Structure Learning with k2

Aaron Havens

September 23, 2017

1 A *Succinct* Description of k2

Bayesian nets are probabilistic graphical models that are extremely useful for modeling causal relationships between variables. It would be great if every process had a true dependency structure that is explained by a finite number of variables. However, in most cases, we don't know anything about this structure and all we have is copious amounts of data. Luckily there are only $G(n) = \sum_n^{k=1}(-1)^{k+1}\binom{n}{k}2^{k(n-k)}G(n-k)$ possible directed acyclic graphs for n nodes. This makes exhaustive search completely intractable. Let us instead look at the topological order space, where there are *only* $G(n) = n!$ possible states or 2^n if you're clever. This is a space we can, to some extent, exhaust or atleast sample sufficiently. Although a DAG belonging to a topological ordering isn't unique, maybe we can somehow search locally with a heuristic or *score*. The scoring function most commonly introduced in these problems is Bayesian Dirichlet score (BD), that assumes a Dirichlet distribution with a uniform prior over discrete state variables (k2) [1]. This function is great because its decomposable which means we can maximize it for a single variable and its parent nodes. This is done by using the log of BD as a summation which is also numerically convenient. Where $\alpha_{i,0}$ and $m_{i,j,0}$ are sums of the priors and sum of frequency counts respectively [1].

$$k2(B, \mathbf{M}) = ln(P(B)) + \sum_i \sum_j [ln(\Gamma(\alpha_{i,0})) - ln(\Gamma(m_{i,j,0} + \alpha_{i,0})) + \sum_k ln(\Gamma(m_{i,j,k} + 1))] \tag{1}$$

Given a topological ordering, we can iteratively check each node to see if adding a parent node edge would increase the k2 score. The end score is the

sum of all log scores of each node (in the above form).

1.1 Other Strategies to Help Super Exponentialness

Assuming that k2 does a good job at finding local maximum scores given a topological ordering we can just sample the order space and obtain a descent result sub-optimal result, even for larger networks (12 or 28 variables). A really simple and intuitive way to sample this space is to randomly assign permutations of variable orders [2]. With random order samples, we aren't guaranteed to exhaust the space, but we obtain distributed samples regardless of how many variables. A random sample is just as good as any other at this point. We also limit maximum number of parents for any given node to a specific range and iterate over that range for each ordering. There are plenty of other way to obtain more optimal results, like pruning and order admissible heuristic search, but random restart will provide a satisfy first attempt.

2 Results

The following section contains the results random restart iterations. We operated on three data sets. We'll go over the final score of each graph, the time it took. achieved with k2 heuristic search and Lastly we'll take a look at each produced graph (some more hideous than others) and try to intuitively explain what's going on with the variables.

Data Set	# Variables	k2 Score	# Restarts	Time per Iteration [s]
Titanic	8	-2.9246e03	1000	.717318
White Wine	12	-3.5053e04	100	8.64
School Grades	28	-4.0265e04	10	265.29

Due time constraints, as the variable number increased, the number of restart iterations decreased, sampling less of the ordering space. Scores for the titanic set were near optimal, but the other two sets were definitely less optimal due to their space size.

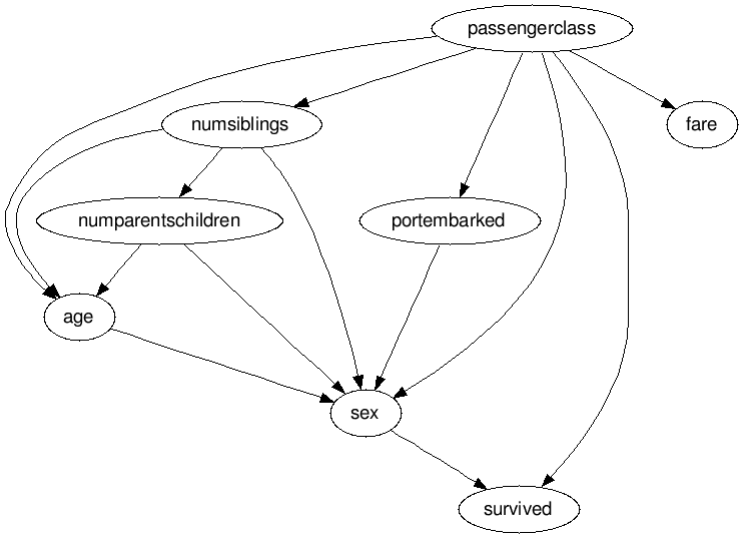

Figure 1: The titanic set seems to be very intuitive. The state of survival was largely dependent on sex and passenger class which largely makes sense, after all they supposedly rescued women and children first. I'm sure the wealthier passengers were rescued with more exigence as well. It also interesting that fare is only dependent on passenger class and so is port embarked.

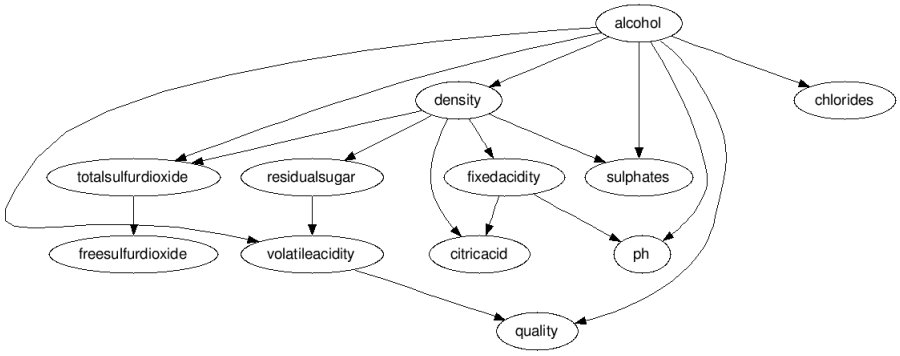

Figure 2: The White wine data set is a bit more complex than the titanic set. We see some very interesting connections between quality, alcohol, and volatile acidity. Interestingly enough, volatile acidity is what gives wine its vinegar characteristic and flavor and is carefully monitored in high quality wines (k2 is really a wine snob) [3].

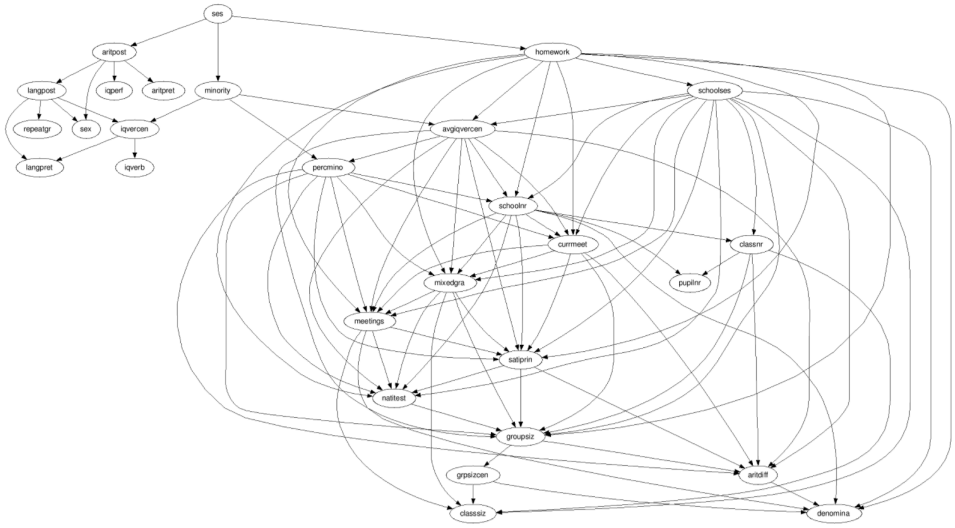

Figure 3: The School grades data is terrible complex. We sampled the space very poorly for this data set. However, there are some very interesting connections to observe. It seems that there is a high dependency on homework as well as supplemental education services (ses) which makes for more engaged students perhaps. There exist very prominent fork structures at the minority node, perhaps suggesting minority students have very different education experiences and different variable dependencies that affect their success.

References

[1] A. M. Carvalho *Scoring function for learning Bayesian networks* , Instituto de telecomunicacoes, 2009.

[2] M. Kochenderfer *Decision Making Under Uncertainty*, *MIT Press*, 2015.

[3] A. L. Bannon *The Perils of Volatile Acidity*,*WineMaker Magazine*, 2006

A Python code

```
import collections
import csv
import matplotlib.pyplot as plt
import time
#Outputs graph edges to .gph file
def graph_out(dag,filename,mapping):
with open(filename, 'w') as f:
for i in range(np.size(dag[0])):
for j in range(np.size(dag[0])):
if(dag[i][j] == 1):
out_string = mapping[i] + ', ' + mapping[j] + '\n'
f.write(out_string)

#Creates a dictionary of node index to category strings
def map_categories(categories):
mapping = {}
for i in range(len(categories)):
mapping[i] = categories[i][1:len(categories[i])-1]
return mapping

#Given a target, find every instance index in array
def find(arr,target):
array = np.array([],dtype='int64')
for i in range(np.size(arr)):
if(arr[i] == target):
array = np.append(array,i)
return array
```

```
#Ln gamma function ln((x-1)!) ->  ln(0) + ln(1) + ... + ln(x-1)
def ln_gamma(x):
return sum(np.log(range(1,int(x))))

#Construct a data structure that stores the possible states (col) for each variabl
#Also returns a range vector that stores the number of states for each variable
def get_dim_range(_data, vec):
count_n = 0
d = np.size(vec[0,:])
dim_length = np.zeros((1,d),dtype = 'int64')
t = -1
#Count number of states
for q in range(d):
temp_vec = np.unique(_data[:,vec[:,q]])
x = temp_vec.reshape(1,np.size(temp_vec))
temp_vec = x
if(temp_vec[:,0] == -1):
temp_vec = np.empty()
range_n = np.size(temp_vec)
dim_length[0,q] = range_n
t += 1
#Assign zeros to the end to create valid matrix dimensions.
if(count_n == 0):
count_n = range_n
dim = np.zeros((d,count_n),dtype = 'int64')
dim[t,:] = temp_vec
elif(count_n >= range_n):
dim[t,:] = np.concatenate((temp_vec,np.zeros((1,count_n - range_n))),axis=1)
elif(count_n < range_n):
dim = np.concatenate((dim,np.zeros((d,range_n - count_n))))
dim[t,:] = temp_vec
return dim,dim_length


def score(blob,var,var_parents):
score = 0
n = blob.n_samples
```

```
dim_var = blob.var_range_length[0,var]
range_var = blob.var_range[var,:]
r_i = dim_var
data_o = blob.data
used = np.zeros(n,dtype='int64')
d = 1
#Get first unproccessed sample
while(d <= n):
freq = np.zeros(int(dim_var),dtype='int64')
while(d <= n and used[d-1] == 1):
d += 1;
if(d > n):
break
for i in range(int(dim_var)):
if(range_var[i] == data_o[d-1,var]):
break
freq[i] = 1
used[d-1] = 1
parent = data[d-1,var_parents]
d += 1
if(d > n):
break
#count frequencies of states while keeping track of used samples
for j in range(d-1,n):
if(used[j] == 0):
if((parent==data[j,var_parents]).all()):
i = 0
while range_var[i] != data[j,var]:
i += 1
freq[i] += 1
used[j] = 1
sum_m = np.sum(freq)
r_i = int(r_i)
#Finally, sum over frequencies to get log likelihood bayesian score
#with uniform priors
for j in range(1,r_i+1):
if(freq[j-1] != 0):
score += ln_gamma(freq[j-1]+1)
```

```
score += ln_gamma(r_i) - ln_gamma(sum_m + r_i)
return score

#Data structure to hold samples and dimension state info.
class data_blob:
def __init__(self, _data):
self.var_number = np.size(_data[0,:])
self.n_samples = np.size(_data[:,0])
self.data = _data
(self.var_range, self.var_range_length) = get_dim_range(_data,np.arange(0,self.var


#k2 uses scoring function to iteratively find best dag given a topological orderin
def k2(blob,order,constraint_u):
dim = blob.var_number
dag = np.zeros((dim,dim),dtype='int64')
k2_score = np.zeros((1,dim),dtype='float')
for i in range(1,dim):
parent = np.zeros((dim,1))
ok = 1
p_old = -1e10
while(ok == 1 and np.sum(parent) <= constraint_u):
local_max = -10e10
local_node = 0
#iterate through possible parent connections to determine best action
for j in range(i-1,-1,-1):
if(parent[order[j]] == 0):
parent[order[j]] = 1
#score this node
local_score = score(blob,order[i],find(parent[:,0],1))
#determine local max
if(local_score > local_max):
local_max = local_score
local_node = order[j]
#mark parent processed
parent[order[j]] = 0
#assign the highest parent
p_new = local_max
```

```
if(p_new > p_old):
p_old = p_new
parent[local_node] = 1
else:
ok = 0
k2_score[0,order[i]] = p_old
dag[:,order[i]] = parent.reshape(blob.var_number)
return dag, k2_score

data_set = 'data/titanic.csv'
categories = np.genfromtxt(data_set, delimiter=',', max_rows=1, dtype=str)
data = genfromtxt(data_set, dtype='int64', delimiter=',',skip_header=True)


#initialize "the blob" and map its variable names to indicies
g = data_blob(data)

mapping = map_categories(categories)
#set the maximum number of parents any node can have
iters = 1
p_lim_max = 5
#iterate from p_lim_floor to p_lim_max with random restart
p_lim_floor = 4
best_score = -10e10
best_dag = np.zeros((1,1))
time.clock()
for i in range(iters):
for u in range(p_lim_floor,p_lim_max):
#generate random ordering
order = np.arange(g.var_number)
(dag,k2_score) = k2(g,order,u)
score = np.sum(k2_score)
if(score > best_score):
best_score = score
best_dag = dag

filename = 'graph_out/titanic.gph'
graph_out(dag,filename,mapping)
```

```
print(score)
print(dag)
print(time.clock())
```