Project 1 - Structure learning of discrete Bayesian Networks

Thomas Hossler

thossler@stanford.edu
Departement of Geological Sciences, Stanford University

Keywords

structure learning, bayesian networks, K2, local search, bayesian score function

ABSTRACT

For this first project, the objective was to code an algorithm to learn the best structure to fit data sets provided. To quantify the fit, a metric is necessary and a Bayesian score function has been implemented. Two algorithms have been built: a K2 algorithm and a local search algorithm. The local search algorithm have been used to refine the results of the K2 but did not succeed as it appears to be stuck in local minima.

1. Introduction

Bayesian networks are compact representation of joint probabilistic distributions. They are represented as Directed Acyclic Graphs (DAGs), graphs consisting of nodes and directed edges. Each node represents a random variable. In this project, only discrete random variables are considered. The directed edges between nodes represent the dependence of one variable to another. Whereas in some cases this relationship is known and the structure of the graph is therefore available, in other cases the relationships between the random variables is unknown. This challenge is defined as a structure learning problem. In a structure learning problem, a set of data is used to understand the connections between the different variables. Structure learning algorithms consist in iterating an existing structure to reach the best fit possible. A metric is required to do so and a Bayesian Score function is used in this project.

In this project, a structure learning algorithm is presented and tested on three different data set. First, the score function and the algorithms implemented are introduced. The resulting Bayesian Networks are then exposed. Finally, possible improvements are discussed. The code is available in the last section.

2. Methods

In the section, data, the algorithms as well as the workflow implemented are presented. The K2 and local search algorithm are susceptible to find local optima instead of global ones. Therefore, a workflow using both methods has been built.

2.1. Datasets

Three data sets of discrete random variables have been used to test the algorithm. The **Titanic** set is made of 8 random variables and 889 observations. The **whitewine** set is made of 12 random variables and 4898 observations. The **schoolgrades** set is made of 28 random variables and 2287 observations.

2.2. Bayesian Score function

In this project, a maximum likelihood approach is used. It involves finding the structure **G** that maximize $P(G \mid D)$ where **D** is the available data.

After some non-trivial algebra, finding the best **G** is equivalent to finding the **G** that maximizes the following expression:

$$\ln P(G \mid D) = \ln P(G) + \sum_{i=1}^{n} \sum_{j=1}^{q_i} \ln \frac{(\gamma(\alpha_i j0))}{\gamma(\alpha_{ij0} + m_{ij0})} + \sum_{k=1}^{r_i} \ln \frac{\gamma(\alpha_{ijk} + m_{ijk})}{\gamma(\alpha_{ijk})} \tag{1}$$

where r_i is the number of instantiations of the random variableX_i, q_i the number of instantiations of the parents of X_i, n the number of nodes (ie, random variables), α the parameter of the Dirichlet distribution and m_{ijk} the number of times $X_i = k$ given the jth instantiation of the parents of X_i.
This metric is used to quantify the fit graph created by the following algorithms.

2.3. K2 algorithm

The K2 algorithm implemented here is described in the following pseudocode. It takes a random sorting of nodes and a maximal number of parents as inputs. The K2 algorithm highly depends on the initial sorting of the nodes as it does not act recursively.

```
inputs: data, maximum number of parents, sorting of n nodes
create an empty graph G
for i in 1:n
        for j in i+1:n
                create an edge between i and j: G'
                if the score is improved and the graph acyclic, keep the edge and
                    update the score
                G <- G'
```

2.4. Local search algorithm

The local search algorithm starts with an existing DAG with connected nodes. By creating neighboring graphs, the algorithm seeks to improve the existing structure. Neighboring graphs are graphs which are one operation away from the existing graph (adding, removing or flipping an edge). The Local Search algorithm implemented here is described in the following pseudocode. A tabu search has been added to the local search by creating a tabu list, which forbid the code to act on nodes that were changed a few iterations ago. The computational time of the local search algorithm depends on the size of the tabu list as well as the number of iterations.

```
inputs: DAG, data, size of the tabu list, number of iterations allowed
old score = score(DAG)
        pick a pair of nodes that is not in the tabu list.
        add them to the tabu list
        perform one of the three operations DAG -> DAG'
        check if the graph is acyclic
        check if score(DAG') > score(DAG)
                DAG <- DAG'
        if not, move back to the previous graph.
        repeat.
```

2.5. Workflow

The structure learning problem can be tackled in several different approach. For this project, rather than trying the K2 algorithm for all existing sorting of nodes (which grow quite fast with the number of variables), the two algorithm have been combined in the following manner:

```
generate N possible sorting of the nodes
G empty graph
for each sorting i,
        G' = K2(sorting i)
        if score(G') > score(G)
                G = G'

apply the local search algorithm to the best graph of K2
```

3. Results

The number of sorting for the K2 algorithm has been fixed to 10,000 for computational reasons. The number of parents has been fixed to two. The results are presented in the figures below, with the corresponding scores and running times in caption.

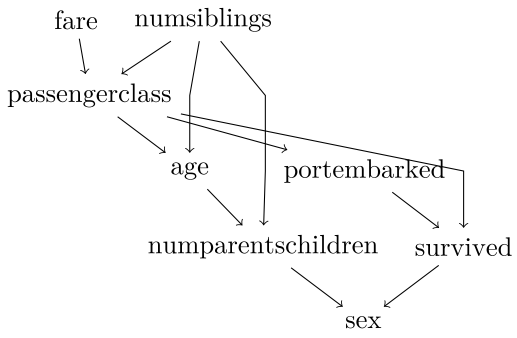

Figure 1: **Titanic** data set. Run time = 489.159 seconds / Score = -3808.842

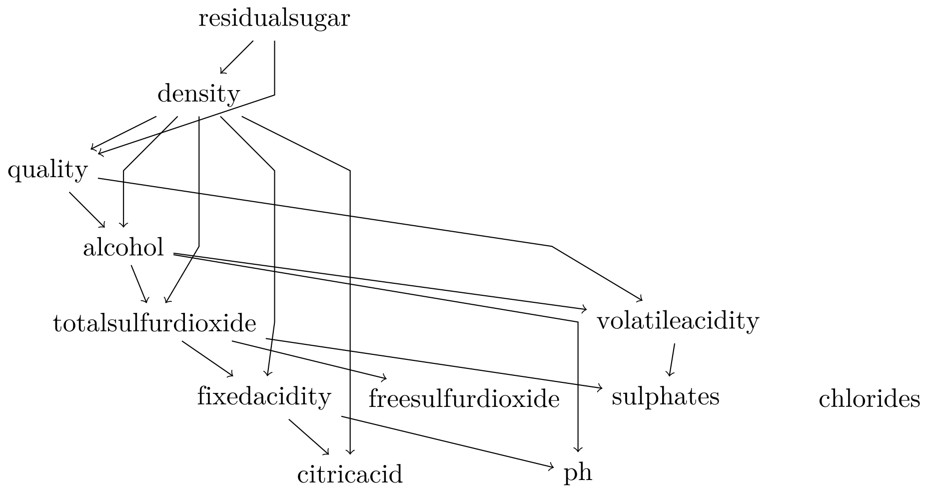

Figure 2: **White wine** data set. Run time = 4147.848 seconds / Score = -42175.078

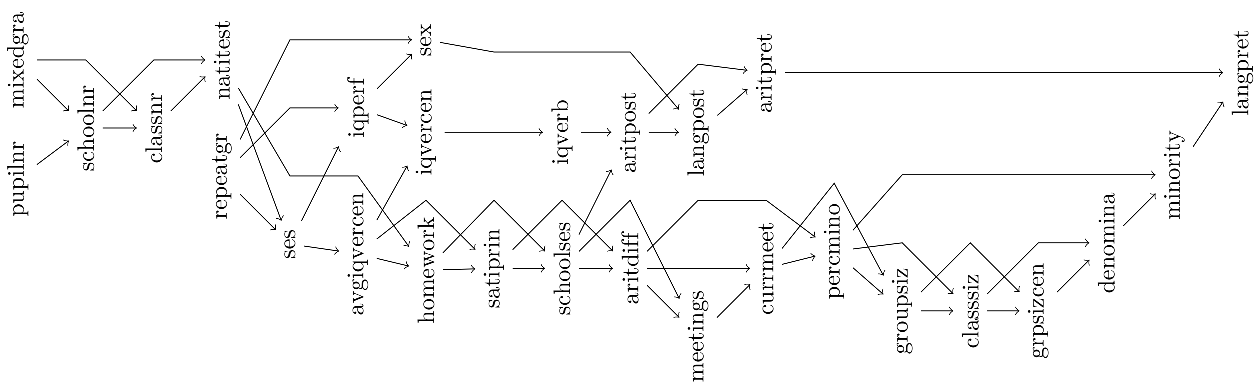

Figure 3: **School grades** data set. Run time = 11817.225 seconds/ Score = -57273.065

The results for the **Titanic** data set presented Figure 1 shows expected relationship between the variables. For example, *passenger class* and *fare* are connected, or the *number of siblings* and the *age*. Everyone remember the eponymous movie and should not be surprised with the relationship between *passengerclass* and *survived* or between *survived* and *sex* (Women and children first).

The author's knowledge about wine, despite his french origin, is not sufficient to lead to similar intuitive reasoning for the **White wine** data set, whose network is presented Figure 2. A relative experience in drinking various beverages with a wide spectrum in alcohol by volume helps to understand the relationship between *alcohol* and *acidity*. However, one could have expected that *quality* would be the last child of the DAG.

The results of the **School grades** data set are presented Figure 3. The author does not want to influence the grading team by offering any hasty conclusion regarding what his grade should be based on these results. Therefore, the author will not propose an intuitive reasoning of this graph.

It appears that the local search does not improve the results of the K2 algorithm, even by increasing the number of iterations and the size of the tabu list.

4. Discussion

The lack of improvement from the local search algorithm can be explained by the fact that no worsened move were allowed, i.e. among the three possible actions, only the ones that improve the score were considered. This could lead to being stuck in a local minima, which is strengthen by the fact that the result of K2 is potentially already a local minima.

The influence of the different parameters such as the number of parents for K2 or the size of the tabu list for the local search have not been studied. Therefore, some improvement in the structure might be obtained by tweaking these parameters. In addition to studying the sensitivity, some improvement can be implemented in the workflow. For example, the K2 search could be performed over the all possible combinations of the sorting. Moreover, by only picking the best resulting graph of the K2 algorithm to apply the local search, we are restricting the search space and might miss some DAGs that could a better fit after a local search.

5. Conclusion

For this project, the author presented a method to learn the structure of a discrete Bayesian Network from given data sets. The method combined K2 and local search algorithms, using a bayesian score function as a metric. However, it appears that the application of the local search algorithm to the best result of the K2 process does not improve the score. This could be fixed by relaxing the local search to accept worsened moves in order to step away from a local minima.

6. Code

It must be noted that this project was the first time ever the author manipulated the Julia language. Despite a strong experience in Matlab and other programming languages, some of the functions might appear as "obscure" or "disgustingly coded". The author apologize in advance for any shocking line of code. The code is definitely not optimized, but it runs. The author found it more exciting and interesting to look at the various algorithm than to slightly improve the performance of each one of them.

6.1. Bayesian score function

```
function bscore(dag,data)
        % bayesian score function
    N = statistics(dag,data); % array of arrays
    n = size(data,2)
    parent_list = badj(dag); % parents of each node
    ncategories = Array(Int, n);

        % look for the number of instations for each variable
    for i = 1 : n
        if isempty(parent_list[i]) % TO FIX
            q = 1;
        end
        ncategories[i] = infer_number_of_instantiations(convert(Vector{Int}, data[i
            ]));
    end
```

```
    % create alpha, matrix of ones - same size of N
        % since a uniform prior is used
    A = Array(Array{Int}, n);
    for i = 1 : n
        B = parent_list[i]; l = length(B);
        dim2 = 1;
        for j = 1 : l
            dim2 = dim2 * ncategories[B[j]];
        end
        if dim2 == 0
            dim2 =1;
        end
        A[i] = ones(ncategories[i],dim2);
    end
    alpha = A;

        % two dummy functions to sum over the array of array
        % could be easier with sum and a better knowledge of julia
        % sum m_ijk and alpha_ijk
    function sumlgamma(A)
        ret = 0;
        l = length(A)
        for k = 1 : l
            t = length(A[k]);
            for m = 1 : t
                ret = ret + lgamma(A[k][m]);
            end
        end
        ret
    end

        % sum only over one dimension
        % sum m_ij0 and alpha_ij0
    function sumArray(A)
        n = length(A);
        ret = Array(Array{Int}, n);
        for i = 1 : n
            ret[i] = sum(A[i],1);
        end
        ret
    end
        % final sum
    sumlgamma(alpha + N) - sumlgamma(alpha) + sumlgamma(sumArray(A)) -  sumlgamma(
        sumArray(A)+sumArray(N))
end
```

6.2. K2 algorithm

```
function kangchenjunga(data,max_parents,rp)
        % in honor to the third highest mountain of the world
        % rp = random permutation of the nodes
        % max_parents = maximal number of parents

    nodes = names(data); n  = size(data,2);
    graph =  DiGraph(size(data,2)); % create an unconnected graph
    new_score = 0;  parent_list = zeros(n);
    for i = 1 : n % go through all the nodes
        old_score = bscore(graph,data);
        for j = i+1 : n % the other nodes
            if parent_list[i] < max_parents
                add_edge!(graph,rp[j],rp[i])
                new_score = bscore(graph,data); % new graph score
                if !is_cyclic(graph)
                    if new_score > old_score % better score?
                        parent_list[i] = parent_list[i] +1;
```

```
                    old_score = new_score;
                else
                    rem_edge!(graph,rp[j],rp[i])
                end
            else
                rem_edge!(graph,rp[j],rp[i])
            end
        end
    end

    end
    graph
end
```

6.3. Local search algorithm

```
function graal(graph,data,tabu,ite)
    % inputs:
    % - graph = dag obtained from k2 search
    % - tabu = size of the memory (number of pairs of nodes that the code remember)
    % - number of iterations to try after we found a local minima
    % output:
    % - dag
    numberOfNodes = size(data,2);
    listOfNodes1 = [(i) for i = 1 : numberOfNodes];
    listOfNodes2 = copy(listOfNodes1);

    iterations = 0; ind = 1;
    tabuList = Array(Int, tabu);
    old_score = bscore(graph,data);

    while iterations <= ite
        graphIni = copy(graph);

        if sum(tabuList) > 0 % if the list is not empty, eg first round
                % remove the nodes which are in the tabulist
            for j = 1 : length(tabuList)
                if tabuList[j] > 0
                    k = find(listOfNodes2 .== tabuList[j])
                    deleteat!(listOfNodes2,convert(Int,k[1])); % remove the node
                        from the tabu list
                end
            end
        end

        node1Ind = convert(Int,floor(length(listOfNodes2)*rand()+1)); % randonly
            select two nodes from the remaining
        node1 = listOfNodes2[node1Ind];
        node2Ind = convert(Int,floor(length(listOfNodes2)*rand()+1));
        node2 = listOfNodes2[node2Ind];
        convert(Int,node1); convert(Int,node2);

        % save the nodes in the tabulist
        if ind < tabu
            tabuList[ind] = node1; tabuList[ind+1] = node2;
            ind = ind+2;
        elseif ind == tabu
            tabuList[ind] = node1;
            ind = 1;
            tabuList[ind] = node2;
            ind = ind+2;
        else
            ind = 1;
            tabuList[ind] = node1; tabuList[ind+1] = node2;
            ind = ind+2;
```

```
end

        % check if the node are connected
neighborsNode1 = badj(graph,node1); a = find(neighborsNode1 .== node2);
neighborsNode2 = badj(graph,node2); b = find(neighborsNode2 .== node1);
        % case 1: node2 -> node1
if !isempty(a)
    score1 = bscore(graph,data);
    rem_edge!(graph,node2,node1);% remove the edge
    score2 = bscore(graph,data);
    add_edge!(graph,node1,node2); % inverse the edge
    score3 = bscore(graph,data);
    score = [score1,score2,score3];
    j = find(score .== maximum(score));

    if maximum(score) >= old_score[1]
        if j == 1 % we did not anything
            rem_edge!(graph,node1,node2);
            add_edge!(graph,node2,node1);
            iterations += 1;
            old_score = score[j];
        elseif j == 2
            rem_edge!(graph,node1,node2);
            if is_cyclic(graph) % if the graph is the cyclic we reverse to
                the former one
                add_edge!(graph,node2,node1);
                iterations += 1;
                old_score = score1;
            else
                old_score = score[j];
            end

        else
            if is_cyclic(graph)
                rem_edge!(graph,node1,node2);
                add_edge!(graph,node2,node1);
                iterations += 1;
                old_score = score1;
            else
                old_score = score[j];
            end

        end
    end
% case 1: node1 -> node2
elseif !isempty(b)
    score1 = bscore(graph,data);
    rem_edge!(graph,node1,node2);% remove the edge
    score2 = bscore(graph,data);
    add_edge!(graph,node2,node1); % inverse the edge
    score3 = bscore(graph,data);
    score = [score1,score2,score3];
    j = find(score .== maximum(score));
    %print(score)
    if maximum(score) >= old_score[1]
        if j == 1
            rem_edge!(graph,node2,node1);
            add_edge!(graph,node1,node2);
            iterations += 1;
            old_score = score[j];
        elseif j == 2
            rem_edge!(graph,node1,node2);
            if is_cyclic(graph)
                add_edge!(graph,node1,node2)
                iterations += 1;
```

```
                        old_score = score1;
                    else
                        old_score = score[j];
                    end
                else
                    if is_cyclic(graph)
                        rem_edge!(graph,node2,node1);
                        add_edge!(graph,node1,node2)
                        iterations += 1;
                        old_score = score1;
                    else
                        old_score = score[j];
                    end
                end
            end
        %case 3: node1    node2
        else
            score1 = bscore(graph,data);
            add_edge!(graph,node1,node2);
            score2 = bscore(graph,data);
            rem_edge!(graph,node1,node2);
            add_edge!(graph,node2,node1);
            score3 = bscore(graph,data);
            score = [score1,score2,score3];
            j = find(score .== maximum(score));

            if maximum(score) >= old_score[1]
                if j == 1 % we did not anything
                    rem_edge!(graph,node2,node1);
                    iterations += 1;
                    old_score = score[j];
                elseif j == 2
                    rem_edge!(graph,node2,node1);
                    add_edge!(graph,node1,node2);
                    if is_cyclic(graph)
                        rem_edge!(graph,node1,node2)
                        iterations += 1;
                        old_score = score1;
                    else
                        old_score = score[j];
                    end
                else
                    if is_cyclic(graph)
                        rem_edge!(graph,node2,node1)
                        iterations += 1;
                        old_score = score1;
                    else
                        old_score = score[j];
                    end
                end
            end
        end
        listOfNodes2 = copy(listOfNodes1);
    end
    graph
end
```

6.4. Main code

```
%% 3 functions
% - bscore(graph,data) = bayesian score
% - kangchenjunga(data,max_parents,rp) = K2 algorithm
% - graal(graph,data,tabu,ite) = local search
```

```
tic();
data = titanic; % define the data set
numberOfDAG = 10000;
rp = Array{Int}(numberOfDAG, size(titanic,2));

for i = 1 : numberOfDAG
    rp[i,:] = randperm(size(titanic,2));
end
old_score = -Inf;
tabu = 4;ite = 10;
max_parents = 2;
new_score = 0;

% for each permutation, run K2
for i = 1 : numberOfDAG
    old_graph = kangchenjunga(data,max_parents,rp[i,:]);
    new_score = bscore(old_graph,data);
    if new_score >= old_score % the new graph is better
        graphK2 = copy(old_graph);
        old_score = new_score;
    end

end
% run the Local Search
graphLS = graal(graphK2,data,tabu,ite);
time = toc();
```