A Step-by-Step Guide in detecting causal relationships using Bayesian Structure Learning in Python

The starter's guide to effectively determine causalities across variables.

Photo by GR Stocks on Unsplash

Determining causality across variables can be a challenging step but it is important for strategic actions. I will summarize the concepts of ***causal models*** in terms of ***Bayesian probabilistic***, followed by a hands-on tutorial to detect *causal* relationships using *Bayesian structure learning*. I will use the *sprinkler dataset* to conceptually explain how structures are learned with the use of the Python library *bnlearn*.

Background

The use of machine learning techniques has become a standard toolkit to obtain useful insights and make predictions in many areas such as disease prediction, recommendation systems, natural language processing. Although good performances can be achieved, it is not straightforward to extract causal relationships with, for example, the target variable. In other words: *which variables have a direct causal effect on the target variable?* Such insights are important to determine *the driving factors* that reach the conclusion, and as such, strategic actions can be taken. A branch of machine learning is Bayesian probabilistic graphical models, also named Bayesian networks (BN), which can be used to determine such causal factors.

Let's rehash some terminology before we jump into the technical details of causal models. It is common to use the terms "*correlation*" and "*association*" interchangeably. But we all know that correlation or association is not causation. Or in other words, observed relationships between two variables do not necessarily mean that one causes the other. Technically, correlation refers to a linear relationship between two variables whereas association refers to any relationship between two (or more) variables. Causation, on the other hand, means that one variable (often called the predictor variable or independent variable) causes the other (often called the outcome variable or dependent variable) [1]. In the next two sections, I will briefly describe *correlation* and *association* by example.

Correlation

Pearson correlation is the most commonly used correlation coefficient. It is so common that it is often used synonymously with correlation. The strength is denoted by *r* and measures the strength of a linear relationship in a sample on a standardized scale from -1 to 1. There are three possible results when using correlation:

- Positive correlation: a relationship between two variables in which both variables move in the same direction
- Negative correlation: a relationship between two variables in which an increase in one variable is associated with a decrease in the other, and
- No correlation: when there is no relationship between two variables.

An example of positive correlation is demonstrated in *Figure 1* where the relationship is seen between chocolate consumption and the number of Nobel Laureates per country [2].

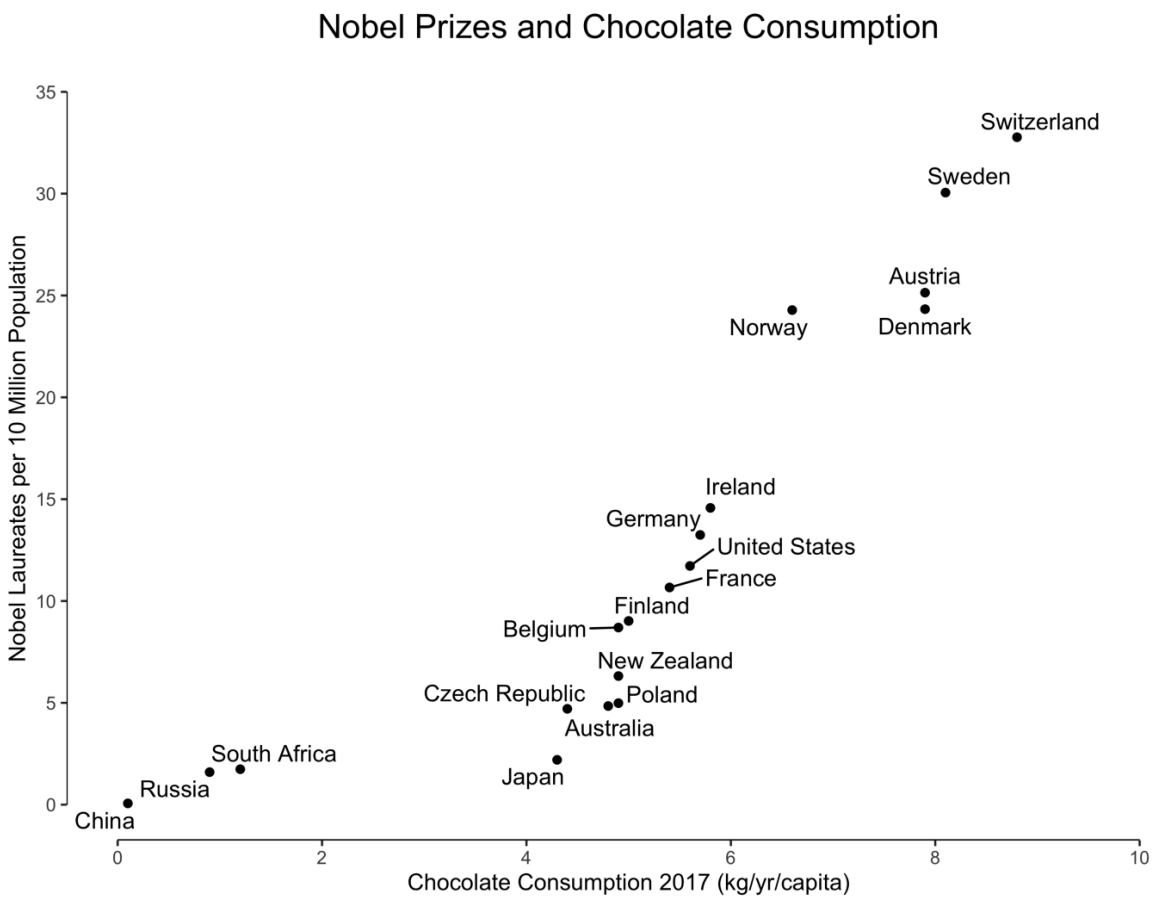


Figure 1: correlation between Chocolate consumption vs. Nobel Laureates

The figure shows that chocolate consumption could imply an increase in Nobel Laureates. Or the other way around, an increase in Nobel laureates could likewise underlie an increase in chocolate consumption. Despite the strong correlation, it is more plausible that unobserved variables such as socioeconomic status or quality of the education system might cause an increase in both chocolate consumption and Nobel Laureates. Or in other words, it is still unknown whether the relationship is causal [2]. This does not mean that correlation by itself is useless, it simply has a different purpose [3]. C***orrelation by itself does not imply causation because statistical relations do not uniquely constrain causal relations.***

Association.

When we talk about *association*, we mean that certain values of one variable tend to co-occur with certain values of the other variable. From a statistical point of view, there are many measures of association (such as chi-square test, Fisher exact test, hypergeometric test, etc) and are often used where one or both of the variables is either ordinal or nominal. It should be noted that *correlation* is a technical term, whereas the term *association* is not, and therefore, there is not always consensus about the meaning in statistics. This means that it's always a good practice to state the meaning of the terms you're using. More information about associations can be found in this blog [4] and read this blog on how to explore and understand your data with a network of significant associations [5].

For the sake of example, I will use the *Hypergeometric test* to demonstrate whether two variables are *associated* using the Titanic dataset. The Titanic dataset is used in many machine learning examples, and it is readily known that the *sex* status (female) is a good predictor for *survival*. Let me demonstrate how to compute the *association* between *survived* and *female*.

First, install the *bnlearn* library and only load the Titanic dataset.

```
pip install bnlearn
```

Q: What is the probability that females survived?

```
import bnlearn
```

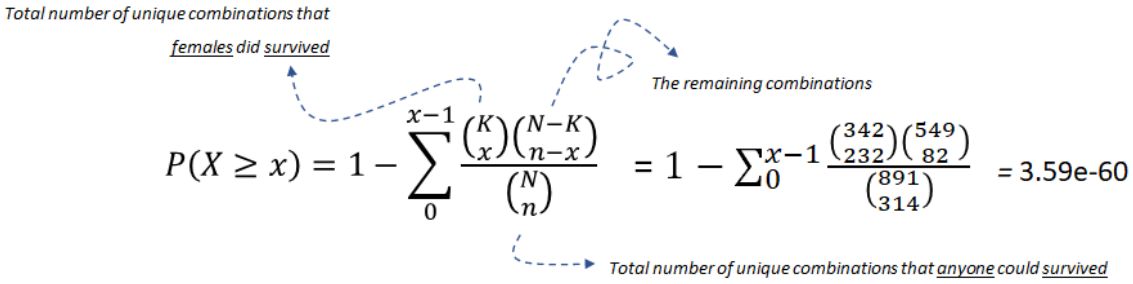
```
import pandas as pd
from scipy.stats import hypergeom

# Load titanic dataset
df = bnlearn.import_example(data='titanic')

print(df[['Survived','Sex']])
#     Survived     Sex
#0           0    male
#1           1  female
#2           1  female
#3           1  female
#4           0    male
#..        ...     ...
#886         0    male
#887         1  female
#888         0  female
#889         1    male
#890         0    male
#[891 rows x 2 columns]

# Total number of samples
N=df.shape[0]
# Number of success in the population
K=sum(df['Survived']==1)
# Sample size/number of draws
n=sum(df['Sex']=='female')
# Overlap between female and survived
x=sum((df['Sex']=='female') & (df['Survived']==1))

print(x-1, N, n, K)
# 232 891 314 342

# Compute
P = hypergeom.sf(x, N, n, K)
P = hypergeom.sf(232, 891, 314, 342)

print(P)
# 3.5925132664684234e-60
```

Hypergeometric test between survived and female

***Null hypothesis**: There is no relation between survived and female.*

The hypergeometric test uses the hypergeometric distribution to measure the statistical significance of a discrete probability distribution. In this example, ***N** is the population size (891), **K** is the number of successful states in the population (342), **n** is the sample*

size/number of draws (314), ***x*** *is the number of success in sample (233).*



Equation 1: Test the association between survived and female using the Hypergeometric test.

We can reject the null hypothesis under *alpha*=0.05 and therefore, we can speak about a statistically significant association *between survived and female.* ***Importantly, association by itself does not imply causation.*** We need to distinguish between ***marginal*** associations and ***conditional*** associations. The latter is the key building block of causal inference.

Causation.

Causation means that one (independent) variable causes the other (dependent) variable and is formulated by *Reichenbach (1956)* as follows:

If two random variables X and Y are statistically dependent (X/Y), then either (a) X causes Y, (b) Y causes X, or (c) there exists a third variable Z that causes both X and Y. Further, X and Y become independent given Z, i.e., X⊥Y/Z.

This definition is incorporated in Bayesian graphical models (*a.k.a. Bayesian networks, Bayesian belief networks, Bayes Net, causal probabilistic networks, and Influence diagrams*). A lot of names for

the same technique. To determine causality, we can use Bayesian networks (BN). Let's start with the graph and visualize the statistical dependencies between the three variables described by *Reichenbach* (X, Y, Z) (see figure 2). Nodes correspond to variables (X, Y, Z) and the directed edges (arrows) indicate dependency relationships or conditional distributions.

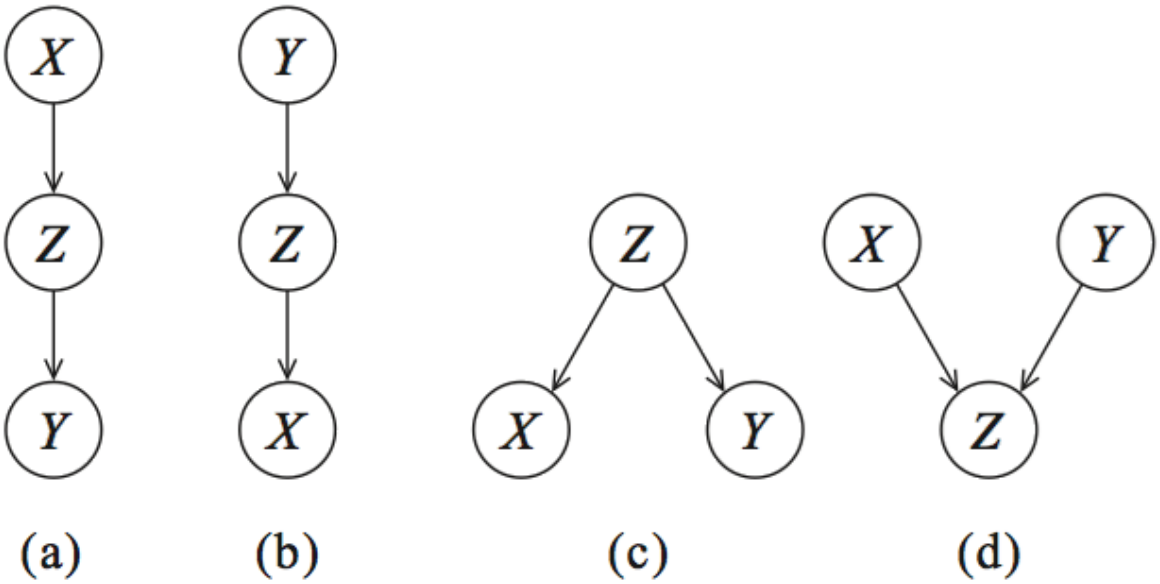


Figure 2: DAGs encode conditional independencies. (a, b, c) are Equivalence classes. (a, b) *Cascade,* (c) *Common parent,* and (d) is a special class with *V-structure.*

Four graphs can be created; *(a, b) Cascade, (c) Common parent and (d)* the *V-structure,* and these graphs form the basis for Bayesian networks.

But how can we tell what causes what?

The conceptual idea to determine the direction of causality, thus which node influences which node, is by holding a node constant and then observe the effect. As an example, let's take DAG (a) in Figure 2, which describes that Z is caused by X, and Y is caused by Z.

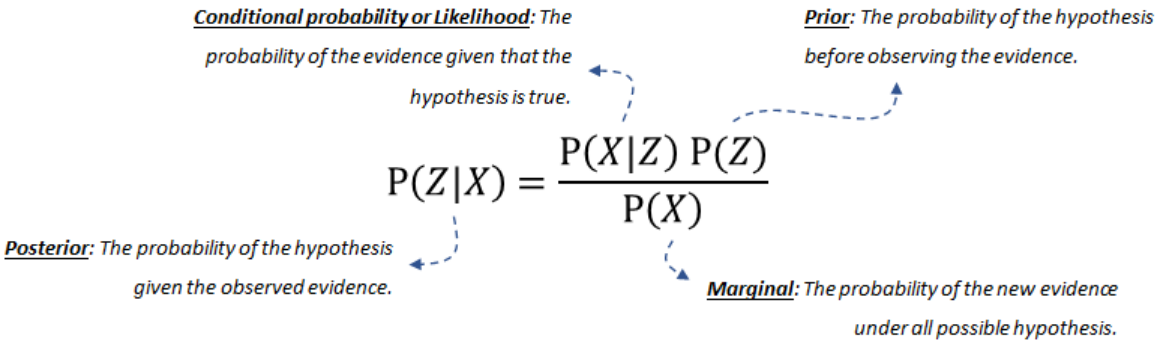
If we now keep Z constant there should not be a change in Y if this model is true. Every Bayesian network can be described by these four graphs, and with ***probability theory*** *(see the section below)* we can glue the parts together.

Bayesian network is a happy marriage between probability and graph theory.

It should be noted that a Bayesian network is a *Directed Acyclic Graph* (DAG) and DAGs are causal. This means that the edges in the *graph* are *directed* and there is no (feedback) loop (*acyclic*).

Probability theory

Probability theory, or more specific *Bayes theorem or Bayes Rule,* forms the fundament for Bayesian networks. The Bayes rule is used to update model information, and stated mathematically as the following equation:



Equation 2: Bayes rule.

The equation consists of four parts; the ***posterior probability*** is the probability that Z occurs given X. The ***conditional probability*** or likelihood is the probability of the evidence given

that the hypothesis is true. This can be derived from the data. Our ***prior*** belief is the probability of the hypothesis before observing the evidence. This can also be derived from the data or domain knowledge. Finally, the ***marginal*** probability describes the probability of the new evidence under all possible hypotheses which needs to be computed. If you want to read more about the (factorized) probability distribution or more details the joint distribution for a Bayesian network, try this blog [6].

Bayesian Structure Learning to estimate the DAG.

With structure learning, we want to determine the structure of the graph that best captures the causal dependencies between the variables in the data set. Or in other words:

What is the DAG that best fits the data?

A naïve manner to find the best DAG is simply creating all possible combinations of the graph, i.e., by making tens, hundreds, or even thousands of different DAGs until all combinations are exhausted. Each DAG can then be scored on the fit of the data. Finally, the best scoring DAG is returned. In the case of variables X, Y, Z, one can make the graphs as shown in Figure 2 and a few more because it is not only X>Z>Y (Figure 2a), but it can also be like Z>X>Y, etc. The variables X, Y, Z can be boolean values (True or False), **but can also have multiple states**. The search space of DAGs becomes so-called super-exponential in the number of variables that maximize the score. This means that an exhaustive search is practically infeasible with a large number of nodes, and therefore, various

greedy strategies have been proposed to browse DAG space. With optimization-based search approaches, it is possible to browse a larger DAG space. Such approaches require a s*coring function* and a s*earch strategy*. A common scoring function is the posterior probability of the structure given the training data, like the BIC or the BDeu.

Structure learning for large DAGs requires a scoring function and search strategy.

Before we jump into the examples, it is always good to understand when to use which technique. There are two broad approaches to search throughout the DAG space and find the best fitting graph for the data.

- **Score-based structure learning**
- **Constraint-based structure learning**

Note that a local search strategy makes incremental changes aimed at improving the score of the structure. A global search algorithm like Markov chain Monte Carlo can avoid getting trapped in local minima but I will not discuss that here.

Score-based structure learning

Score-based approaches have two main components:

1. The *search algorithm* to optimize throughout the search space of all possible DAGs; such as *ExhaustiveSearch*, *Hillclimbsearch*, *Chow-Liu*.

2. The *scoring function* indicates how well the Bayesian network fits the data. Commonly used scoring functions are *Bayesian Dirichlet scores* such as *BDeu* or *K2* and the *Bayesian Information Criterion* (*BIC*, also called MDL).

Four common score-based methods are depicted below, but more detail about the Bayesian scoring methods can be found here [9].

- ***ExhaustiveSearch***, as the name implies, scores every possible DAG and returns the best-scoring DAG. This search approach is only attractable for very small networks and prohibits efficient local optimization algorithms to always find the optimal structure. Thus, identifying the ideal structure is often not tractable. Nevertheless, heuristic search strategies often yield good results if only a few nodes are involved (read: less than 5 or so).

- ***Hillclimbsearch*** is a heuristic search approach that can be used if more nodes are used. *HillClimbSearch* implements a greedy local search that starts from the DAG "start" (default: disconnected DAG) and proceeds by iteratively performing single-edge manipulations that maximally increase the score. The search terminates once a local maximum is found.

- ***Chow-Liu*** algorithm is a specific type of tree-based approach. The *Chow-Liu* algorithm finds the maximum-likelihood tree structure where each node has at most one

parent. The complexity can be limited by restricting to tree structures.

- ***Tree-augmented Naive Bayes (TAN)*** algorithm is also a tree-based approach that can be used to model huge datasets involving lots of uncertainties among its various interdependent feature sets [6].

Constraint-based structure learning

- ***Chi-square test.*** A different, but quite straightforward approach to construct a DAG by identifying independencies in the data set using hypothesis tests, such as chi2 test statistic. This approach does rely on statistical tests and conditional hypotheses to learn independence among the variables in the model. The P-value of the chi2 test is the probability of observing the computed chi2 statistic, given the null hypothesis that X and Y are independent given Z. This can be used to make independent judgments, at a given level of significance. An example of a constraint-based approach is the PC algorithm which starts with a complete fully connected graph and removes edges based on the results of the tests if the nodes are independent until a stopping criterion is achieved.

The bnlearn library

A few words about the *bnlearn* library that is used for all the analysis in this article. The *bnlearn* library is designed to tackle a few challenges such as:

- ***Structure learning***: Given the data: Estimate a DAG that captures the dependencies between the variables.

- ***Parameter learning***: Given the data and DAG: Estimate the (conditional) probability distributions of the individual variables.
- ***Inference***: Given the learned model: Determine the exact probability values for your queries.

What benefits does bnlearn *offer over other bayesian analysis implementations?*

- Build on top of the pgmpy library
- Contains the most-wanted bayesian pipelines
- Simple and intuitive
- Open-source
- Documentation page

Structure learning in the sprinkler dataset.

Let's start with a simple and intuitive example to demonstrate the working of structure learning. Suppose you have a **sprinkler system** in your backyard and for the last 1000 days, you measured four variables, each with two states: *Rain* (yes or no), *Cloudy* (yes or no), *Sprinkler system* (on or off), and *Wet grass* (true or false). Based on these four variables and your conception of the real world, you may have an intuition how the graph should look like. right? right? If not, it is good that you read this article because with structure learning you will find out!

With *bnlearn* it is easy to determine the causal relationships with only a few lines of code.

In the example below, we will import the bnlearn library, load the sprinkler dataset, and determine which DAG fits best the data. Note that the sprinkler dataset is readily cleaned without missing values and all values have the state 1 or 0.

```
import bnlearn as bn
# Load sprinkler dataset
df = bn.import_example('sprinkler')
# Print to screen for illustration
print(df)
'''
+----+----------+-------------+--------+-------------+
|    |   Cloudy |   Sprinkler |   Rain |   Wet_Grass |
+====+==========+=============+========+=============+
|  0 |        0 |           0 |      0 |           0 |
+----+----------+-------------+--------+-------------+
|  1 |        1 |           0 |      1 |           1 |
+----+----------+-------------+--------+-------------+
|  2 |        0 |           1 |      0 |           1 |
+----+----------+-------------+--------+-------------+
| .. |        1 |           1 |      1 |           1 |
+----+----------+-------------+--------+-------------+
|999 |        1 |           1 |      1 |           1 |
+----+----------+-------------+--------+-------------+
'''

# Learn the DAG in data using Bayesian structure learning:
DAG = bn.structure_learning.fit(df)

# print adjacency matrix
print(DAG['adjmat'])
# target     Cloudy  Sprinkler   Rain  Wet_Grass
# source
# Cloudy      False      False   True      False
# Sprinkler    True      False  False       True
# Rain        False      False  False       True
# Wet_Grass   False      False  False      False

# Plot
G = bn.plot(DAG)

# Interactive plotting
G = bn.plot(DAG, interactive=True)
```

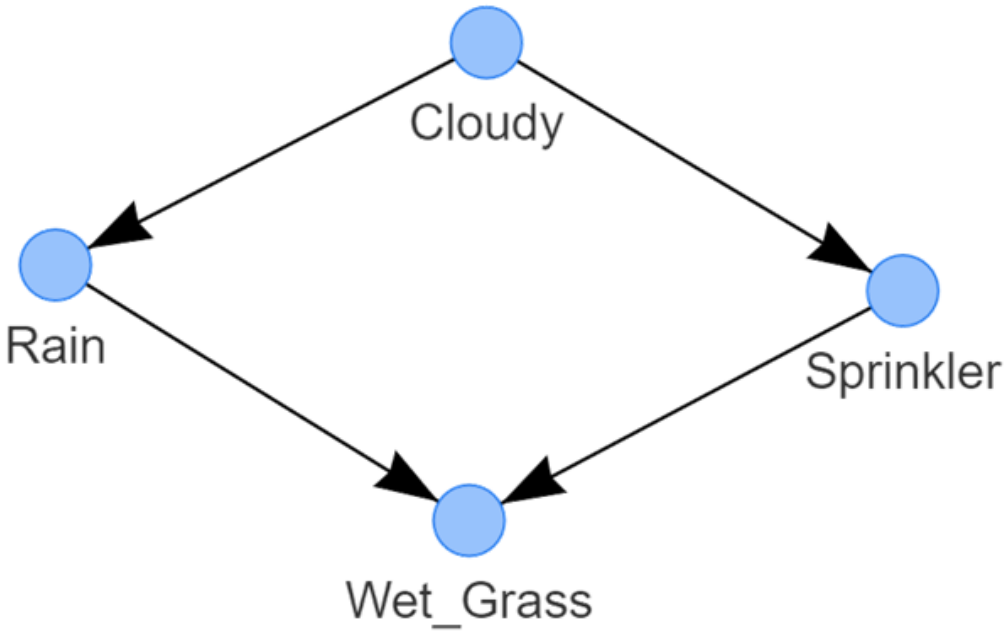


Figure 3: Example of the best DAG for the Sprinkler system. It encodes the following logic: the probability that the grass is wet is dependent on Sprinkler and Rain. The probability that the sprinkler is on is dependent on Cloudy. The probability that it rains is dependent on Cloudy.

That's it! We have the learned structure as shown in Figure 3. The detected DAG consists of four nodes that are connected through edges, each edge indicates a causal relation. The state of *Wet grass* depends on two nodes, *Rain* and *Sprinkler*. The state of *Rain* is conditioned by *Cloudy*, and separately, the state *Sprinkler* is also conditioned by *Cloudy*. This DAG represents the (factorized) probability distribution, where S is the random variable for sprinkler, R for the rain, G for the wet grass, and C for cloudy.

$$P(S,R,G,C) = P(C)\ P(S|C)\ P(R|C)\ P(W|R,S)$$

By examining the graph, you quickly see that the only independent variable in the model is *C*. The other variables are conditioned on the probability of cloudy, rain, and/or the sprinkler. In general, the joint distribution for a Bayesian Network is the product of the conditional probabilities for every node given its parents:

$$P(X) = \prod_{i=1}^{n} P(X_i \mid Parents\, X_i)$$

The default setting in *bnlearn* for structure learning is the **hillclimbsearch** method and **BIC** scoring. Notably, different methods and scoring types can be specified. See the example to specify the search and scoring type:

```
# 'hc' or 'hillclimbsearch'
model_hc_bic  = bn.structure_learning.fit(df, methodtype='hc',
scoretype='bic')
model_hc_k2   = bn.structure_learning.fit(df, methodtype='hc',
scoretype='k2')
model_hc_bdeu = bn.structure_learning.fit(df, methodtype='hc',
scoretype='bdeu')

# 'ex' or 'exhaustivesearch'
model_ex_bic  = bn.structure_learning.fit(df, methodtype='ex',
scoretype='bic')
model_ex_k2   = bn.structure_learning.fit(df, methodtype='ex',
scoretype='k2')
model_ex_bdeu = bn.structure_learning.fit(df, methodtype='ex',
scoretype='bdeu')

# 'cs' or 'constraintsearch'
model_cs_k2   = bn.structure_learning.fit(df, methodtype='cs',
scoretype='k2')
model_cs_bdeu = bn.structure_learning.fit(df, methodtype='cs',
scoretype='bdeu')
model_cs_bic  = bn.structure_learning.fit(df, methodtype='cs',
scoretype='bic')

# 'cl' or 'chow-liu' (requires setting root_node parameter)
```

```
model_cl      = bn.structure_learning.fit(df, methodtype='cl',
root_node='Wet_Grass')
```

Example of the various structure learning methods and scoring types in bnlearn.

Although the detected DAG for the sprinkler dataset is insightful and shows the causal dependencies for the variables in the dataset, it does not allow you to ask all kinds of questions, such as:

- *How probable is it to have wet grass given the sprinkler is off?*
- *How probable is it to have a rainy day given the sprinkler is off and it is cloudy*?

In the sprinkler dataset, it may be evident what the outcome is, given your knowledge about the world and by logical thinking. But once you have larger, more complex graphs it may not be so evident anymore. With so-called ***inferences,*** we can answer "*what-if-we-did-x*" type questions that would normally require controlled experiments and explicit interventions to answer.

How to make inferences?

To make inferences we need two ingredients;
the ***DAG*** and ***Conditional Probabilistic Tables (CPTs)***. At this point, we have the data stored in the data frame (*df*) and we readily computed the *DAG* that describes the structure of the data. The CPTs are needed to quantitatively describe the statistical relationship between each node and its parents. The CPTs can be computed using ***Parameter learning***, so let's jump into

parameter learning first, and then we move back to making inferences.

Parameter learning

Parameter learning is the task to estimate the values of the *Conditional Probability Tables (CPTs)*. The *bnlearn* library supports Parameter learning for discrete nodes:

- ***Maximum Likelihood Estimation*** is a natural estimate by using the relative frequencies with which the variable states have occurred. When estimating parameters for Bayesian networks, lack of data is a frequent problem and the ML estimator has the problem of overfitting to the data. In other words, if the observed data is not representative (or too small) for the underlying distribution, ML estimations can be extremely far off. As an example, if a variable has 3 parents that can each take 10 states, then state counts will be done separately for 10^3 = 1000 parents configurations. This can make MLE very fragile for learning Bayesian Network parameters. A way to mitigate MLE's overfitting is Bayesian Parameter Estimation.
- ***Bayesian Estimation*** starts with readily existing prior CPTs, that express our beliefs about the variables *before* the data was observed. Those "priors" are then updated using the state counts from the observed data. One can think of the priors as consisting in *pseudo-state counts*, that are added to the actual counts before normalization. A very simple prior is the so-called *K2* prior, which simply adds "1" to the count of every single state. A somewhat more

sensible choice of prior is *BDeu* (Bayesian Dirichlet equivalent uniform prior).

Parameter learning on the sprinkler dataset.

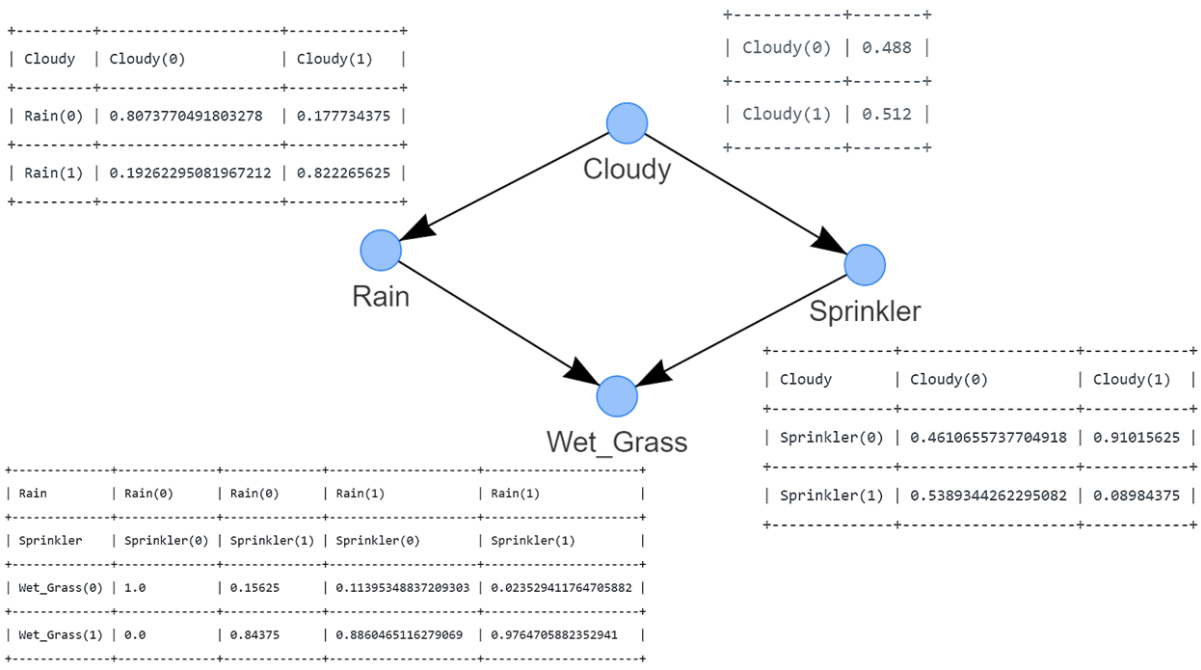
I will continue with the sprinkler dataset to learn its parameters, resulting in the detection of *Conditional Probabilistic Tables (CPTs)*. To learn parameters, we need a *Directed Acyclic Graph (DAG)* and a dataset with exactly the same variables. The idea is to connect the dataset with the DAG. In the previous example, we readily computed the DAG (Figure 3). You can use it in this example or alternatively, you can create your own DAG based on your knowledge of the world! In the example, I will demonstrate how to create your own DAG which can be based on expert/domain knowledge.

```
import bnlearn as bn
# Load sprinkler dataset
df = bn.import_example('sprinkler')
# The edges can be created using the available variables.
print(df.columns)
# ['Cloudy', 'Sprinkler', 'Rain', 'Wet_Grass']

# Define the causal dependencies based on your expert/domain knowledge.
# Left is the source, and right is the target node.
edges = [('Cloudy', 'Sprinkler'),
         ('Cloudy', 'Rain'),
         ('Sprinkler', 'Wet_Grass'),
         ('Rain', 'Wet_Grass')]


# Create the DAG
DAG = bn.make_DAG(edges)

# Plot the DAG. This is identical as shown in Figure 3
bn.plot(DAG)

# Print the Conditional probability Tables
bn.print_CPD(DAG)
# [bnlearn] >No CPDs to print. Tip: use bnlearn.plot(DAG) to make a plot.
# This is correct, we did not learn any CPTs yet! We only defined the graph
without defining any probabilities.
```

```
# Parameter learning on the user-defined DAG and input data using
maximumlikelihood
model_mle = bn.parameter_learning.fit(DAG, df,
methodtype='maximumlikelihood')

# Print the learned CPDs
bn.print_CPD(model_mle)

# CPD of Cloudy:
# +-----------+-------+
# | Cloudy(0) | 0.488 |
# +-----------+-------+
# | Cloudy(1) | 0.512 |
# +-----------+-------+
# CPD of Rain:
# +---------+---------------------+-------------+
# | Cloudy  | Cloudy(0)           | Cloudy(1)   |
# +---------+---------------------+-------------+
# | Rain(0) | 0.8073770491803278  | 0.177734375 |
# +---------+---------------------+-------------+
# | Rain(1) | 0.19262295081967212 | 0.822265625 |
# +---------+---------------------+-------------+
# CPD of Sprinkler:
# +--------------+--------------------+------------+
# | Cloudy       | Cloudy(0)          | Cloudy(1)  |
# +--------------+--------------------+------------+
# | Sprinkler(0) | 0.4610655737704918 | 0.91015625 |
# +--------------+--------------------+------------+
# | Sprinkler(1) | 0.5389344262295082 | 0.08984375 |
# +--------------+--------------------+------------+
# CPD of Wet_Grass:
# +--------------+-------------+-------------+---------------------+-----
-----------------+
# | Rain         | Rain(0)     | Rain(0)     | Rain(1)             |
Rain(1)             |
# +--------------+-------------+-------------+---------------------+-----
-----------------+
# | Sprinkler    | Sprinkler(0) | Sprinkler(1) | Sprinkler(0)        |
Sprinkler(1)        |
# +--------------+-------------+-------------+---------------------+-----
-----------------+
# | Wet_Grass(0) | 1.0         | 0.15625     | 0.11395348837209303 |
0.023529411764705882 |
# +--------------+-------------+-------------+---------------------+-----
-----------------+
# | Wet_Grass(1) | 0.0         | 0.84375     | 0.8860465116279069  |
0.9764705882352941   |
# +--------------+-------------+-------------+---------------------+-----
-----------------+
# [bnlearn] >Independencies:
# (Cloudy _|_ Wet_Grass | Sprinkler, Rain)
# (Sprinkler _|_ Rain | Cloudy)
# (Rain _|_ Sprinkler | Cloudy)
# (Wet_Grass _|_ Cloudy | Sprinkler, Rain)
# [bnlearn] >Nodes: ['Cloudy', 'Sprinkler', 'Rain', 'Wet_Grass']
# [bnlearn] >Edges: [('Cloudy', 'Sprinkler'), ('Cloudy', 'Rain'),
('Sprinkler', 'Wet_Grass'), ('Rain', 'Wet_Grass')]
```

Example for parameter learning using the sprinkler dataset.

If you reached this point, you have computed the *CPTs* based on the *DAG* and the input dataset *df* using *Maximum Likelihood Estimation (MLE)* (Figure 4). Note that the CPTs are included in Figure 4 for clarity purposes.



Cloudy	Cloudy(0)	Cloudy(1)
Rain(0)	0.8073770491803278	0.177734375
Rain(1)	0.19262295081967212	0.822265625

Cloudy(0)	0.488
Cloudy(1)	0.512

Cloudy	Cloudy(0)	Cloudy(1)
Sprinkler(0)	0.4610655737704918	0.91015625
Sprinkler(1)	0.5389344262295082	0.08984375

Rain	Rain(0)	Rain(0)	Rain(1)	Rain(1)
Sprinkler	Sprinkler(0)	Sprinkler(1)	Sprinkler(0)	Sprinkler(1)
Wet_Grass(0)	1.0	0.15625	0.11395348837209303	0.023529411764705882
Wet_Grass(1)	0.0	0.84375	0.8860465116279069	0.9764705882352941

Figure 4: CPTs are derived with Parameter learning using *Maximum Likelihood Estimation.*

Computing the CPTs using MLE is straightforward, let me demonstrate this by example by computing the CPTs manually for the nodes *Cloudy* and *Rain*.

```
# Examples to illustrate how to manually compute MLE for the node Cloudy
and Rain:

# Compute CPT for the Cloudy Node:
# This node has no conditional dependencies and can easily be computed as
following:

# P(Cloudy=0)
sum(df['Cloudy']==0) / df.shape[0] # 0.488

# P(Cloudy=1)
sum(df['Cloudy']==1) / df.shape[0] # 0.512

# Compute CPT for the Rain Node:
# This node has a conditional dependency from Cloudy and can be computed as
following:
```

```
# P(Rain=0 | Cloudy=0)
sum( (df['Cloudy']==0) & (df['Rain']==0) ) / sum(df['Cloudy']==0) # 394/488
= 0.807377049

# P(Rain=1 | Cloudy=0)
sum( (df['Cloudy']==0) & (df['Rain']==1) ) / sum(df['Cloudy']==0) # 94/488
= 0.192622950

# P(Rain=0 | Cloudy=1)
sum( (df['Cloudy']==1) & (df['Rain']==0) ) / sum(df['Cloudy']==1) # 91/512
= 0.177734375

# P(Rain=1 | Cloudy=1)
sum( (df['Cloudy']==1) & (df['Rain']==1) ) / sum(df['Cloudy']==1) # 421/512
= 0.822265625
```

Example to compute MLE manually for the node Cloudy and Rain.

Note that conditional dependencies can be based on limited data points. Aa an example, *P(Rain=1|Cloudy=0)* is based on 91 observations. If *Rain* had more than two states and/or more dependencies, this number would have been even lower. ***Is more data the solution?*** Maybe. Maybe not. Just keep in mind that even if the total sample size is very large, the fact that state counts are conditionally for each parent's configuration can also cause fragmentation. Check out the differences with the CPTs compared to the MLE approach.

```
# Parameter learning on the user-defined DAG and input data using Bayes
model_bayes = bn.parameter_learning.fit(DAG, df, methodtype='bayes')

# Print the learned CPDs
bn.print_CPD(model_bayes)

# CPD of Cloudy:
# +-----------+-------+
# | Cloudy(0) | 0.494 |
# +-----------+-------+
# | Cloudy(1) | 0.506 |
# +-----------+-------+
# CPD of Rain:
# +---------+--------------------+---------------------+
# | Cloudy  | Cloudy(0)          | Cloudy(1)           |
# +---------+--------------------+---------------------+
# | Rain(0) | 0.6518218623481782 | 0.33695652173913043 |
# +---------+--------------------+---------------------+
```

```
# | Rain(1) | 0.3481781376518219 | 0.6630434782608695  |
# +---------+--------------------+---------------------+
# CPD of Sprinkler:
# +--------------+--------------------+--------------------+
# | Cloudy       | Cloudy(0)          | Cloudy(1)          |
# +--------------+--------------------+--------------------+
# | Sprinkler(0) | 0.4807692307692308 | 0.7075098814229249 |
# +--------------+--------------------+--------------------+
# | Sprinkler(1) | 0.5192307692307693 | 0.2924901185770751 |
# +--------------+--------------------+--------------------+
# CPD of Wet_Grass:
# +--------------+--------------------+---------------------+--------------
-------+---------------------+
# | Rain         | Rain(0)            | Rain(0)             | Rain(1)
| Rain(1)             |
# +--------------+--------------------+---------------------+--------------
-------+---------------------+
# | Sprinkler    | Sprinkler(0)       | Sprinkler(1)        | Sprinkler(0)
| Sprinkler(1)        |
# +--------------+--------------------+---------------------+--------------
-------+---------------------+
# | Wet_Grass(0) | 0.7553816046966731 | 0.33755274261603374 |
0.25588235294117645 | 0.37910447761194027 |
# +--------------+--------------------+---------------------+--------------
-------+---------------------+
# | Wet_Grass(1) | 0.2446183953033268 | 0.6624472573839663  |
0.7441176470588236  | 0.6208955223880597  |
# +--------------+--------------------+---------------------+--------------
-------+---------------------+
# [bnlearn] >Independencies:
# (Cloudy _|_ Wet_Grass | Sprinkler, Rain)
# (Sprinkler _|_ Rain | Cloudy)
# (Rain _|_ Sprinkler | Cloudy)
# (Wet_Grass _|_ Cloudy | Sprinkler, Rain)
# [bnlearn] >Nodes: ['Cloudy', 'Sprinkler', 'Rain', 'Wet_Grass']
# [bnlearn] >Edges: [('Cloudy', 'Sprinkler'), ('Cloudy', 'Rain'),
('Sprinkler', 'Wet_Grass'), ('Rain', 'Wet_Grass')]
```

Inferences on the sprinkler dataset.

To make inferences, it requires the Bayesian network to have two main components: A *Directed Acyclic Graph (DAG)* that describes the structure of the data and *Conditional Probability Tables (CPT)* that describe the statistical relationship between each node and its parents. At this point you have the dataset, you computed the

DAG using structure learning and estimated the CPTs using parameter learning. You can now make inferences!

With inferences, we marginalize variables in a procedure that is called variable elimination. Variable elimination is an exact inference algorithm. It can also be used to figure out the state of the network that has maximum probability by simply exchanging the sums by max functions. Its downside is that for large BNs it might be computationally intractable. Approximate inference algorithms such as Gibbs sampling or rejection sampling might be used in these cases [7].

With *bnlearn* we can make inferences as follow:

```
import bnlearn as bn

# Load sprinkler dataset
df = bn.import_example('sprinkler')

# Define the causal dependencies based on your expert/domain knowledge.
# Left is the source, and right is the target node.
edges = [('Cloudy', 'Sprinkler'),
         ('Cloudy', 'Rain'),
         ('Sprinkler', 'Wet_Grass'),
         ('Rain', 'Wet_Grass')]


# Create the DAG
DAG = bn.make_DAG(edges)

# Parameter learning on the user-defined DAG and input data using Bayes to
estimate the CPTs
model = bn.parameter_learning.fit(DAG, df, methodtype='bayes')
bn.print_CPD(model)

q1 = bn.inference.fit(model, variables=['Wet_Grass'],
evidence={'Sprinkler':0})
print(q1.df)
# +--------------+------------------+
# | Wet_Grass    |   phi(Wet_Grass) |
# +==============+==================+
# | Wet_Grass(0) |           0.4869 |
# +--------------+------------------+
```

```
# | Wet_Grass(1) |           0.5131 |
# +--------------+------------------+


q2 = bn.inference.fit(model, variables=['Rain'], evidence={'Sprinkler':0,
'Cloudy':1})
print(q2.df)
# +---------+-------------+
# | Rain    |   phi(Rain) |
# +=========+=============+
# | Rain(0) |      0.3370 |
# +---------+-------------+
# | Rain(1) |      0.6630 |
# +---------+-------------+

# Inferences with two or more variables can also be made such as:
q3 = bn.inference.fit(model, variables=['Wet_Grass','Rain'],
evidence={'Sprinkler':1})
print(q3.df)
# +--------------+---------+-----------------------+
# | Wet_Grass    | Rain    |   phi(Wet_Grass,Rain) |
# +==============+=========+=======================+
# | Wet_Grass(0) | Rain(0) |                0.1811 |
# +--------------+---------+-----------------------+
# | Wet_Grass(0) | Rain(1) |                0.1757 |
# +--------------+---------+-----------------------+
# | Wet_Grass(1) | Rain(0) |                0.3555 |
# +--------------+---------+-----------------------+
# | Wet_Grass(1) | Rain(1) |                0.2877 |
# +--------------+---------+-----------------------+
```

And now we have the answers to our questions:

How probable is it to have wet grass given the sprinkler is off?
P(Wet_grass=1 | Sprinkler=0) = 0.51
How probable is it have a rainy day given sprinkler is off and it is cloudy?
P(Rain=1 | Sprinkler=0, Cloudy=1) = 0.663

How do I know my causal model is right?

If you solely used data to compute the causal diagram, it is hard to fully verify the validity and completeness of your causal diagram. However, some solutions can help to get more trust in the causal diagram. For example, it may be possible to empirically test certain conditional independence or dependence relationships between sets

of variables. If they are not in the data, it is an indication of the correctness of the causal model [8]. Alternatively, prior expert knowledge can be added, such as a DAG or CPTs, to get more trust in the model when making inferences.

Discussion

In this article, I touched on a few concepts about why correlation or association is not causation and how to go from data towards a causal model using structure learning. A summary of the advantages of Bayesian techniques is that:

1. The outcome of posterior probability distributions, or the graph allows the user to make a judgment on the model predictions instead of having one single value as an outcome.
2. The possibility to incorporate domain/expert knowledge in the DAG and reason with incomplete information and missing data. This is possible because Bayes theorem is built on updating the prior term with evidence.
3. It has a notion of modularity.
4. A complex system is built by combining simpler parts.
5. Graph theory provides intuitively highly interacting sets of variables.
6. Probability theory provides the glue to combine the parts.

A weakness on the other hand of Bayesian networks is that finding the optimum DAG is computationally expensive since an exhaustive

search over all the possible structures must be performed. The limit of nodes for exhaustive search can already be around 15 nodes but also depends on the number of states. If you have more nodes, alternative methods with a scoring function and search algorithm are required. Nevertheless, to deal with problems with hundreds or maybe even thousands of variables, a different approach, such as tree-based or constraint-based approaches is necessary with the use of black/whitelisting of variables. Such an approach first determines the order and then finds the optimal BN structure for that ordering. This implies working on the search space of the possible orderings, which is convenient as it is smaller than the space of network structures.

Determining causality can be a challenging task but the *bnlearn* library is designed to tackle some of the challenges, such as ***Structure learning, Parameter learning,*** and ***Inferences***. But it can also derive the topological ordering of the (entire) graph, or compare two graphs. Documentation can be found here that also contains the examples of the *Alarm, Andes, Asia, Pathfinder, Sachs* models.