The `bnclassify` package

Bojan Mihaljevic, Concha Bielza, Pedro Larranaga

2015-07-27

Contents

1 Introduction

The `bnclassify` package implements algorithms for learning discrete Bayesian network classifiers from data. It handles both incomplete and complete data, although it is much better suited for the latter. Prediction with incomplete data is notably slower, rendering the wrapper learning algorithms infeasible in some cases, whereas parameter estimation is no longer that of maximum likelihood.

We begin with an example showing the main functionalities and then go into some detail with structure and parameter learning, prediction, cross-validation, and how to leverage related `R` packages.

2 An example

This sections shows some of the main functionalities.

First, we load the package and an included data set, `car`.

```
library(bnclassify)
data(car)
summary(car)
#>    buying      maint       doors      persons     lug_boot    safety
#>  low  :432   high :432   2    :432   2   :576   big  :576   high:576
#>  med  :432   low  :432   3    :432   4   :576   med  :576   low :576
#>  high :432   med  :432   4    :432   more:576   small:576   med :576
#>  vhigh:432   vhigh:432   5more:432
#>    class
#>  unacc:1210
#>  acc  : 384
#>  good :  69
#>  vgood:  65
```

Now, we the learn a naive Bayes from the `car` data set.

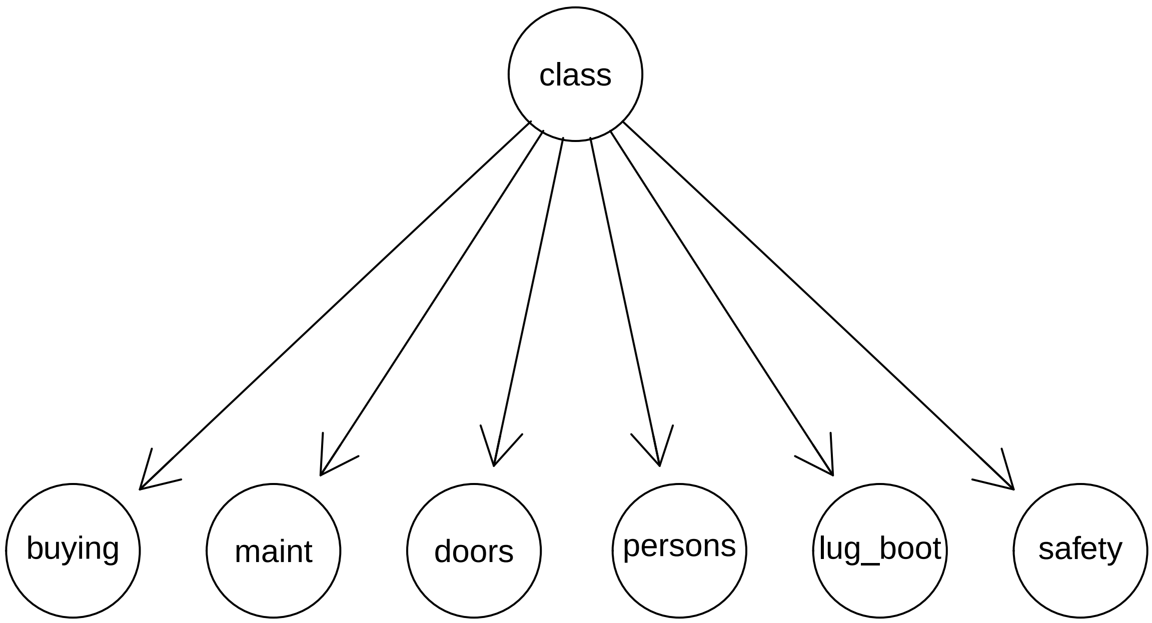
```
a <- nb('class', car)
a
#>
#>   Bayesian network classifier
#>
#>   class variable:        class
#>   num. features:   6
#>   arcs:   6
#>   learning algorithm:    nb
```

`nb` has returned a `bnc_dag` object, which contains just the network structure, without any parameters.

We can query this object for its features, it factorization type (e.g., whether is a naive Bayes), or plot its network structure.

```
features(a)
#> [1] "buying"   "maint"    "doors"    "persons"  "lug_boot" "safety"
is_nb(a)
#> [1] TRUE
```

```
plot(a)
```



For more functions to query a `bnc_dag` object, see `?bnc_dag_object`.

We need to learn the parameters before we can classify unseen data. We do this with the `lp` function.

```
b <- lp(a, car, smooth = 1)
```

`lp` returns a fully specified Bayesian network, an object of class `bnc_bn`.

We can get the CPT of each variable, including the class, with `params`. So, the class prior is

```
params(b)$class
#> class
#>      unacc        acc       good      vgood 
#> 0.69919169 0.22228637 0.04041570 0.03810624
```

where is the CPT for `buying` is

```
params(b)$class
#> class
#>      unacc        acc       good      vgood 
#> 0.69919169 0.22228637 0.04041570 0.03810624
```

For more functions that can be called on a `bnc_bn` object see `?bnc_bn_object`

Once we have fit parameters, we can predict the class or class posterior of unseen data (although in this example it is the data we used to learn the model).

```
p <- predict(b, car, prob = TRUE)
head(p)
#>          unacc          acc         good        vgood
#> [1,] 0.9999978 2.170707e-06 6.993227e-08 2.896447e-09
#> [2,] 0.9993626 6.328439e-04 4.505620e-06 4.665331e-09
#> [3,] 0.9990724 9.227497e-04 4.495399e-06 3.964044e-07
#> [4,] 0.9999966 3.196080e-06 9.119657e-08 8.642164e-08
#> [5,] 0.9990625 9.315006e-04 5.873884e-06 1.391584e-07
#> [6,] 0.9986243 1.358020e-03 5.859692e-06 1.182227e-05
p <- predict(b, car)
head(p)
#> [1] unacc unacc unacc unacc unacc unacc
#> Levels: unacc acc good vgood
```

We can estimate the classifier's predictive accuracy on the training set

```
accuracy(p, car$class)
#> [1] 0.8709491
```

or with cross-validation.

```
cv(b, car, k = 10, dag = FALSE)
#> [1] 0.8581917
```

3 Structure learning

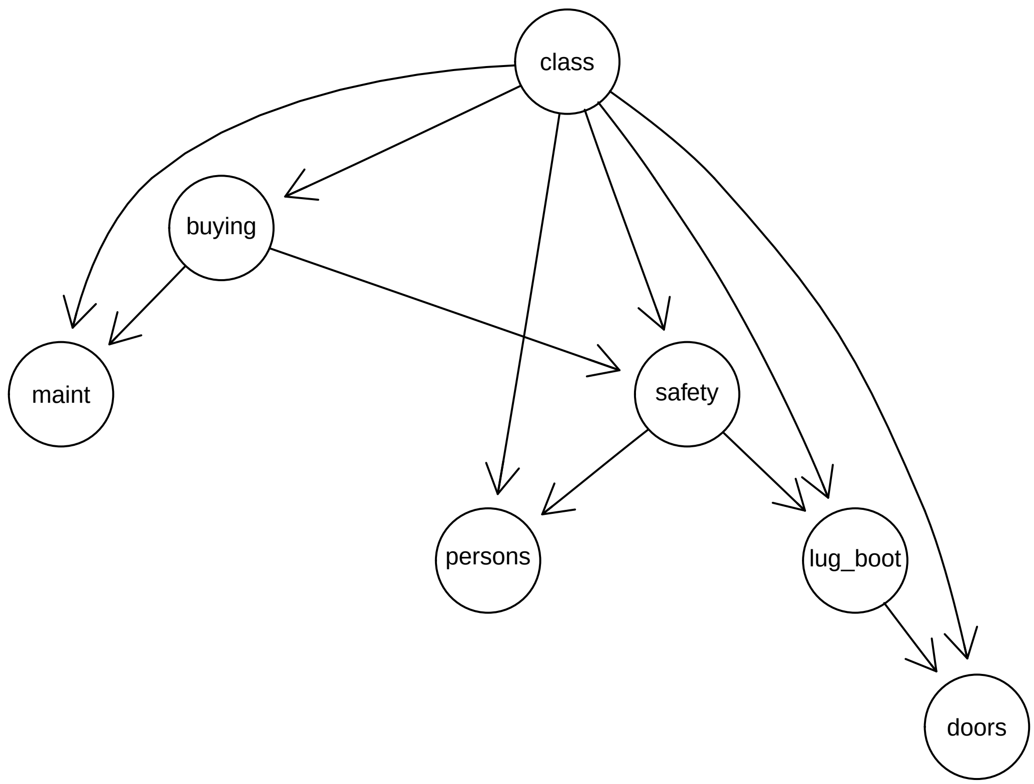
This section briefly lists the available structure learning algorithms. For additional information see `?bnclassify` and the documentation of each particular function regarding the available options.

3.1 The Chow-Liu algorithm

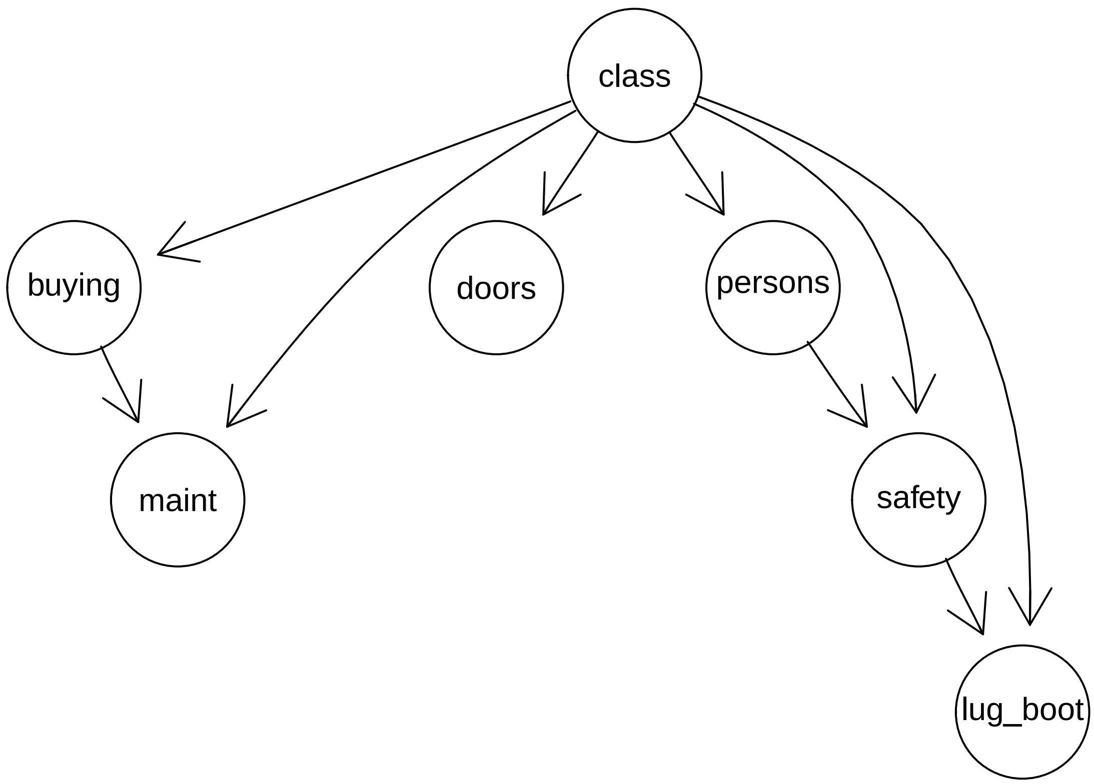
For some network scores, the Chow-Liu algorithm can efficiently (time quadratic in the number of features) learn optimal one-dependence estimators (i.e., with each feature conditioned on at most one feature). For three such scores, the log-likelihood, the BIC and the AIC, the `tan_cl` function learns the Bayesian network classifier using the Chow-Liu algorithm.

We set the score with the `score` argument.

```
t <- tan_cl(class = 'class', dataset = car)
ta <- tan_cl(class = 'class', dataset = car, score = 'aic')
plot(t)
```




```
plot(ta)
```



We can check whether the obtained structures are indeed one-dependence estimators.

```
is_ode(t)
#> [1] TRUE
is_nb(t)
#> [1] FALSE
is_ode(ta)
#> [1] TRUE
is_nb(ta)
#> [1] FALSE
```

Note that the BIC and AIC scores may render a forest instead of a tree in the features subgraph. Log-likelihood, on the other hand, always returns the maximal tree-like network.

See `?tan_chowliu` for more information on the Chow-Liu algorithm for Bayesian network classifiers.

3.2 Wrapper

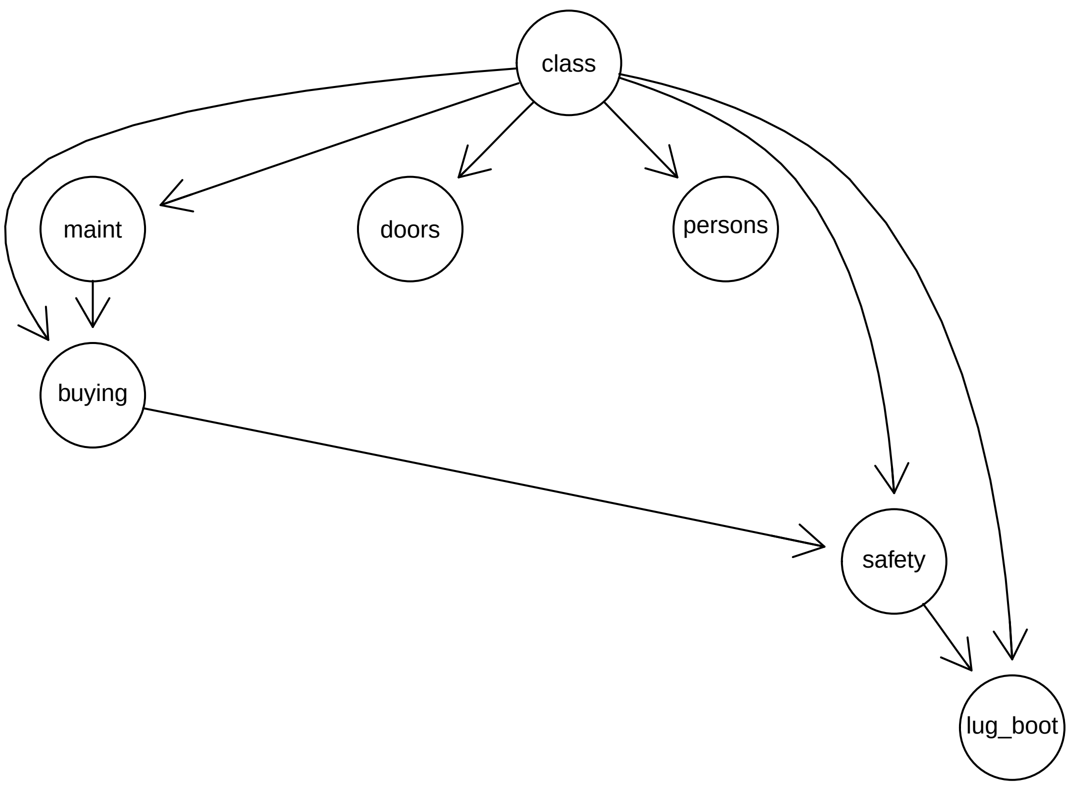
Wrapper learners search the space of structures and select the one that optimizes predictive performance. This can yield accurate classifiers but is more time consuming than the Chow-Liu algorithm. Note that this is especially true if the data contains missing values.

Below are examples of four wrapper learning algorithms. Two of them produce one-dependence estimators (`tan_hc` and `tan_hcsp`) whereas two produce `semi-naive Bayes' structures.`

See?wrapper‘ for more information.

The one-dependence estimators:

```
set.seed(0)
a <- tan_hc('class', car, k = 10, epsilon = 0, smooth = 1)
b <- tan_hcsp('class', car, k = 10, epsilon = 0, smooth = 1)
is_ode(a)
#> [1] TRUE
is_ode(b)
#> [1] TRUE
plot(a)
```

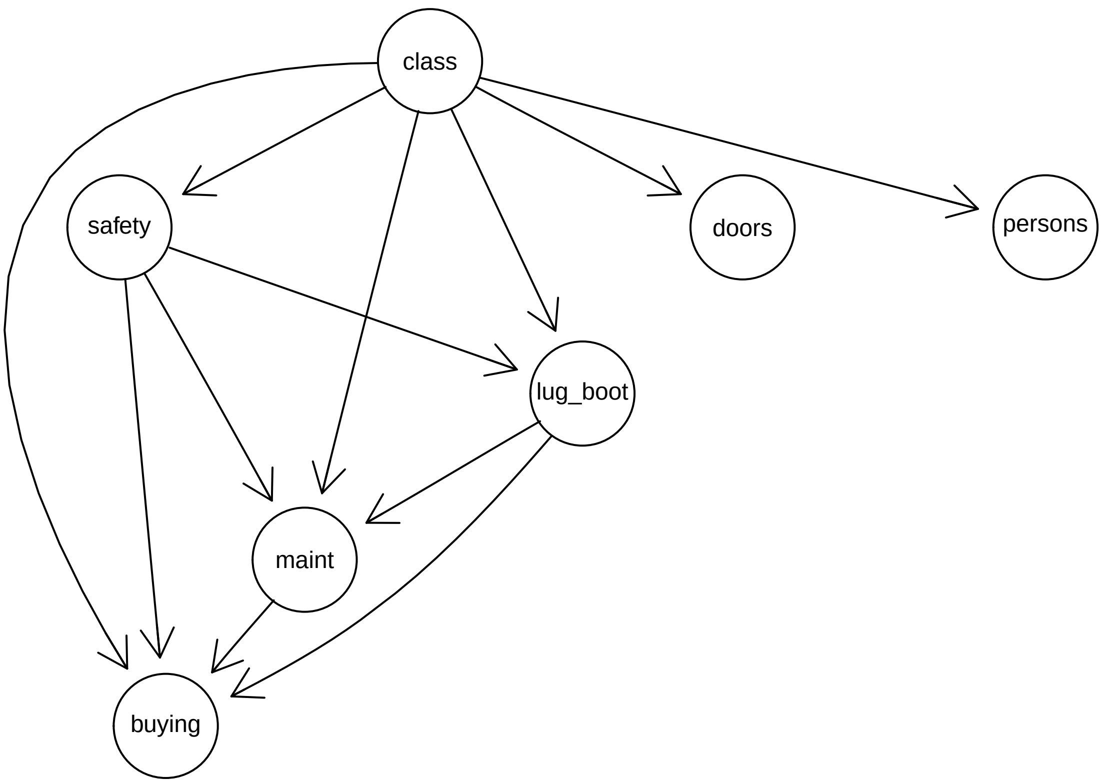


We can check whether they effectively are one-dependence estimators

```
is_ode(a)
#> [1] TRUE
is_ode(b)
#> [1] TRUE
```

The semi-naive structure learners:

```r
c <- bsej('class', car, k = 10, epsilon = 0, smooth = 1)
d <- fssj('class', car, k = 10, epsilon = 0, smooth = 1)
is_ode(c)
#> [1] FALSE
is_ode(d)
#> [1] TRUE
is_semi_naive(c)
#> [1] TRUE
is_semi_naive(d)
#> [1] TRUE
plot(c)
```



4 Parameter estimation

You may use the `bnc()` function as shorthand for the chained application of structure learning and `lp()`. Provide the name of the learning function (e.g., `tan_cl`) as first argument.

```r
a <- tan_cl('class', car, score = 'aic')
a <- lp(a, car, smooth = 1)
b <- bnc('tan_cl', 'class', car, smooth = 1, dag_args = list(score = 'aic'))
```

```r
identical(a, b)
#> [1] TRUE
```

4.1 Parameter weighting

For naive Bayes, one can combine maximum likelihood and Bayesian parameter estimation with posterior feature parameter weighting. This involves exponentiating the features' CPT entries by a value between 0 and 1 and can alleviate some of the negative effects of redundancy. See `?awnb` for more information.

We use `lpawnb` instead of `lp`.

```r
a <- nb('class', car)
b <- lp(a, car, smooth = 1)
c <- lpawnb(a, car, smooth = 1, trees = 20, bootstrap_size = 0.5)
sum(abs(params(b)$safety - params(c)$safety))
#> [1] 0.1308008
```

While this is intented for naive Bayes you can use it with other classifiers.

```r
t <- tan_cl('class', car)
t <- lp(t, dataset = car, smooth = 1)
ta <- lpawnb(t, car, smooth = 1, trees = 10, bootstrap_size = 0.5)
params(t)$buying
#>        class
#> buying       unacc        acc       good      vgood
#>   low   0.21334432 0.23195876 0.64383562 0.57971014
#>   med   0.22158155 0.29896907 0.32876712 0.39130435
#>   high  0.26771005 0.28092784 0.01369863 0.01449275
#>   vhigh 0.29736409 0.18814433 0.01369863 0.01449275
params(ta)$buying
#>        class
#> buying       unacc        acc       good      vgood
#>   low   0.22865173 0.24033379 0.52770941 0.49147055
#>   med   0.23370788 0.27825807 0.35799258 0.39169316
#>   high  0.26067042 0.26843623 0.05714901 0.05841814
#>   vhigh 0.27696997 0.21297192 0.05714901 0.05841814
```

5 Predicting

5.1 0 probabilities

If for some instance there is 0 probability for each class, then a uniform distribution over the classes is returned (not the class prior).

```r
nb <- nb('class', car)
nb <- lp(nb, car[c(1, 700), ], smooth = 0)
predict(object = nb, newdata = car[1000:1001, ], prob = TRUE)
#>      unacc  acc good vgood
#> [1,]  0.25 0.25 0.25  0.25
#> [2,]  0.25 0.25 0.25  0.25
```

5.2 Incomplete data

For instances that have missing (`NA`) values, `bnclassify` uses the `gRain` package to compute its class posterior, since `gRain` implements exact inference for Bayesian networks. This is much slower than the prediction for complete data implemented in `bnclassify`.

```r
library(microbenchmark)
nb <- nb('class', car)
nb <- lp(nb, car, smooth = 0)
gr <- as_grain(nb)
microbenchmark(predict(object = nb, newdata = car, prob = TRUE))
#> Unit: milliseconds
#>                                              expr      min     lq    mean
#>  predict(object = nb, newdata = car, prob = TRUE) 6.347417 7.0568 9.08768
#>   median       uq      max neval
#>  7.41699 7.993295 44.79595   100
microbenchmark(gRain::predict.grain(gr, 'class', newdata = car),
                               times = 1)
#> Unit: seconds
#>                                              expr      min       lq
#>  gRain::predict.grain(gr, "class", newdata = car) 4.187424 4.187424
#>      mean   median       uq      max neval
#>  4.187424 4.187424 4.187424 4.187424     1
```

With even a single missing value in the data set, the prediction can become notably slower. This is relevant when performing cross-validation, such as within wrapper learning.

```r
a <- bnc('nb', 'class', car, smooth = 1)
car_cv <- car[1:300, ]
microbenchmark::microbenchmark(cv(a, car_cv, k = 2, dag = FALSE), times = 3e1)
#> Unit: milliseconds
#>                               expr     min       lq     mean   median
#>  cv(a, car_cv, k = 2, dag = FALSE) 17.1447 18.36256 23.58018 19.29741
#>        uq      max neval
#>  20.29934 79.96117    30

car_cv[1, 4] <- NA
microbenchmark::microbenchmark(cv(a, car_cv, k = 2, dag = FALSE), times = 3e1)
#> Unit: milliseconds
```

```
#>                                 expr      min       lq     mean   median
#>  cv(a, car_cv, k = 2, dag = FALSE) 50.47248 52.12184 63.54114 55.40608
#>        uq      max neval
#>  84.26659 114.5849    30
```

6 Cross-validation

To perform cross valiation, pass a list of classifiers (or a single one) to the 'cv' function. Each classifier may be a `bnc_dag` or a `bnc_bn` object.

In the example below, we compare a naive Bayes, a weighted naive Bayes, and a one-dependence estimators with 3-fold cross-validation. We keep the structures fixed (`dag = FALSE`) and only learn parameters from the training sets.

```
data(voting)
dag <- nb('Class', voting)
a <- lp(dag, voting, smooth = 1)
b <- lpawnb(dag, voting, smooth = 1, trees = 40, bootstrap_size = 0.5)
c <- bnc('tan_cl', 'Class', voting,  smooth = 1)
r <- cv(list(a, b, c), voting, k = 3, dag = FALSE)
r
#> [1] 0.9034483 0.9494253 0.9517241
```

If we wanted to also perform structure learning, we would need to set `dag = TRUE` (this would have only affected the one-dependence estimator, since naive Bayes' structure is fixed).

7 Miscelaneous

You can compute the log-likelihood of a network with `compute_ll`.

```
a <- bnc('tan_cl', 'class', car, smooth = 0.01)
b <- bnc('nb', 'class', car, smooth = 0.01)
compute_ll(a, car)
#> [1] -13250.74
compute_ll(b, car)
#> [1] -13503.84
```

Also the (conditional) mutual information between two variables. Mutual information of `maint` and `buying`:

```
cmi('maint', 'buying', car)
#> [1] 0
```

and of `maint` and `buying` conditioned to `class`:

```r
cmi('maint', 'buying', car, 'class')
#> [1] 0.07199921
```

8 Interface to other packages

You can convert a `bnclassify` object to `bnlearn`, `gRain` and `mlr` objects and use functionalities from those packages.

8.1 Selecting features with mlr

Some of the implemented algorithms, such as the `fssj` and `bsej` perform implicit feature selection. However, 'outer' loop of feature selection is not within the scope of `bnclassify` and best done with another package such as `mlr`.

Assuming you have mlr installed, call `as_mlr()` to convert a `bnc_bn` to an mlr *learner*. This allows you to use mlr functionalities: selecting features, benchmarking, etc.

Set up a mlr task

```r
library(mlr)
#> Loading required package: BBmisc
#> Loading required package: ggplot2
#> Loading required package: ParamHelpers
ct <- mlr::makeClassifTask(id = "compare", data = car, target = 'class',
                           fixup.data = "no", check.data = FALSE)
```

Learn a naive Bayes and convert to mlr learner

```r
nf <- lp(nb('class', car), car, 1)
bnl <- as_mlr(nf, dag = TRUE)
```

Then use wrapper feature selection

```r
ctrl = makeFeatSelControlSequential(alpha = 0, method = "sfs")
rdesc = makeResampleDesc(method = "Holdout")
sfeats = selectFeatures(learner = bnl, task = ct, resampling = rdesc,
                        control = ctrl, show.info = FALSE)
sfeats$x
#> [1] "buying"
detach('package:mlr')
```

8.2 Operate with Bayesian networks with gRain and bnlearn

`gRbase` and `bnlearn` provide multiple functionalities for querying and manipulating Bayesian networks. We can convert a `bnc_bn` to a `gRain` via `as_grain()`. From the `gRain` object you can then obtain a `bnlearn` one (see `bnlearn` docs).

Using `as_grain`:

```
a <- lp(nb('class', car), car, smooth = 1)
g <- as_grain(a)
gRain::querygrain.grain(g)$buying
#> buying
#>       low       med      high     vhigh
#> 0.2488415 0.2495832 0.2507330 0.2508423
```

9 Runtime

The wrapper algorithms can be computationlly intensive, especially with large data sets. I get the following times for `bsej` and `tan_hc` on my Windows 2.80 GHz, 16 GB RAM machine.

```
microbenchmark::microbenchmark(
  bsej = {b <- bsej('class', car, k = 10, epsilon = 0)} ,
  tan_hc = {t <- b <- tan_hc('class', car, k = 10, epsilon = 0)},
  times = 10)
```

```
#> Unit: seconds
#>    expr      min       lq     mean   median       uq      max neval
#>    bsej 2.578518 2.720906 3.188944 3.341617 3.389287 3.677820    10
#>  tan_hc 1.968562 2.201919 2.238606 2.246080 2.361516 2.420327    10
```

10-fold cross-validation of these two classifiers should take rougly 10 times more than learning them the full data set.

```
microbenchmark::microbenchmark(
  cv(list(b, t), car, k = 10, dag = TRUE, smooth = 0.01), times = 10)
```

```
#> Unit: seconds
#>                                                     expr      min       lq
#>  cv(list(b, t), car, k = 10, dag = TRUE, smooth = 0.01) 49.64341 50.63624
#>      mean   median       uq      max neval
#>  51.28273 51.45354 51.96889 52.28374    10
```

Thus, it takes about a minute to cross-validate these two classifiers on the car data (6 features, 1728 instances).

Note that non-wrapper classifiers are much faster.

```
nb <- nb('class', car)
tcl <- tan_cl('class', car)
microbenchmark::microbenchmark(
  cv(list(nb, tcl), car, k = 10, dag = TRUE, smooth = 0.01), times = 10)
```

```
#> Unit: milliseconds
#>                                                         expr      min
#>  cv(list(nb, tcl), car, k = 10, dag = TRUE, smooth = 0.01) 709.4578
#>        lq     mean   median       uq      max neval
#>  712.8001 721.6526 720.6743 724.6895 737.8496    10
```

Let us a look at a data set with 36 features.

```
library(mlbench)
data(Soybean)
dim(Soybean)
```

```
#> [1] 683  36
```

Inference with incomplete data is slow. Thus, we remove incomplete instances.

```
soy_complete <- na.omit(Soybean)
```

`bsej` takes almost 10 minutes.

```
microbenchmark::microbenchmark(
  b <- bsej('Class', soy_complete, k = 10, epsilon = 0),
  times = 1)
```

```
#> Unit: seconds
#>                                                     expr      min       lq
#>  b <- bsej("Class", soy_complete, k = 10, epsilon = 0) 569.6894 569.6894
#>      mean   median       uq      max neval
#>  569.6894 569.6894 569.6894 569.6894     1
```

We could expect a 10-fold cross-validation to take around 100 minutes. Note that we have a nested 10×10 cross-validation, though. Decreasing k would decrease runtime and increasing epsilon would likely do the same.