bnlearn

Learning Bayesian Networks 10 Years Later

Marco Scutari

scutari@stats.ox.ac.uk
Department of Statistics
University of Oxford

September 19, 2017

bnlearn, an R package for Bayesian networks

bnlearn aspires to provide a free-software implementation of the scientific literature on Bayesian networks (BNs) for

- learning the structure of the network;
- for a given structure, learning the parameters;
- perform inference, mainly in the form of conditional probability queries.

It also tries to

- provide import and export functions to integrate other software and R packages; and
- use R plotting facilities to create publication-quality plots.

It All Began Here

projects / **bnlearn** / **commit**

git

summary | shortlog | log | commit | commitdiff | tree (initial) | patch

commit ▼ ? search:

Initial commit (v 0.1).

```
author     Marco Scutari <fizban@pluto.it>
           Tue, 12 Jun 2007 18:53:43 +0000 (20:53 +0200)
committer  Marco Scutari <fizban@pluto.it>
           Tue, 12 Jun 2007 18:53:43 +0000 (20:53 +0200)
commit     b8c24c841b6941fc631031ba061fbd3b0ac71de6
tree       48ad0bfcc78e0123df87bdc82b74d195ce46877b        tree | snapshot
```

```
Initial commit (v 0.1).
```

```
DESCRIPTION             [new file with mode: 0644]  blob
NAMESPACE               [new file with mode: 0644]  blob
R/cibn.R                [new file with mode: 0644]  blob
R/test.R                [new file with mode: 0644]  blob
R/utils.R               [new file with mode: 0644]  blob
man/bnlearn-package.Rd  [new file with mode: 0644]  blob
man/gs.Rd               [new file with mode: 0644]  blob
```

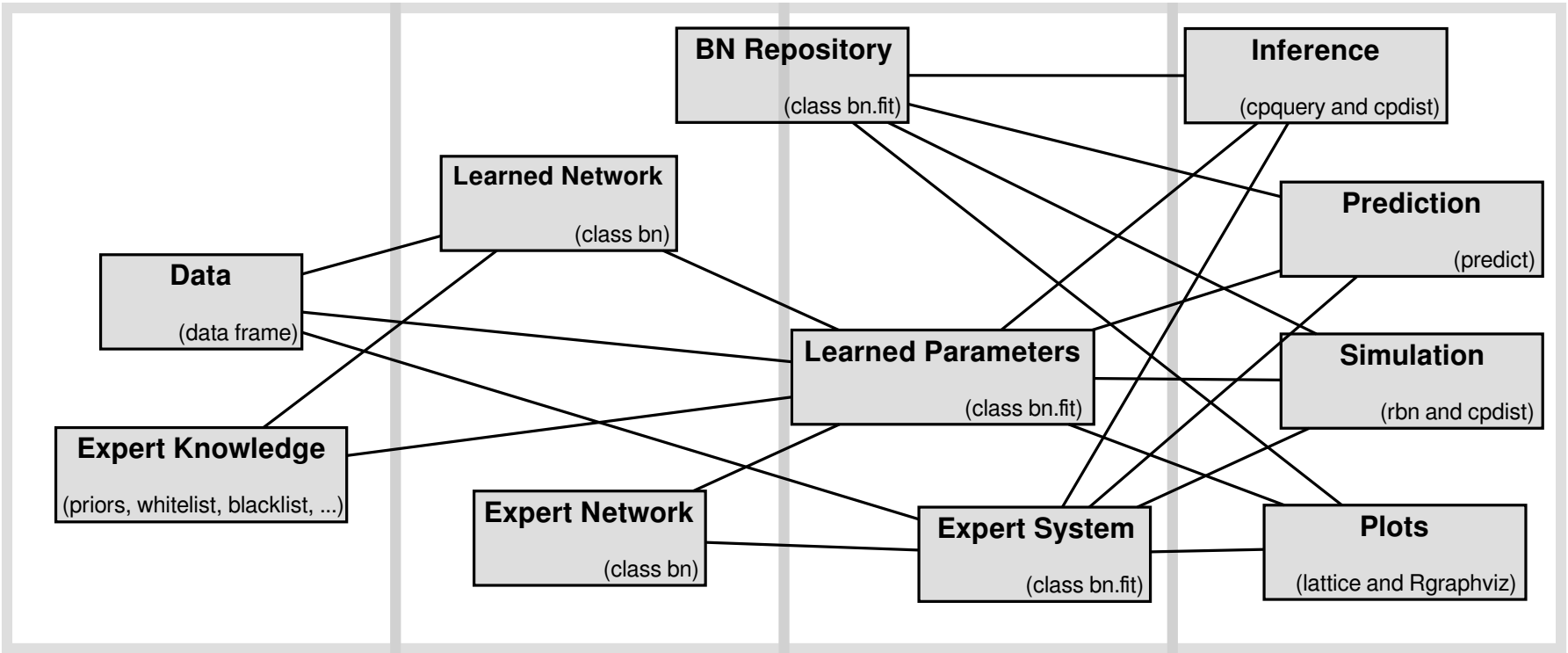
bnlearn R package

Atom RSS

Today: 17K lines of R code, 18K lines of C, and 5K lines of unit tests R code.

The Scope and Philosophy of **bnlearn**

bnlearn is designed to provide a flexible simulation suite for methodological research and effective and scalable data analysis tools for working with real-world data.



Separation of Concerns and Modularity

This is achieved by a modular architecture in which algorithms are decoupled from model assumptions, to make it possible to mix and match the methods found in the literature. For instance, for discrete data

```
dag = hc(learning.test, score = "bic")
```

but we can use the same structure learning algorithm with a different score if the data are continuous

```
dag = hc(gaussian.test, score = "bic-g")
```

or we can use the same score with a different algorithm.

```
dag = tabu(gaussian.test, score = "bic-g")
```

Finally, **bnlearn** tries to guess sensible defaults for the arguments from the data, so command lines can be rather compact.

Two Case Studies: Statistical Genetics and Environmental Statistics

Case Study: Statistical Genetics

DNA is routinely used in statistical genetics to understand human diseases, and to breed traits of commercial interest in plants and animals. One example is disease resistance in wheat, which I studied using data with 721 varieties, 16K genes, 7 traits. (I ran the same analysis on rice with similar results.)

Traits of interest for plants typically include flowering time, height, yield, and disease scores. The goal of the analysis is to find key genes controlling the traits; to identify any causal relationships between them; and to keep a good predictive accuracy.

Genetics Society of America

Multiple Quantitative Trait Analysis Using Bayesian Networks
M. Scutari *et al.*, Genetics, **198**, 129–137 (2014);
DOI: 10.1534/genetics.114.165704

In the spirit of classic statistical genetics models, I used a Gaussian BN.

Bayesian Networks in Genetics

If we have a set of traits and genes for each variety, all we need are the Markov blankets of the traits; most genes are discarded in the process. Using common sense, we can make some assumptions:

- traits can depend on genes, but not vice versa;
- dependencies between traits should follow the order of the respective measurements (e.g. longitudinal traits, traits measured before and after harvest, etc.).

Assumptions on the direction of the dependencies reduce Markov blanket learning to learning the parents and the children of each trait, which is a much simpler task.

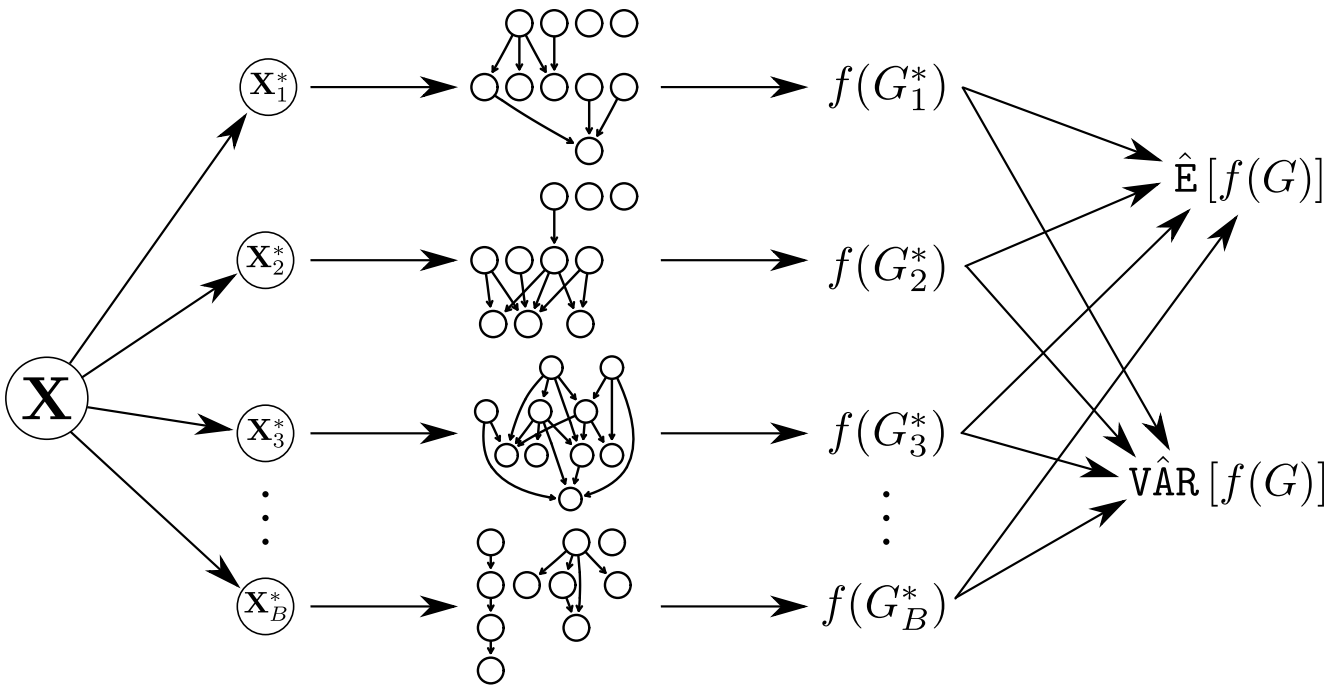
bnlearn provides tools for all these tasks: `learn.mb()`, `learn.nbr()`, `hc()`, whitelists and blacklists.

Learning The Structure

```
fit.the.model = function(data, traits, genes, alpha) {
  qtls = vector(length(traits), mode = "list")
  names(qtls) = traits
  # find the parents and children of each trait.
  for (q in seq_along(qtls)) {
    # BLUP away the family structure.
    m = lmer(as.formula(paste(traits[q], "~ (1|FUNNEL:PLANT)")), data = data)
    data[!is.na(data[, traits[q]]), traits[q]] = data[, traits[q]] -
      ranef(m)[[1]][paste(data$FUNNEL, data$PLANT, sep = ":"), 1]
    # identify parents and children.
    qtls[[q]] = learn.nbr(data[, c(traits, genes)], node = traits[q],
                  method = "si.hiton.pc", test = "cor", alpha = alpha)
  }#FOR
  # yield has no children, and genes cannot depend on traits.
  nodes = unique(c(traits, unlist(qtls)))
  blacklist = tiers2blacklist(list(nodes[nodes %in% genes],
                c("FT", "HT"),
                traits[!(traits %in% c("YLD", "FT", "HT"))], "YLD"))
  # build the overall network.
  hc(data[, nodes], blacklist = blacklist)
}#FIT.THE.MODEL
```

Model Averaging and Assessing Predictive Accuracy

With cross-validation we can assess predictive accuracy and produce an averaged, de-noised consensus network with model averaging.



bnlearn implements both with `bn.cv()` and `averaged.network()`, but makes it easy to code custom implementations for complex analyses.

Performing Cross-Validation (Single Fold)

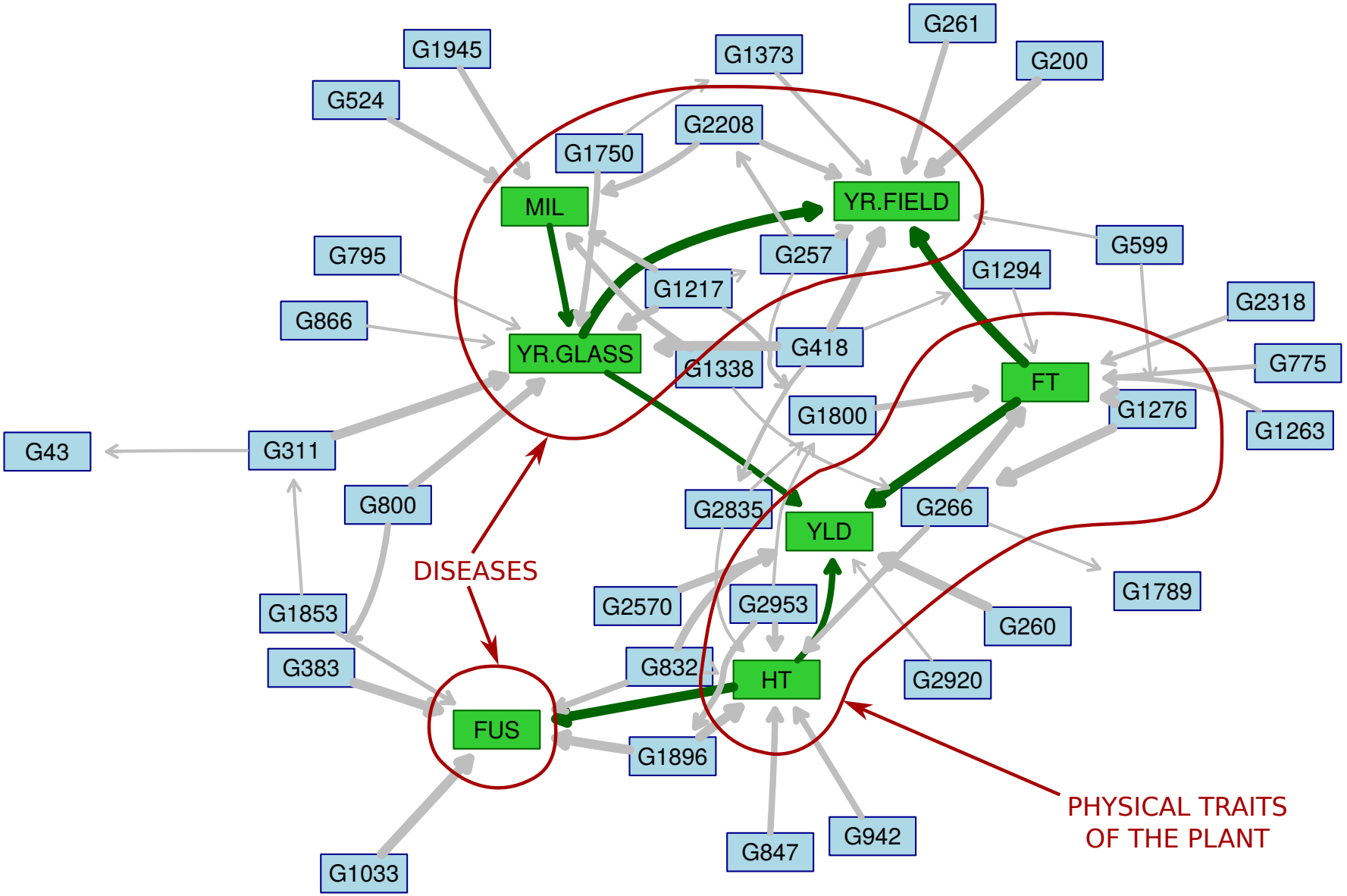
```
predicted = parLapply(kcv, cl = cluster, function(test) {
  # create matrices to store the predicted values.
  pred = matrix(0, nrow = length(test), ncol = length(traits))
  post = matrix(0, nrow = length(test), ncol = length(traits))
  colnames(pred) = colnames(post) = traits
  # split training and test.
  dtraining = data[-test, ]
  dtest = data[test, ]
  # fit the model on the training data.
  model = fit.the.model(dtraining, traits, genes, alpha = alpha)
  fitted = bn.fit(model, dtraining[, nodes(model)])
  # subset the test data.
  dtest = dtest[, nodes(model)]
  # predict each trait in turn, given all the parents.
  for (t in traits)
    pred[, t] = predict(fitted, node = t, data = dtest[, nodes(model)])
  # predict each trait in turn, given all the genes.
  for (t in traits)
    post[, t] = predict(fitted, node = t,
                        data = dtest[, names(dtest) %in% genes, drop = FALSE],
                        method = "bayes-lw", n = 1000)
  return(list(model = fitted, pred = pred, post = post))
})
```

Averaging the Models from Cross-Validation

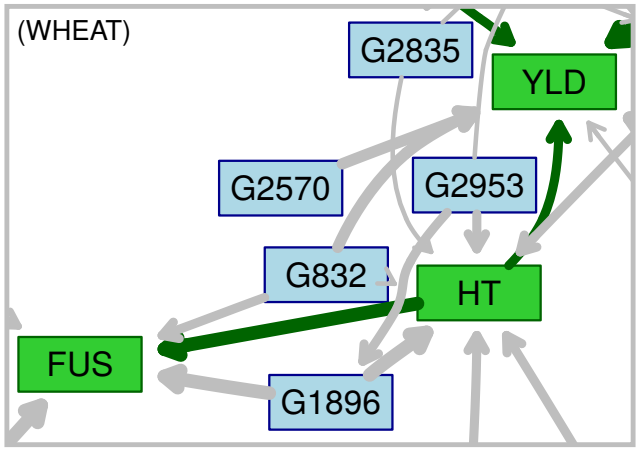
```
average.the.model = function(batch, data) {
  # gather all the arc lists.
  arclist = list()
  for (i in seq_along(batch)) {
    # extract the models.
    run = batch[[i]]$models
    for (j in seq_along(run))
      arclist[[length(arclist) + 1]] = arcs(run[[j]])
  }#FOR
  # compute arc strengths.
  nodes = unique(unlist(arclist))
  str = custom.strength(arclist, nodes = nodes)
  # estimate the significance threshold and average the networks.
  averaged = averaged.network(str)
  # subset the network to remove isolated nodes.
  relnodes = nodes(averaged)[sapply(nodes, degree, object = averaged) > 0]
  averaged2 = subgraph(averaged, relnodes)
  str2 = str[(str$from %in% relnodes) & (str$to %in% relnodes), ]
  # save the fitted averaged network.
  fitted = bn.fit(averaged2, data[, nodes(averaged2)])

  return(list(model = averaged2, strength = str2, fitted = fitted))
}#AVERAGE.THE.MODEL
```

UNIVERSITY OF OXFORD

The Averaged Bayesian Network (44 nodes, 66 arcs)



Spotting Confounding Effects



Traits and genes can interact in complex ways that may not be obvious when they are studied individually, but that can be explained by considering neighbouring variables in the network.
An example: yield apparently increases with FUS disease scores!

What we are actually measuring is the confounding effect of the plant's height (FUS $\leftarrow$ HT $\rightarrow$ YLD); if we simulate FUS and yield conditional on each quartile of height, FUS has a negative effect on yield.

We can verify this by simulation using conditional probability queries implemented in **bnlearn** in the `cpquery()` and `cpdist()` functions.

Spotting Confounding Effects

```
sim = cpdist(fitted, node = c("FUS", "YLD"),
        evidence = (HT > quantile(wheat$HT, 0.75)))
cor(sim$FUS, sim$HT)
```

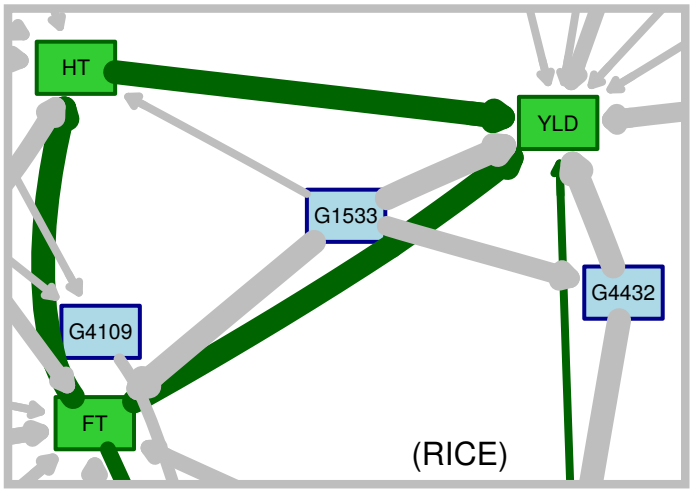
```
sim = cpdist(fitted, node = c("FUS", "YLD"),
        evidence = (HT < quantile(wheat$HT, 0.75)) &
                   (HT > quantile(wheat$HT, 0.50)))
cor(sim$FUS, sim$HT)
```

```
sim = cpdist(fitted, node = c("FUS", "YLD"),
        evidence = (HT < quantile(wheat$HT, 0.50)) &
                   (HT > quantile(wheat$HT, 0.25)))
cor(sim$FUS, sim$HT)
```

```
sim = cpdist(fitted, node = c("FUS", "YLD"),
        evidence = (HT < quantile(wheat$HT, 0.25)))
cor(sim$FUS, sim$HT)
```

Separating Direct and Indirect Effects

Living beings work as complex systems in which all parts interact with each other; hence it is important to separate its direct and indirect effects. Normally that would require expensive experiments; but it can be done in BNs with Pearl's causal inference. Consider gene G1533 in the rice BN: it is putative causal for yield (YLD), height (HT) and flowering time (FT).



- The difference in mean between the two homozygotes is +4.5cm in HT, +2.28 weeks in FT and +0.28 t/ha in YLD.
- Controlling for YLD and FT, the difference for HT halves (+2.1cm);
- Controlling for YLD and HT, the difference for FT is about the same (+2.3 weeks);
- Controlling for HT and FT the difference for YLD halves (+0.16 t/ha).

We can determine by simulation that the gene has a direct causal effect on FT and that the effect on the other traits is partly indirect because it is much smaller in our simulated experiments.

Separating Direct and Indirect Effects

```
control.ht = mutilated(bn.net(fitted), list("YLD" = 0, "FT" = 0))
control.ht = bn.fit(control.ht, indica[, nodes(control.ht)])
sim.aa = cpdist(control.ht, node = c("HT"), evidence = list(G1533 = 0),
        method = "lw")
sim.AA = cpdist(control.ht, node = c("HT"), evidence = list(G1533 = 2),
        method = "lw")
colMeans(sim.AA) - colMeans(sim.aa)
```

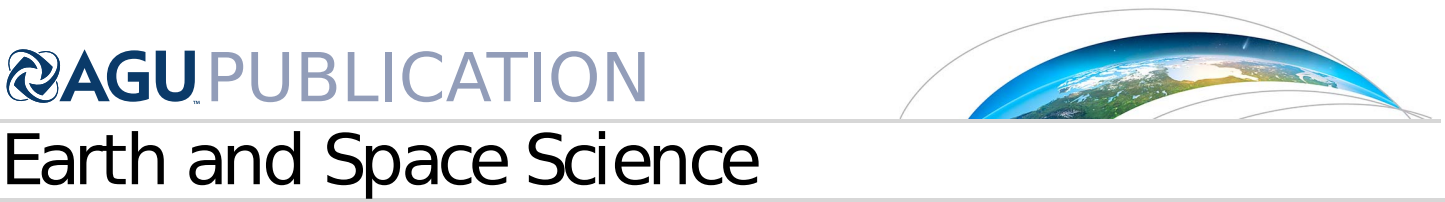
```
control.ft = mutilated(bn.net(fitted), list("YLD" = 0, "HT" = 0))
control.ft = bn.fit(control.ft, indica[, nodes(control.ft)])
sim.aa = cpdist(control.ft, node = c("FT"), evidence = list(G1533 = 0),
        method = "lw")
sim.AA = cpdist(control.ft, node = c("FT"), evidence = list(G1533 = 2),
        method = "lw")
colMeans(sim.AA) - colMeans(sim.aa)
```

```
control.yld = mutilated(bn.net(fitted), list("FT" = 0, "HT" = 0))
control.yld = bn.fit(control.yld, indica[, nodes(control.yld)])
sim.aa = cpdist(control.yld, node = c("YLD"), evidence = list(G1533 = 0),
        method = "lw")
sim.AA = cpdist(control.yld, node = c("YLD"), evidence = list(G1533 = 2),
        method = "lw")
colMeans(sim.AA) - colMeans(sim.aa)
```

Case Study: Environmental Statistics and Epidemiology

Another challenging application I worked on with **bnlearn** is an environmental data analysis on various air pollutants in English regions and their effect on public health. Big data (28 millions records) with missing values (again in the millions) and heterogeneous variables (continuous and discrete).
This prompted me to implement several things:

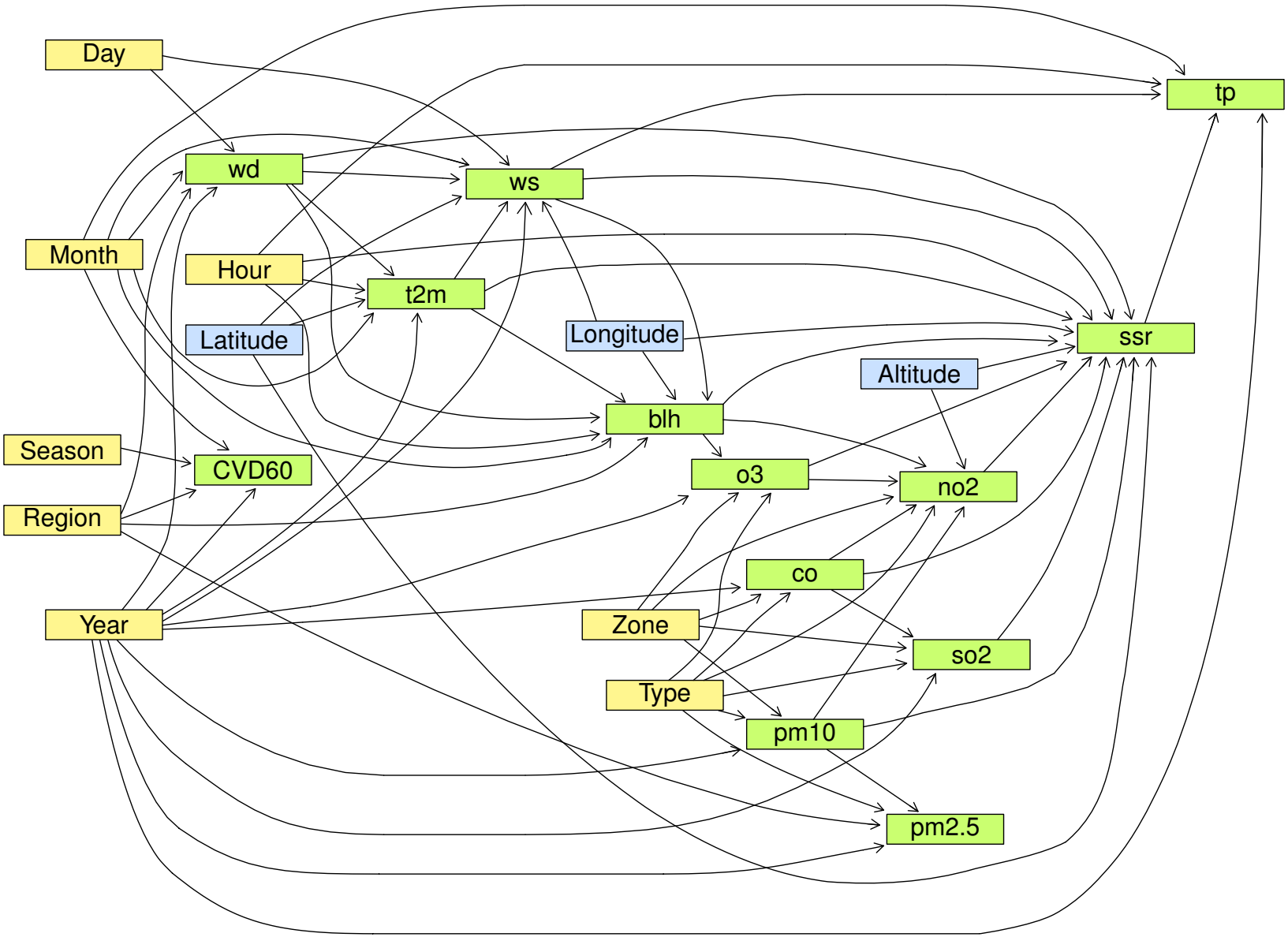
- the Structural EM algorithm to learn the structure of a BN in the presence of missing values, in `structural.em()`;
- hybrid (conditional Gaussian) BNs that can handle both discrete and continuous data at the same time;
- parallel parameter learning in `bn.fit()`.



Modelling Air Pollution, Climate and Health Data Using Bayesian Networks: a Case Study of the English Regions.
C. Vitolo *et al.*, Earth and Space Science (2017).

UNIVERSITY OF OXFORD

The Bayesian Network



Parallel Structural EM (Initialisation)

The main challenge in this analysis was to combine missing data imputation with EM with structure learning while using parallel computing, which was crucial given the size of the data.

```
# initialise the cluster, load bnlearn and export the blacklist.
library(parallel)
cl = makeCluster(20)
invisible(clusterEvalQ(cl, library(bnlearn)))
clusterExport(cl, "bl")

# split the data and export one part to each slave.
split = split(sample(nrow(training)), seq(length(cl)))

for (i in seq_along(split)) {

  data_split = training[split[[i]], , drop = FALSE]
  clusterExport(cl[i], "data_split")

}#FOR
```

Parallel Structural EM (Single Iteration)

```
# export the current network.
dagCurrent = dagNew
bnCurrent = bnNew
clusterExport(cl, c("dagCurrent", "bnCurrent"))

# expectation step: impute the missing data points on the data splits.
current = training
clusterEvalQ(cl, complete = impute_split())

# maximisation step: learn one network structure from each split.
models = parLapply(cl, seq(length(cl)), function(...) {

  hc(complete, blacklist = bl, start = dagCurrent)

})

# average the networks (and make sure the result is completely directed).
strengthNew = custom.strength(models, nodes = nodes(dagCurrent))
dagNew = averaged.network(strengthNew)
dagNew = cextend(dagNew)
```

Parallel Structural EM (Single Iteration)

```
# if there was no change, the network from the previous iteration is final.
if (isTRUE(all.equal(dagCurrent, dagNew)))
  break

# retrieve the imputed values.
for (i in incompleteColumns) {

 imputed = clusterCall(cl, function(col) complete[, col], col = i)
 current[unlist(split), i] = unlist(imputed)

}#THEN

# fit the parameters.
bnNew = bn.fit(dagNew, data = current, keep.fitted = FALSE)
```

So, in each iteration of EM we perform both imputation (`impute()`, the E step) and structure learning (`hc()`, the M step) in a distributed manner thanks to the **parallel** package.

Conclusions

- **bnlearn** is an R package that implements structure learning, parameter learning and inference for Bayesian networks: its aim is to provide a complete, integrated workflow.
- **bnlearn** aims to be useful for both methodological research and simulation studies; and for analysing challenging real-world data.
- **bnlearn** is designed to be modular, decoupling algorithms and statistical approaches; scalable, thanks to an efficient C backend; and with a public interface that is easy to use.
- **bnlearn** has been continuously maintained and it is under ongoing development; proposals and contributions are welcome.