`bnstruct`: an R package for Bayesian Network Structure Learning

Francesco Sambo, Alberto Franzin

March 12, 2015

1 Introduction

Bayesian Networks are a powerful tool for probabilistic inference among a set of variables, modeled using a directed acyclic graph. However, one often does not have the network, but only a set of observations, and wants to reconstruct the network that generated the data. The `bnstruct` package provides objects and methods for learning the structure and parameters of the network in various situations, such as in presence of missing data, for which it is possible to perform *imputation* (guessing the missing values, by looking at the data). The package also contains methods for learning using the Bootstrap technique. Finally, `bnstruct`, has a set of additional tools to use Bayesian Networks, such as methods to perform belief propagation.

This document is intended to show some examples of how `bnstruct` can be used to learn and use Bayesian Networks. First we describe how to manage data sets, how to use them to discover a Bayesian Network, and finally how to perform some operations on a network. Complete reference for classes and methods can be found in the package documentation.

1.1 Overview

We provide here some general informations about the context for understanding and using properly this document.

1.1.1 The data

A *dataset* is a collection of rows, each of which is composed by the same number of values. Each value corresponds to an observation of a *variable*, which is a feature, an event or an entity considered significant and therefore measured. In a Bayesian Network, each variable is associated to a node. The number of variables is the *size* of the network. Each variable has a certain range of values it can take. If the variable can take any possible value in its range, it is called a *continuous* variable; otherwise, if a variable can only take some values in its range, it is a *discrete* variable. The number of values a variable can take is called its *cardinality*. A continuous variable has infinite cardinality; however, in order to deal with it, we have to restrict its domain to a smaller set of values, in order to be able to treat it as a discrete variable; this process is called *quantization*, the number of values it can take is called the number of *levels* of the quantization step, and we will therefore call the cardinality of a continuous variable the number of its quantization levels, with a little abuse of terminology.

In many real-life applications and contexts, it often happens that some observations in the dataset we are studying are absent, for many reasons. In this case, one may want to "guess" a reasonable (according to the other observations) value that would have been present in the dataset, if the observations was successful. This inference task is called *imputation*. In this case, the dataset with the "holes filled" is called the *imputed dataset*, while the original dataset with missing values is referred to as the *raw dataset*. In section **??** we show how to perform imputation in `bnstruct`.

Another common operation on data is the employment of resampling techniques in order to estimate properties of a distribution from an approximate one. This usually allows to have more confidence in the results. We implement the *bootstrap* technique, and provide it to generate samples of a dataset, with the chance of using it on both raw and imputed data.

1.1.2 Bayesian Networks

After introducing the data, we are now ready to talk about *Bayesian Networks*. A Bayesian Network (hereafter sometimes simply *network*, *net* or *BN* for brevity) is a probabilistic graphical model that encodes the conditional dependency relationships of a set of variables using a Directed Acyclic Graph (DAG). Each node of the graph represents one variable of the dataset; we will therefore interchange the terms *node* and *variable* when no confusion arises. The set of directed edges connecting the nodes forms the *structure* of the network, while the set of conditional probabilities associated with each variable forms the set of *parameters* of the net.

The problems of learning the structure and the parameters of a network from data define the *structure learning* and *parameter learning* tasks, respectively.

COMPLETARE

2 Installation

The latest version of `bnstruct` can be found at `http://github.com/sambofra/bnstruct`.

In order to install the package, it suffices to launch

```
R CMD INSTALL path/to/bnstruct
```

from a terminal, or to use R command `install.packages`.

Being hosted on GitHub, it is also possible to use Hadley Wickham's `install_github` tool from an R session:

```
> library("devtools")
> install_github("sambofra/bnstruct")
```

For Windows platforms, a binary executable will be provided.

`bnstruct` requires R ≥ 2.10, and depends on `bitops`, `igraph`, `Matrix` and `methods`. Package `Rgraphviz` is requested in order to plot graphs, but is not mandatory.

3 Data sets

The class that `bnstruct` provides to manage datasets is `BNDataset`. It contains all of the data and the informations related to it: raw and imputed data, raw and imputed bootstrap samples, and variable names and cardinality.

3.1 Creating a BNDataset

There are two ways to build a BNDataset: using two files containing respectively header informations and data, and manually providing the data table and the related header informations (variable names, cardinality and discreteness).

```
> dataset.from.data <- BNDataset(data = data,
+                                 discreteness = rep('d',4),
+                                 variables = c("a", "b", "c", "d"),
+                                 node.sizes = c(4,8,12,16))
>
> dataset.from.file <- BNDataset("path/to/data.file",
+                                 "path/to/header.file")
```

The key informations needed are:

1. the data;
2. the state of variables (discrete or continuous);
3. the names of the variables;
4. the cardinalities of the variables (if discrete), or the number of levels they have to be quantized into (if continuous).

Names and cardinalities/leves can be guessed by looking at the data, but it is strongly advised to provide *all* of the informations, in order to avoid problems later on during the execution.

Data can be provided in form of data.frame or matrix. It can contain NAs. By default, NAs are indicated with '?'; to specify a different character for NAs, it is possible to provide also the `na.string.symbol` parameter. The values contained in the data have to be numeric (real for continuous variables, integer for discrete ones). The default range of values for a discrete variable `X` is `[1,|X|]`, with `|X|` being the cardinality of `X`. The same applies for the levels of quantization for continuous variables. If the value ranges for the data are different from the expected ones, it is possible to specify a different starting value (for the whole dataset) with the `starts.from` parameter. E.g. by `starts.from=0` we assume that the values of the variables in the dataset have range `[0,|X|-1]`. Please keep in mind that the internal representation of bnstruct starts from 1, and the original starting values are then lost.

It is possible to use two files, one for the data and one for the metadata, instead of providing manually all of the info. bnstruct requires the data files to be in a format subsequently described. The actual data has to be in (a text file containing data in) tabular format, one tuple per row, with the values for each variable separated by a space or a tab. Values for each variable have to be numbers, starting from `1` in case of discrete variables. Data files can have a first row containing the names of the corresponding variables.

In addition to the data file, a header file containing additional informations can also be provided. An header file has to be composed by three rows of tab-delimited values:

1. list of names of the variables, in the same order of the data file;

2. a list of integers representing the cardinality of the variables, in case of discrete variables, or the number of levels each variable has to be quantized in, in case of continuous variables;

3. a list that indicates, for each variable, if the variable is continuous (`c` or `C`), and thus has to be quantized before learning, or discrete (`d` or `D`).

In case of need of more advanced options when reading a dataset from files, please refer to the documentation of the `read.dataset` method. Imputation and bootstrap are also available as separate routines (`impute` and `bootstrap`, respectively).

We provide two sample datasets, one with complete data (the `Asia` network) and one with missing values (the `Child` network), in the `extdata` subfolder; the user can refer to them as an example. The two datasets have been created with

```
> asia <- BNDataset("asia_10000.data",
+                   "asia_10000.header",
+                   starts.from=0)
> child <- BNDataset("Child_data_na_5000.data",
+                    "Child_data_na_5000.header",
+                    starts.from=0)
```

and are also available with

```
> asia  <- asia()
> child <- child()
```

3.2 Imputation

A dataset may contain various kinds of missing data, namely unobserved variables, and unobserved values for otherwise observed variables. We currently deal only with this second kind of missing data. The process of guessing the missing values is called *imputation*.

We provide the `impute` function to perform imputation.

```
> dataset <- BNDataset(data.file   = "path/to/file.data",
+                      header.file = "path/to/file.header")
> dataset <- impute(dataset)
```

Imputation is accomplished with the k-Nearest Neighbour algorithm. The number of neighbours to be used can be chosen specifying the `k.impute` parameter.

3.3 Bootstrap

`BNDataset` objects have also room for bootstrap samples, i.e. random samples with replacement of the original data with the same number of observations, both for raw and imputed data. Samples for imputed data are generated by imputing the corresponding sample of raw data. Therefore, by requesting imputed samples, also the raw samples will be generated.

We provide the `bootstrap` method for this.

```
> dataset <- BNDataset("path/to/file.data",
+                      "path/to/file.header")
> dataset <- bootstrap(dataset, num.boots = 100)
> dataset <- impute(dataset)
> dataset.with.imputed.samples <- bootstrap(dataset,
+                                           num.boots = 100,
+                                           imputation = TRUE)
```

The sample datasets provided have no bootstrap samples in them.

3.4 Using data

After a `BNDataset` has been created, it is ready to be used. The complete list of methods available for a `BNDataset` object is available in the package documentation; we are not going to cover all of the methods in this brief series of examples.

For example, one may want to see the dataset.

```
> # the following are equivalent:
> print(dataset)
> show(dataset)
> dataset
```

The main operation that can be done with a BNDataset is to get the data it contains. The main methods we provide are raw.data and imputed.data, which provide the raw and the imputed data, respectively. The data must be present in the object; conversely, an error will be raised. To avoid an abrupt termination of the execution in case of error, one may run these methods in a tryCatch() construct and manage the errors in case they happen. Another alternative is to test the presence of data before attempting to retrieve it, using the tester methods has.raw.data and has.imputed.data.

```
> options(max.print = 200, width = 60)
>
> dataset <- child()
> # if we want raw data
> raw.data(dataset)

FALSE         V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15
FALSE    [1,]  2  3  3 NA  1 NA  1  1  2   1   1   1  NA   2   2
FALSE    [2,] NA NA  2  1  1  2  1  2  2   1   2   1   2   2  NA
FALSE    [3,]  2  3  1  2  1 NA NA  2  2   1   2   2   2   2   2
FALSE    [4,]  2  4  1  1  1  3 NA  1  2  NA   3  NA   1   2   1
FALSE    [5,]  2  2  1 NA  2  4  1  1  1   1  NA   1  NA   2  NA
FALSE    [6,] NA  2 NA  2  1  4 NA  3  2   1   3   1   3   2   2
FALSE    [7,]  2  2  1  2 NA  4 NA  3 NA   1  NA   1   1  NA  NA
FALSE    [8,] NA  1  1 NA  3  1  1  1  2   2   2   1   4   2   2
FALSE    [9,]  2  3  2  2  1  3  1  2  2   1   2   1   2   2   2
FALSE   [10,]  2  4  1  1  1  3 NA NA  2  NA   3   3   2   2   1
FALSE         V16 V17 V18 V19 V20
FALSE    [1,]   2   3   2   1   2
FALSE    [2,]   2   2   1   2   2
FALSE    [3,]   1   2   1   2   2
FALSE    [4,]   3   1  NA   1  NA
FALSE    [5,]  NA   1   1   2   2
FALSE    [6,]   2   1   1   3   2
FALSE    [7,]  NA   1   1   1  NA
FALSE    [8,]   1   1  NA   4  NA
FALSE    [9,]  NA   1   1   2   2
FALSE   [10,]   1   1   2   2  NA
FALSE  [ reached getOption("max.print") -- omitted 4990 rows ]

> # if we want imputed dataset: this raises an error
> imputed.data(dataset)

FALSE Error in imputed.data(dataset):  The dataset contains no imputed
data.  Please impute data before learning.
FALSE See > ?impute for help.

> # with tryCatch we manage the error
> tryCatch(
+   imp.data <- imputed.data(dataset),
+   error = function(e) {
+     cat("Hey! Something went wrong. No imputed data present maybe?")
+     imp.data <<- NULL
+   }
+ )

FALSE Hey! Something went wrong. No imputed data present maybe?

> imp.data

FALSE NULL

> # test before trying
> if (has.imputed.data(dataset)) {
+   imp.data <- imputed.data(dataset)
+ } else {
+   imp.data <- NULL
+ }
> imp.data

FALSE NULL

> # now perform imputation on the dataset
> dataset <- impute(dataset)

FALSE bnstruct :: performing imputation ...
FALSE bnstruct :: imputation finished.

> imputed.data(dataset)

FALSE         V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15
FALSE    [1,]  2  3  3  2  1  3  1  1  2   1   1   1   1   2   2
FALSE    [2,]  2  4  2  1  1  2  1  2  2   1   2   1   2   2   1
FALSE    [3,]  2  3  1  2  1  3  1  2  2   1   2   2   2   2   2
FALSE    [4,]  2  4  1  1  1  3  1  1  2   1   3   1   1   2   1
FALSE    [5,]  2  2  1  2  2  4  1  1  1   1   3   1   1   2   2
FALSE    [6,]  2  2  1  2  1  4  1  3  2   1   3   1   3   2   2
FALSE    [7,]  2  2  1  2  2  4  1  3  2   1   3   1   1   2   2
FALSE    [8,]  2  1  1  2  3  1  1  1  2   2   2   1   4   2   2
FALSE    [9,]  2  3  2  2  1  3  1  2  2   1   2   1   2   2   2
FALSE   [10,]  2  4  1  1  1  3  1  2  2   1   3   3   2   2   1
FALSE         V16 V17 V18 V19 V20
FALSE    [1,]   2   3   2   1   2
FALSE    [2,]   2   2   1   2   2
FALSE    [3,]   1   2   1   2   2
FALSE    [4,]   3   1   1   1   2
FALSE    [5,]   1   1   1   2   2
FALSE    [6,]   2   1   1   3   2
FALSE    [7,]   1   1   1   1   2
FALSE    [8,]   1   1   1   4   2
FALSE    [9,]   1   1   1   2   2
FALSE   [10,]   1   1   2   2   2
FALSE  [ reached getOption("max.print") -- omitted 4990 rows ]
```

In order to retrieve bootstrap samples, one can use the `boots` and `imp.boots` methods for samples made of raw and imputed data. The presence of raw and imputed samples can be tested using `has.boots` and `has.imputed.boots`. Trying to access a non-existent sample (e.g. imputed sample when no imputation has been performed, or sample index out of range) will raise an error. The method `num.boots` returns the number of samples.

We also provide the `boot` method to directly access a single sample.

```
> # get imputed samples
> for (i in 1:num.boots(dataset))
+   print( boot(dataset, i) )
```

```
> # get raw samples
> for (i in 1:num.boots(dataset))
+   print( boot(dataset, i, use.imputed.data = TRUE) )
```

4 Bayesian Networks

Bayesian Network are represented using the `BN` object. It contains information regarding the variables in the network, the directed acyclic graph (DAG) representing the structure of the network, the conditional probability tables entailed by the network, and the weighted partially DAG representing the structure as learnt using bootstrap samples.

The following code will create a `BN` object for the `Child` network, with no structure nor parameters.

```
> dataset <- child()
> net     <- BN(dataset)
```

Then we can fill in the fields of `net` by hand. See the inline help for more details.

The method of choice to create a `BN` object is, however, to create it from a `BNDataset` using the `learn.network` method.

4.1 Network learning

When constructing a network starting from a dataset, the first operation we may want to perform is to learn a network that may have generated that dataset, in particular its structure and its parameters. `bnstruct` provides the `learn.network` method for this task.

```
> dataset <- child()
> net     <- learn.network(dataset)
```

The `learn.network` method returns a new `BN` object, with a new DAG (or WPDAG, if the structure learning has been performed using bootstrap – more on this later).

Here we briefly describe the two tasks performed by the method, along with the main options.

4.1.1 Structure learning

We provide three algorithms in order to learn the structure of the network, that can be chosen with the `algo` parameter. The first is the Silander-Myllymäki (`sm`) exact search-and-score algorithm (see Silander and Myllymaki [4]), that performs a complete evaluation of the search space in order to discover the best network; this algorithm may take a very long time, and can be inapplicable when discovering networks with more than 25–30 nodes. Even for small networks, users are strongly encouraged to provide meaningful parameters such as the layering of the nodes, or the maximum number of parents – refer to the documentation in package manual for more details on the method parameters.

The second algorithm (and the default one) is the Max-Min Hill-Climbing heuristic (`mmhc`, see Tsamardinos, Brown, and Aliferis [5]), that performs a statistical sieving of the search space followed by a greedy evaluation. It is considerably faster than the complete method, at the cost of a (likely) lower quality. Also note that in the case of a very dense network and lots of obsevations, the statistical evaluation of the search space may take a long time. Also for this algorithm there are parameters that may need to be tuned, mainly the confidence threshold of the statistical pruning.

The third method is the Structural Expectation-Maximization (`sem`) algorithm (Friedman [1, 2]), for learning a network from a dataset with missing values. It iterates a sequence of Expectation-Maximization (in order to “fill in” the holes in the dataset) and structure learning from the guessed dataset, until convergence. The structure learning used inside SEM, due to computational reasons, is MMHC.

Search-and-score methods also need a scoring function to compute an estimated measure of each configuration of nodes. We provide three of the most popular scoring functions, `BDeu` (Bayesian-Dirichlet equivalent uniform, default), `AIC` (Akaike Information Criterion) and `BIC` (Bayesian Information Criterion). The scoring function can be chosen using the `scoring.func` parameter.

```
> dataset <- child()
> net.1   <- learn.network(dataset,
+                            algo = "sem",
+                            scoring.func = "AIC")
> dataset <- impute(dataset)
> net.2   <- learn.network(dataset,
+                            algo = "mmhc",
+                            scoring.func = "BDeu",
+                            use.imputed.data = TRUE)
```

The structure learning task by default computes the structure as a DAG. We can however use bootstrap samples to learn a weighted partially DAG, in order to get a weighted confidence on the presence or absence of an edge in the structure (Friedman, Goldszmidt, and Wyner [3]). This can be done by providing the constructor or the `learn.network` method a `BNDataset` with bootstrap samples, and the additional parameter `bootstrap = TRUE`.

```
> dataset <- child()
> dataset <- bootstrap(dataset, 100, imputation = TRUE)
> net.1   <- learn.network(dataset,
+                            algo = "mmhc",
+                            scoring.func = "AIC",
+                            bootstrap = TRUE)
> # or, for learning from imputed data
> net.2   <- learn.network(dataset,
+                            algo = "mmhc",
+                            scoring.func = "AIC",
+                            bootstrap = TRUE,
+                            use.imputed.data = TRUE)
```

Structure learning can be performed also using the `learn.structure` method, which has a similar syntax, only requiring as first parameter an already initialized network for the dataset. More details can be found in the inline helper.

4.1.2 Parameter learning

Parameter learning is the operation that learns the conditional probabilities entailed by a network, given the data and the structure of the network. In `bnstruct` this is done by `learn.network` performing a Maximum-A-Posteriori (MAP) estimate of the parameters.

`bnstruct` also provides the `learn.params` method for this task.

5 Using a network

Once a network is created, it can be used. Here we briefly mention some of the basic methods provided in order to manipulate a network and access its components.

First of all, it is surely of interest to obtain the structure of a network. The `bnstruct` package provides the `dag()` and `wpdag()` methods in order to access the structure of a network learnt without and with bootstrap (respectively).

```
> dag(net)
> wpdag(net.boot)
```

Then we may want to retrieve the parameters, using the `cpts()` method.

```
> cpts(net)
```

Another common operation that we may want to perform is displaying the network, or printing its main informations, using the `plot()`, `print()` and `show()` methods. Note that the `plot()` method is flexible enough to allow some custom settings such as the choice of the colors of the nodes, and, more importantly, some threshold settings for the networks learnt with bootstrap. As default, the DAG of a network is selected for plotting, if available, otherwise the WPDAG is used. In case of presence of both the DAG and the WPDAG, in order to specify the latter as structure to be plotted, the `plot.wpdag` logical parameter is provided.

```
> print(net)
> plot(net) # regular DAG
> plot(net, plot.wpdag=T) # wpdag
> plot(net.boot)
```

The `show()` method is an alias for the `print()` method, but allows to print the state of an instance of an object just by typing its name in an `R` session.

```
> # TFAE
> print(net)
> show(net)
> net
```

6 Inference in networks

6.1 Belief Propagation

The `bnstruct` package provides a tool to perform belief propagation using a junction tree. This tool is the `InferenceEngine` object. It contains a copy of a network, an updated network, the adjacency matrix of the junction tree computed starting from the original network, the list of cliques of variables that form the nodes of the junction tree and the list of joint probability tables for the cliques composing the junction tree.

An `InferenceEngine` can be built from a network: in this case, the junction tree is immediately constructed.

```
> dataset <- asia() # or any other way to create a custom BNDataset
> net     <- BN(dataset)
> inf.eng <- InferenceEngine(net)
```

Belief propagation over the junction tree can be then performed using the `belief.propagation` method.

```
> dataset <- asia() # or any other way to create a custom BNDataset
> net     <- BN(dataset)
> inf.eng <- InferenceEngine(net)
> inf.eng <- belief.propagation(inf.eng)
```

Belief propagation can be fed with a list of observations. This can be done in two ways: as parameters in the method, or inserting them directly into the inference engine. Note that the two options are mutually exclusive, in the sense that the list of observations given as parameter replaces (in the whole `InferenceEngine` object returned) the observations contained in the engine. Furthermore, if a list of observations contains multiple observations of the same variable, only the last one is considered.

```
> dataset <- asia() # or any other way to create a custom BNDataset
> net     <- BN(dataset)
> net     <- learn.structure(net, dataset)
> net     <- learn.params(net, dataset)
> inf.eng <- InferenceEngine(net)
> inf.eng <- belief.propagation(inf.eng,
+                               observed.vars = c("Asia","X-ray"),
+                               observed.vals = c(1,1))
> print(updated.bn(inf.eng))
> # is equivalent to
> observations(inf.eng) <- list(c("Asia","X-ray"), c(1,1))
> inf.eng <- belief.propagation(inf.eng)
> plot(updated.bn(inf.eng))
```

6.2 The Expectation-Maximization algorithm

Package `bnstruct` can also use an `InferenceEngine` and a `BNDataset` to perform the Expectation-Maximization algorithm to estimate the parameters of the network. It suffices to use the em method, that returns an InferenceEngine containing an updated network with the newly estimated conditional probability tables.

It is possible to control the algorithm by specifying the threshold parameter, that specifies a threshold for the convergence of the algorithm.

```
> dataset <- child() # or any other way to create a custom BNDataset
> net     <- BN(dataset)
> net     <- learn.structure(net, dataset)
> net     <- learn.params(net, dataset)
> inf.eng <- InferenceEngine(net)
> inf.eng <- em(inf.eng, dataset)
> print(updated.bn(inf.eng))
```

6.3 The Structural EM algorithm

We provide an implementation of the Structural Expectation-Maximization algorithm (Friedman [1, 2]), in order to learn both the structure and the parameters of a network in case of a dataset containing missing values. The method provided is sem. It accepts as parameters an InferenceEngine and a BNDataset. Also this method can be controlled with thresholds for the structure learning and the parameter learning steps, namely struct.threshold and param.threshold.

The method accepts also all of the parameters available for the structure learning method (see section ??). However, the method is designed in order to follow as much as possible the settings of the original learning, whenever possible. For example, it is strongly recommended to adopt the same scoring function used in the learning of the network structure. If the network has been first learnt using bnstruct, the method will take care of that; in case of network read from a file, this is left to the user.

```
> dataset <- child() # or any other way to create a custom BNDataset
> net     <- BN(dataset)
> net     <- learn.structure(net, dataset)
> net     <- learn.params(net, dataset)
> inf.eng <- InferenceEngine(net)
> inf.eng <- sem(inf.eng, dataset)
> plot(updated.bn(inf.eng))
```

7 Other utilities

7.1 Sample data

We provide also methods for generating a sample of data, or a complete dataset.

The two methods for this purpose are the `sample.row` and `sample.dataset`, that generate, respectively, a vector of values and a `BNDataset` object. Both the methods accept as first argument a `BN` or an `InferenceEngine` object. To generate a `BNDataset` with `sample.dataset` one should also provide the number of observations to sample, via the `n` parameter.

```
> net <- BN(...)
> eng <- InferenceEngine(net)
> sample.row(net)
> sample.row(eng)
> sample.dataset(net,1000)
> sample.dataset(eng,10000)
```

References

[1] Nir Friedman. Learning belief networks in the presence of missing values and hidden variables. In *ICML*, volume 97, pages 125–133, 1997.

[2] Nir Friedman. The bayesian structural em algorithm. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pages 129–138. Morgan Kaufmann Publishers Inc., 1998.

[3] Nir Friedman, Moises Goldszmidt, and Abraham Wyner. Data analysis with bayesian networks: A bootstrap approach. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 196–205. Morgan Kaufmann Publishers Inc., 1999.

[4] Tomi Silander and Petri Myllymaki. A simple approach for finding the globally optimal bayesian network structure. *arXiv preprint arXiv:1206.6875*, 2012.

[5] Ioannis Tsamardinos, Laura E Brown, and Constantin F Aliferis. The max-min hill-climbing bayesian network structure learning algorithm. *Machine learning*, 65(1):31–78, 2006.