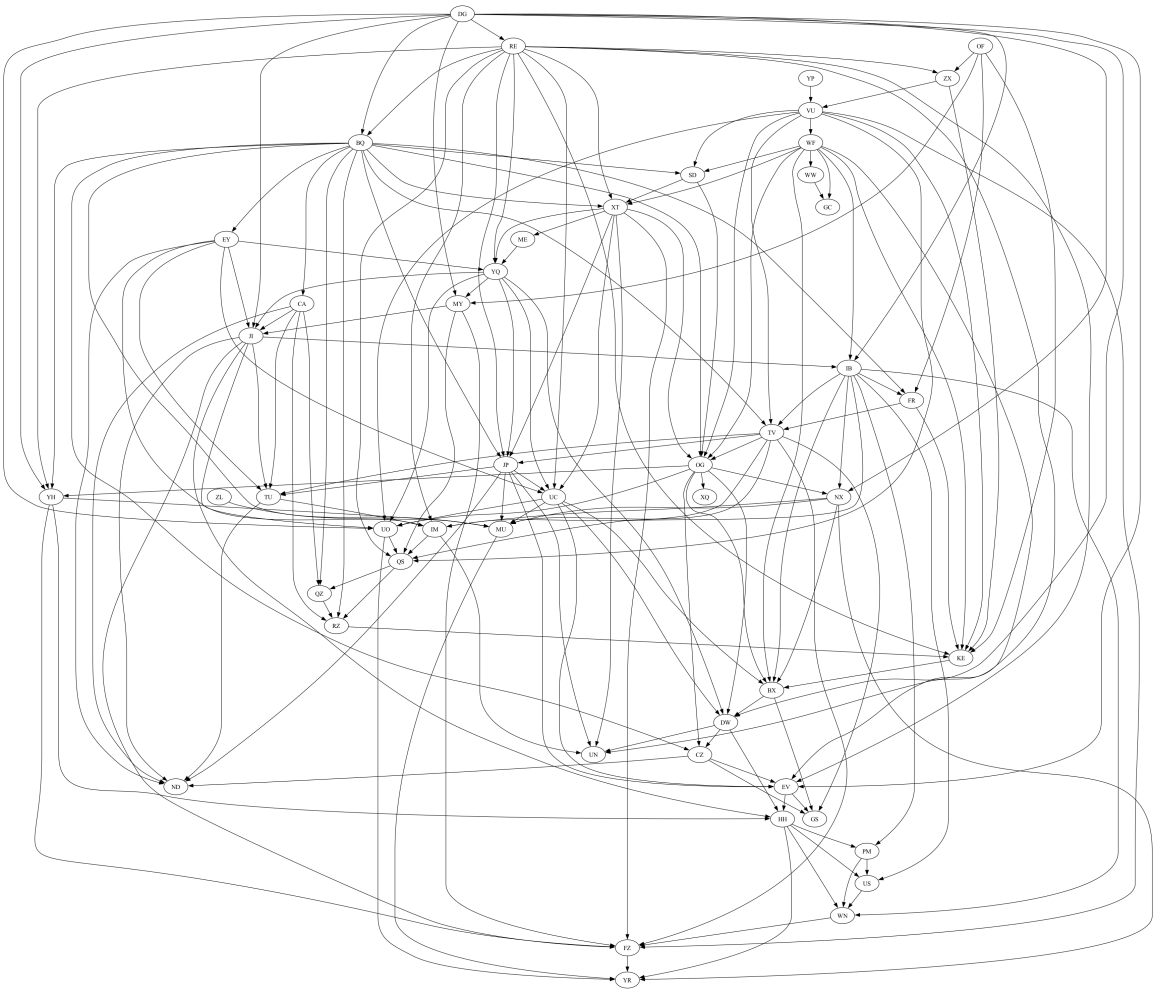

Figure 3: Graph learned from `large.csv`; Running time: 108.43 s; Bayesian score: -416824.54704279185

3. Code

```
import sys

import networkx

import pandas as pd
import itertools
import numpy as np
from scipy.special import gammaln
import networkx as nx
import matplotlib.pyplot as plt
import time
import random
import math
from networkx.drawing.nx_pydot import write_dot
import os
import graphviz

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

    #We are using a Uniform Dirichlet Prior

    data = read_csv(infile)

    # Initialize the graph with no edges (nodes only)
    graph = nx.DiGraph()
    graph.add_nodes_from(data.columns)  # Add the variable names as nodes

    # Track the start time
    start_time = time.time()

    # Choose a structure learning algorithm below
    best_graph = local_search(graph, data)
    # best_graph = hill_climbing_search(graph, data)
    # best_graph = simulated_annealing_search(graph, data)

    # Track the end time
    end_time = time.time()

    # Calculate the time taken and print it
    elapsed_time = end_time - start_time
    print(f"Structure learning completed in {elapsed_time:.2f} seconds.")

    save_gph(best_graph, outfile)
    pass
```

```
# Function to read the .csv file and return data as a pandas DataFrame
def read_csv(file_path):
    data = pd.read_csv(file_path)
    return data

def local_search(graph, data):
    best_graph = graph.copy()
    best_score = bayesian_score(best_graph, data)
    improving = True

    while improving:
        improving = False
        current_score = best_score

        print("Local search in progress. Current bayesian score: ",
    current_score)

        # Try adding, removing, or reversing edges between each pair of nodes
        for u, v in itertools.permutations(graph.nodes(), 2):
            if best_graph.has_edge(u, v):
                best_graph.remove_edge(u, v)
                new_score = bayesian_score(best_graph, data)
                if new_score > current_score:
                    current_score = new_score
                    improving = True
                else:
                    # Revert change if it doesn't improve the score
                    best_graph.add_edge(u, v)
            else:
                # Try adding or reversing the edge
                best_graph.add_edge(u, v)
                new_score = bayesian_score(best_graph, data)
                if new_score > current_score and nx.is_directed_acyclic_graph
    (best_graph):
                    current_score = new_score
                    improving = True
                else:
                    # Revert change if it doesn't improve the score
                    best_graph.remove_edge(u, v)

        if improving:
            best_score = current_score

    return best_graph

# Function to save a .gph file from a NetworkX graph
def save_gph(G, file_path):
    with open(file_path, 'w') as f:
        for parent, child in G.edges():
```

```
            f.write(f'{parent},{child}\n')

# Function to parse the .gph file
def parse_gph(file_path):
    graph = nx.DiGraph()
    with open(file_path, 'r') as file:
        for line in file:
            parent, child = line.strip().split(',')
            graph.add_edge(parent, child)
    return graph

def visualize_gph(gph_file_path, dot_file_path, png_file_path):
    G = parse_gph(gph_file_path)

    # Write the graph to a .dot file using NetworkX
    write_dot(G, dot_file_path)

    # Use GraphViz to convert the .dot file to a .png
    os.system(f"dot -Tpng {dot_file_path} -o {png_file_path}")
    print(f"Graph saved as {png_file_path}")

def bayesian_score(G, data, cache={}):
    total_score = 0

    # Convert DataFrame to NumPy array for efficient processing
    data_np = data.to_numpy()

    for child in G.nodes():
        parents = tuple(sorted(G.predecessors(child)))  # Get the parents of
    the child node, sorted to create a unique key

        # Create a unique cache key based on the child and its parents
        cache_key = (child, parents)

        if cache_key in cache:
            total_score += cache[cache_key]
        else:
            score = compute_local_bayesian_score_np(data, child, parents,
    data_np)
            cache[cache_key] = score  # Cache the score for future use
            total_score += score

    return total_score

def compute_local_bayesian_score_np(data, child, parents, data_np):
    """
    Computes the Bayesian score for a single node (child) and its parent set
    using NumPy for optimization.
    """
    total_score = 0
```

```
        child_idx = data.columns.get_loc(child)
        ri = np.max(data_np[:, child_idx])

        if len(parents) == 0:
            # Vectorized computation for marginal distribution
            counts = np.bincount(data_np[:, child_idx], minlength=ri + 1)[1:]  #
        Counts for each state of the child (excluding 0)
            total_score += np.sum(gammaln(1 + counts) - gammaln(1))  #
        Pseudocount = 1
            total_score += gammaln(ri) - gammaln(ri + len(data_np))

        else:
            parent_idxs = [data.columns.get_loc(p) for p in parents]
            parent_combinations = np.array(list(itertools.product(*[range(1, np.
        max(data_np[:, idx]) + 1) for idx in parent_idxs])))

            for parent_inst in parent_combinations:
                # Vectorized filtering for parent configurations
                mask = np.all([data_np[:, parent_idxs[i]] == parent_inst[i] for i
         in range(len(parents))], axis=0)
                subset = data_np[mask]

                m_ij0 = len(subset)
                total_score += gammaln(ri) - gammaln(ri + m_ij0)

                # Counts for each state of the child, given this parent
        configuration
                if len(subset) > 0:
                    child_counts = np.bincount(subset[:, child_idx], minlength=ri
         + 1)[1:]
                    total_score += np.sum(gammaln(1 + child_counts) - gammaln(1))

        return total_score

    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]

        # Comment the following line to only visualize a gph file.
        compute(inputfilename, outputfilename)

        print("Bayesian score: ", bayesian_score(parse_gph(outputfilename),
        read_csv(inputfilename)))

        visualize_gph(outputfilename, outputfilename[:-3]+"dot", outputfilename
        [:-3]+"png")
```

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