Inductive Inference 1.1.2: Inductive Programming and a Case Study of Bayesian Networks

Lloyd Allison

Technical Report 2005/177
School of Computer Science and Software Engineering,
Clayton School of Information Technology,
Monash University, Clayton, Victoria, Australia 3800.
tel:(+61) 3 9905 5205, lloyd.allison@infotech.monash.edu.au
21 September 2005.

Abstract. *Inductive programming* is a new machine learning paradigm combining functional programming (FP) for writing statistical models and the information theoretic criterion, *Minimum Message Length* (MML), to prevent overfitting. Type-classes specify general properties that statistical models must have. Many statistical models, estimators and operators have polymorphic types. Useful operators transform and combine models, and estimators, to form new ones. FP's *compositional* style of programming is a great advantage in this domain. MML fits well with FP in providing a compositional measure of the complexity of a model from its parts.
Inductive programming is illustrated by a case study of Bayesian networks. Networks are built from classification- (decision-) trees. Trees, and networks, are general [4] as a natural consequence of the method. Discrete and continuous variables, and missing values are handled. Trees are built from partitioning functions and models on dataspaces. Finally the Bayesian networks are applied to a challenging data set on lost persons.

Keywords: Bayesian networks, classification trees, decision trees, function models, inductive inference, machine learning, minimum message length, MML, MDL, regression, statistical models.

1 Introduction

The paper describes *inductive programming* (IP) a paradigm for quickly writing succinct solutions to inductive inference problems from machine learning. Solutions take the form of *statistical models* and their estimators: Given particular, invariably noisy, data infer a general model of the data. IP uses functional programming (FP) to program models and estimators, and the information theoretic criterion, minimum message length (MML), to prevent over-fitting.

Much research in machine learning involves devising a new kind of statistical model and implementing a program to learn (infer, fit, estimate) a model given data. Such stand-alone programs are often hard to modify and combine with others to implement new statistical models. To address this, IP defines types and classes of statistical models and the properties that instances, that is particular models, must have and provides a library of such instances.

Given the huge variety of computing problems, the chances of having a ready-made program that already solves a new problem is small. The situation is no different in inductive inference so it is useful to have a way of creating new solutions quickly and easily. Programming languages exist to make it easier to write new solutions. One could devise a special purpose language for inductive inference, and examples can be found, sometimes as a "scripting" language distinct from the "implementation" language in a data analysis platform such as R [25] and S-Plus [11]. But such a language is often interpreted and lacks compile-time type checking, and it is rarely maintained by programming language "professionals" who ensure that it grows and develops. Instead IP uses an existing general purpose FP language that is compiled and has a strong type system – Haskell [17]. Haskell is a good choice [5] for inductive inference because it is expressive and has a powerful system of polymorphic types and type classes; it is good language technology. FP encourages the *composition* of functions, and polymorphic types lead to general solutions; this all makes for short and general programs. We see these benefits rubbing-off on statistical models when they are transformed and composed.

Previous work on IP [2, 1, 4, 3, 5] created basic but useful statistical models, estimators and functions. This paper shows how they can be tailored quickly to suit a new problem and used as parts of a new model. Many models and associated functions are polymorphic; a good type and class system reveals their true generality. Statistical models, functions and data can be very general – any computable model inferred from almost any type of data by an arbitrary algorithm.

Over-fitting is a well known problem in machine learning and it is essential that a machine learning system do something about it. William of Occam argued in medieval times that an explanation should be kept simple unless necessity dictates otherwise. A computer program doing inductive inference must address model complexity. In particular, if sub-models are to be composed to make new models, the complexity of the parts and the whole must be dealt with. With its compositional nature, minimum message length (MML) (section 2.1) inference [27, 26] is a natural partner for FP in this domain.

The questions that are raised, and that are being answered as IP develops, include: what are the types and classes of statistical models, what can be done to models, and how can they be transformed and combined? Depending on one's background and inclination, IP can be seen as a software engineering analysis of machine learning, as a compositional denotational semantics of statistical models, as an application of FP, or as an embedded language [24].

The next section covers some background material. After that, inductive programming (IP) is illustrated by a case study of Bayesian networks. Bayesian networks are then applied to a data set of lost persons [15]. It is a challenging data set of 363 records and 15 variables, half of them missing on average. It shows the kind of problem that typically pops up with real data, if any data set can be said to be typical. Bayesian networks [4] form a case study; the main point of the paper is to show how a new statistical model can be programmed quickly to suit a new problem. All code shown is Haskell-98 in the interests of standardisation and has been compiled under the Glasgow Haskell Compiler, ghc, version 6.0.1.

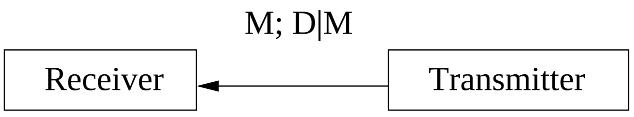

Fig. 1. Message Paradigm.

2 Background

For completeness, this section introduces MML and IP's main classes [2,3,5].

2.1 MML

Minimum message length (MML) inference [27,26] builds on Shannon's mathematical theory of communication [19], hence 'message', and on Bayes's theorem [7]:

```
Pr(M&D) = Pr(M).Pr(D|M) = Pr(D).Pr(M|D)
msgLen(E) = -log(Pr(E))
msgLen(M&D) = msgLen(M)+msgLen(D|M) = msgLen(D)+msgLen(M|D)
```

where M is a model (theory, hypothesis, parameter estimate) of prior probability Pr(M) over some data, D, and E is an event of probability Pr(E). MsgLen(E) is the length of a message, in an optimal code, announcing E; the units are *nits* for natural logs, *bits* for base 2 logs.

MML notionally considers a *transmitter* sending a two-part message to a *receiver* (figure 1). The first part, of length msgLen(M), states a model which is an answer to some inference problem. The second part, msgLen(D|M), states the data encoded as if the answer, M, is true; note that the receiver cannot decode the second part without the first part. There is a trade-off between the complexity of the model, M, and its fit to the data, D|M. In some simple cases MML is equivalent to maximum aposteriori (MAP) estimation but this is not true in general [28,12]; for example if one or more continuous parameters are involved they must be stated to *finite*, optimal precision. Strict MML (SMML) relies on the design of a full optimal code book. Unfortunately SMML is infeasible for most inference problems [12, 26]. Fortunately there are efficient, accurate MML approximations [28, 26] for many useful problems and models.

MML is a *compositional* criterion because the complexity of data, models and sub-models are all measured in the same units: "[It is possible] to use [message] length to select among competing sub-theories at some low level of abstraction, which in turn can form the basis (i.e., the 'data') for theories at a higher level of abstraction. There is no guarantee that such an approach will lead to the best global theory, but it is reasonable to expect in most natural domains that the resulting global theory will at least be near-optimal" [29, 14]. MML's compositional nature is a good fit with functional programming's compositional style of programming. This is illustrated in the Bayesian network case study of section 3. MML has been used to assess the complexity of some specific kinds of combined models (e.g. [14, 6, 18]) but its general *programming* potential has only recently started to be studied [2, 4, 5]. A functional language with a parametric polymorphic type system is a sound foundation for such a study.

```
class ... SuperModel sMdl where
 prior   :: sMdl -> Probability
 msg1    :: sMdl -> MessageLength
 mixture :: ... mx sMdl -> sMdl
 ...

class Model mdl where
 pr   :: (mdl dataSpace) -> dataSpace -> Probability
 nlPr :: (mdl dataSpace) -> dataSpace -> MessageLength
 msg  :: ... (mdl dataSpace) -> [dataSpace] -> MessageLength
 msg2 :: (mdl dataSpace) -> [dataSpace] -> MessageLength
 ...

class FunctionModel fm where
 condModel :: (fm inSpace opSpace) -> inSpace -> ModelType opSpace
 ...

class TimeSeries tsm where
  predictors :: (tsm dataSpace) -> [dataSpace] -> [ModelType dataSpace]
  ...
```

'`...`' stands for omitted details, '`::`' for 'has type',
'`[t]`' for 'list of a type `t`', and '`->`' for function type.

Fig. 2. Classes of Statistical Model.

2.2 Types and Classes of Statistical Models

We want to be able to program as large as possible a set of things that people call *statistical models*, i.e. that assign probabilities to data, and yet have the set clean, orthogonal and built on a small foundation. For simplicity, we also want a small set of just their essential properties. Haskell type classes (figures 2 and 3) were previously defined [2, 5] for basic models (distributions), function-models (regressions) and time-series models; the last are not used here. A basic `Model`, `mdl`, can return the probability, `pr`, and the negative log probability, `nlPr`, of a datum from its data-space. It can also compute the second-part, `msg2`, and the total message length, `msg`, for a data set. We are only concerned with its most important properties here; a statistical model might be able to do other things.

A function-model has an input-space (exogenous variables) and an output space (endogenous variables). Its principal ability is to return a model of its output space conditional, `condModel`, on a value from the input space.

A super-class, `SuperModel`, states that an instance of one of the various subclasses must return its own prior probability and message length, `msg1`, and that it must be able to form mixtures; it must also be in class `Show` so that we can see the answers to inference problems.

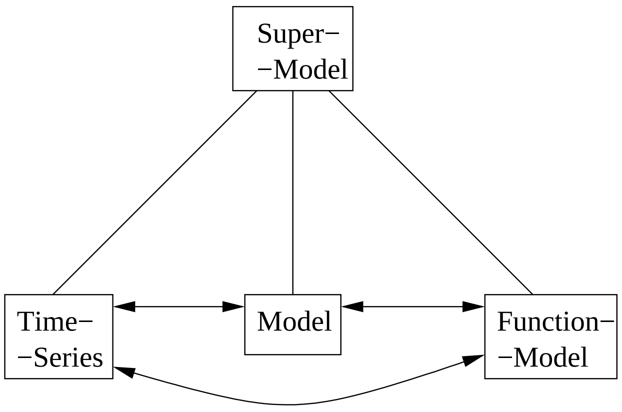

Fig. 3. Classes and Conversions.

Types are provided for models to be built in standard ways: Type `ModelType` is an instance of type-class `Model`, and types `FunctionModelType` and `CTreeType` (classification tree type) are instances of type-class `FunctionModel`.

Operators are defined to implement familiar laws of probability. For example, assuming that variables over the data-spaces `ds1` and `ds2` are independent, `bivariate m1 m2` forms a model of the product data-space, `(ds1, ds2)`, from `m1`, a model of `ds1`, and `m2`, a model of `ds2`. For the case where `ds2` is conditionally dependent on `ds1`, `condition m1 fm` forms a model of `(ds1, ds2)`, from a model of `ds1` and a function-model, `fm`, from `ds1` to `ds2`. There are corresponding operators on estimators – `estBivariate`, `estCondition` and so on. Many of these operators are polymorphic in that their types contain type variables such as `ds1` and `ds2`.

Useful statistical models, including multi-state, normal and multivariate distributions, mixture models, Markov models, finite function-models (conditional probability tables) and classification trees, have been defined and made instances of the appropriate classes, and conversion functions (figure 3) defined on them [2, 5]. Below, these building blocks are extended, tested and used in a case study of Bayesian networks to explore and illustrate IP.

3 Case-Study: Bayesian Networks

A Bayesian network [16] is a good tool to investigate relationships among the variables of a data set. A Bayesian network is a directed acyclic graph (DAG). A node represents a variable. An edge represents a (direct) conditional dependence of a *child* on a *parent* and, in a suitable context, has a causal interpretation. Figure 4 shows a small example network [4] in which variable @2 is a child of variables @0 and @1, and is a parent of variable @4; variable @3 is independent of the other variables. It happens that variables @0 and @4 are continuous and that @1, @2 and @3 are discrete.

Creating and applying an estimator for Bayesian networks [4] forms our case study to illustrate IP. A Bayesian network is in class `Model` (section 2.2) and can assign a probability to a data tuple; belief updating has not yet been implemented.

Friedman and Goldszmidt [13] first suggested using classification-trees (decision trees), in place of the full conditional probability tables (CPTs) often used within the nodes of Bayesian networks. Comley and Dowe [9] have also used trees within the nodes of networks. A classification tree can "become" a full CPT in the limit but can be much more economical, that is less complex, in many cases. It happens that previous work included a general classification tree algorithm [2, 1, 3, 5]. The classification tree type is an instance of function-model. A tree can test discrete and continuous variables from its input space and can have discrete or continuous distributions, or even function-models (regressions), in the leaves. It is reused as the basis of our general Bayesian networks [4]; also see section 3.7. Each classification tree consists of at least one leaf-node, `CTleaf`, and possibly also fork-nodes, `CTfork`. These leaf- and fork-nodes are *not to be confused* with network nodes; there is one tree per network node. The classification tree type is an instance of the class `FunctionModel`. A fork tests a parent (input) variable value. A leaf models the appropriate child (output) variable. The multi-state distribution, `mState`, models a discrete variable, and the normal distribution models a continuous variable; other distributions can be used if necessary because the tree estimator is parameterised by the leaf estimator. MML gives a trade-off between the complexity of a tree and its fit to the data which is used to control the search. Note that one or more tests on a parent variable in a tree indicate a parent-child dependency, an edge, at the network level.

The following sections describe the application of Bayesian networks to lost person data. As an example of IP we see the composition of statistical models: Multi-state and normal distributions within models of missing data within

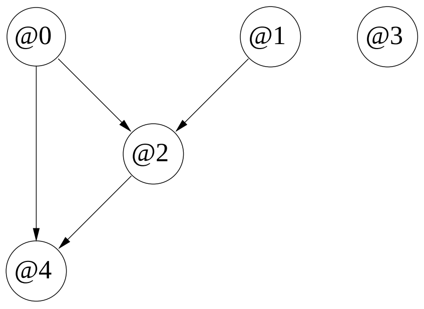

Fig. 4. Example Bayesian Network.

classification trees within a Bayesian network. Some new generic features were required to handle this data set. Any of those features may exist in some other data analysis platform, perhaps this is true of all of them, but it is unlikely that they all exist in the same platform, and unlikely that such a platform could be as easily adapted to further new features. It often seems that every data set has its own oddities as one gets to know it.

3.1 Application of Bayesian Networks: Lost Person

Koester's [15] lost person data set has been studied in CSSE, Monash [22, 23]. There are 363 records, and 15 variables, numbered 0-14; half of the values are missing on average. Attention is sometimes restricted to the first eight variables; one aim is to predict distance travelled, `DistIPP` variable 7, from variables zero to six.

3.2 Describing the Data

The first step is to define the variable types in the lost person data set; in Haskell this is:

```
data Tipe = Alzheimers| Child| Despondent|
            Hiker| Other| Retarded| Psychotic deriving ...
type Age  = Double
data Race = ...
data Gender = ...
data Topography = Mountains| Piedmont| Tidewater deriving (Ord, Enum,...)
data Urban = Rural | Suburban | Urban deriving (Ord, Enum, ...)
type HrsNt   = Double                                   -- hours notified
type DistIPP = Double                                   -- distance
 ...
type MissingPerson = (Maybe Tipe, Maybe Age, ...)
```

'`deriving`' directs the compiler to add a new data type to standard Haskell classes such as `Ord` (ordered) and `Enum` (enumerated). Missing values are an issue and are represented by `Maybe t` where `Maybe t = Nothing | Just t` is a standard Haskell type with parameter t; see section 3.6. Haskell's standard Prelude [17] instantiates tuples, up to 7-tuples, in classes `Read` and `Show`, so the 15-tuples here need to be made instances of those classes for input and output respectively. This is an easy, if tedious, job and could be automated in one of the Haskell extensions such as template Haskell [20], say.

3.3 Partitioning Data Spaces: Class Splits

A classification tree, as used in a network node, operates by *splitting*, that is partitioning, a data set from its input space by tests on input variables; a `Splitter` partitions a data set. In this way the data are directed into subtrees and eventually into leaves where the output variable(s) can be well modelled. Function `splits` of class `Splits` [2, 1] proposes, in order of decreasing prior probability, `Splitters` for use by the classification tree estimator, `estCTree`.

```
class Splits ds where splits :: [ds] -> [Splitter ds]
```

The current tree estimator [2, 1] uses a simple zero-lookahead algorithm in the search to balance tree complexity (`msg1`) against fit to the data (`msg2`). A continuous, ordered (`Ord`), variable, such as `Age`, is split on being $<$ or $\geq$ some value; by default `splitsOrd` proposes values as follows: Median, quartiles, octiles, and so on [30]. A discrete, i.e. `Bounded`, enumerated (`Enum`), variable, such as `Gender`, of a k-valued type is conventionally split into k parts, as defined in the obvious way by function `splitsBE`. However `Topography` and `Urban` are instances of the standard Haskell classes `Bounded`, `Enum` (enumerated) *and* `Ord` (ordered), so we also have the options of splitting each of them into two parts on the basis of order, as covered by `splitsOrd`:

```
instance Splits Tipe       where splits = splitsBE
 ...
instance Splits Topography where splits = splitsBE  --or splitsOrd
instance Splits Urban      where splits = splitsBE  --or splitsOrd
```

Yet another option was implemented and tested: `Tipe` has seven values and high-arity discrete types like `Tipe` can cause function-models difficulties because of the large number of options and the few data in some or all of them. If some options are thought to behave in similar ways then, rather than using `splitsBE`, values can be grouped into nominated sub-sets (and their complement) accordingly. This only affects splitting on `Tipe`, not modelling of it. A function to implement this `setSplits` option is just five lines.

```
setSplits sets [] = []
setSplits sets xs =
  let y:ys = map (memberships sets) xs
  in if all ((==) y) ys then []
     else [setSplitter sets]
```

```
instance Splits Tipe where
  splits = setSplits [[Alzheimers],[Child]]  -- e.g.
```

If ‘k’ subsets are specified, their complement is taken to be the (k+1)st. Note that the programmer decides how to group the values in `Tipe`. In principle a function could cost and search through the grouping possibilities but this would, of course, increase the overall search time.

3.4 Modelling the Variables

The question of which distribution, and therefore which estimator, to use for each variable now arises. The standard estimator for the normal (Gaussian) distribution uses a uniform prior on the mean and an inverse prior on the standard deviation and requires their ranges, and also the data measurement accuracy. Note, the multi-state distribution and its estimator are polymorphic.

```
e0 = (estModelMaybe estMultiState)              -- Tipe
e1 = (estModelMaybe (estNormal 0 90 1 70 0.5))  -- Age
 ...
```

Function `estModelMaybe` was quickly created to allow for missing values in a variable; it is discussed in section 3.6.

Finally the individual estimators are assembled into `estMissingPerson`, a composite that matches a data tuple.

```
estMissingPerson =
  estVariate15 e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 e11 e12 e13 e14
```

Function `estVariate15` estimates a 15-variate probability distribution which is an instance of class `Project` – a piece of inductive inference machinery described next.

3.5 Selecting Sub-Spaces: Projections

In a typical application of a classification tree [2] the input variables and the output variable (usually singular) are fixed. But here, in a Bayesian network [4], the selection of parent (input) and child (output) variables must be under program control. Class `Project`, as in *projection*, was created for this purpose. Some such mechanism is needed for heterogeneous variable types in a strongly typed language; the network estimator does not “care” what types the data and sub-estimators have provided that they agree. An instance, `t`, of class `Project` is some multi-dimensional type for which a list of Boolean flags can be used to restrict `t`’s activities to certain selected dimensions. The non-selected dimensions behave in a trivial, identity manner, that is appropriate to type t – if they are ever called upon. In the case of a `Model` this is to return a message length of zero for non-selected variables, that is they are taken to be already known, or to be of no interest.

```
class Project t where
  select :: [Bool] -> t -> t
  ...
```

As discussed in section 3.3, class `Splits` exists for partitioning data – discrete, continuous or multi-variate. A class `Splits2`, inspired by `Project`, was defined (it could perhaps be folded into class `Splits`) to allow splitting on selected variables:

```
class Splits2 t where
  splitSelect :: [Bool] -> [t] -> [Splitter t]
```

The situation for adding k-ary types to class `Project`, or k-tuples to `Splits2`, is similar to that for k-tuples with respect to classes `Read` and `Show` as discussed in section (3.2).

3.6 Handling Missing Data

The lost person data set is challenging in having many missing values. Most data have at least one missing value, and some have several. Every variable is missing in some datum. Haskell has the ideal type to represent possibly missing values: `Maybe`. New operators were needed to extend arbitrary statistical models to cover possibly missing values. Rather than imputing (replacing) missing values the phenomenon is built into our model.

Function `modelMaybe m1 m2` might be called a "high-order" function on models. because it acts on models which are, if not literally functions, principally made up of them. It turns an *arbitrary* model, `m2`, of non-missing data of type `t` into the corresponding model of `Maybe t`. It requires a model, `m1`, of `Bool`, for whether the data is present (True) or missing (False).

```
modelMaybe m1 m2 =
 let
  negLogPr (Just x) = (nlPr m1 True) + (nlPr m2 x)
  negLogPr Nothing  =  nlPr m1 False
 in MnlPr (msg1 m1 + msg1 m2) negLogPr ...show method omitted
```

`MnlPr` is a constructor for a type in class `Model` (section 2.2); it takes a message length, a negative log likelihood function, and a description which shows the model.

The related function, `estModelMaybe` acts on estimators; it turns an estimator of non-missing data into the corresponding estimator where the data may include missing values:

```
estModelMaybe estModel dataSet =
 let present (Just _) = True
     present Nothing  = False
     m1 = uniformModelOfBool
     m2 = estModel (map (\(Just x)->x) (filter present dataSet))
 in modelMaybe m1 m2
```

In the present application the missing-ness of values is certainly non-random for some variables, for example `Age` is often unrecorded for cases of `Hiker::Tipe`. However, we are not interested in modelling missing-ness so a fixed unbiased model, `m1`, is used above to "predict" missing (`Nothing`) or present (`Just`...). The following definition can be used if it is necessary to estimate missing-ness:

```
m1 = estMultiState (map present dataSet)
```

Missing values also affect splits, that is partitions of the data (section 3.3). A simple strategy is for the variable to be split as for the underlying type but with an extra option for missing (`Nothing`) cases:

```
maybeSplitter (Splitter n f d) =
 let f' Nothing  = n
     f' (Just x) = f x  -- Just x, as x was, 0..n-1
 in Splitter (n+1) f' ...show method omitted
```

Other strategies, not examined here, could try to predict in various ways what the missing value, or its distribution, really is and act on that. There are a great many possibilities and, this being an example, we just give one reasonable, simple approach.

3.7 Mixed Bayesian Networks and the Lost Person Network

The function, `estNetwork`, for inferring a Bayesian network is given a permutation, a total ordering, of the selected variables that are to be considered; a variable may be dependent on none, some or all of the variables preceding it in the permutation. The use of total or partial orders on variables is not uncommon in network learners [16]. It would be possible to search over permutations, heuristically if there were many variables, and MML does suggest various heuristics. However the present application itself suggests likely orderings of the variables, the simple algorithm accepts one of these and that permutation is currently taken to be common knowledge:

```
estNetwork perm estMissingPerson dataSet =
 let n = (length . selAll) (estMV [])
     search _  []     = []  -- done
     search ps (c:cs) =  -- parents ps, children c:cs
      let opFlag  = ints2flags [c] n  --identify child
          ipFlags = ints2flags ps  n  --possible parents
          cTree = estCTree (estAndSelect estMV opFlag)
                           (splitSelect ipFlags)
                           dataSet dataSet
      in cTree : (search (c:ps) cs)
     trees     = search [] perm
     msgLen    = sum (map msg1 trees)  -- total msg1
     nlP datum =
       sum (map (\t -> condNlPr t datum datum) trees)
 in MnlPr msgLen nlP (\() -> "Net:"++(show trees))
```

Internally `estNetwork` uses the estimator for classification trees [2, 5], `estCTree`, to do much of the work. The remainder consists of organising selector flags corresponding to allowed parents for the child in the current node. Note that the `dataSet` seems to be passed to `estCTree` twice – as both input and output variables, but straightforward auxiliary functions `ints2flags`, `estAndSelect` and `splitSelect`, use the flags to cause the child (output) to be predicted by the estimator and the parents (input) to be split on as appropriate at each node in the network.

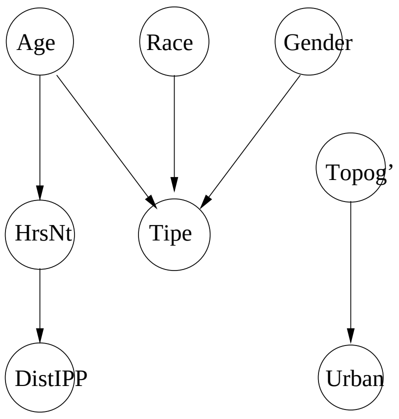

Fig. 5. Lost Person Network 1 (see 3.2, 3.7 for details).

For lost persons, variables 1 to 3, `Age`, `Race` and `Gender` cannot, in a causal sense, depend on other variables and should come first, in some arbitrary order, say [1,2,3]. `Tipe` possibly depends on them, for example there are few young Alzheimers cases. `Topography` and `Urban` can sensibly come next, and one expects a relationship between them. That leaves `HrsNt` and finally `DistIPP` to make up a plausible ordering, [1, 2, 3, 0, 4, 5, 6, 7], of the first eight variables. There is also a natural null hypothesis which models the variables independently. The code to run the inference is shown below:

```
dataSet = read (readFile theDataFile) :: [MissingPerson]     -- input
nw = estNetwork [1,2,3,0,4,5,6,7] estMissingPerson dataSet   -- infer BN
nullModel = estMissingPerson dataSet                         -- null H
```

Figure 5 shows the first network inferred for variables 0 to 7. an abridged node listing is given in figure 6. `Tipe` depends strongly on `Age` and also on `Gender` and `Race`. As expected, `Urban` is dependent on `Topography`. There is some direct dependence of `DistIPP` on `HrsNt`, and of the latter on `Age`, but there seems to be no strong predictor of `DistIPP` from other variables. The model is significant with a total two-part message length, for the first eight variables, of 5512 nits against 5936 nits under the null model. Other analyses were tried, for example

```
Net:[
-- @1, Age:
{CTleaf _,(Maybe 50:50,N(40.6,27.5)(+-0.5)),...},

-- @2, Race:
{CTleaf _,_,(Maybe 50:50,mState[0.66,0.34]),...},

-- @3, Gender:
{CTleaf _,_,_,(Maybe 50:50,mState[0.72,0.28]),...},

-- @0, Tipe:
{CTfork @1(<|>=19.0|?)[ ...uses @1, @2, @3... ]},

-- @4, Topography:
{CTleaf ..,(Maybe 50:50,mState[0.17,0.52,0.31]),..},

-- @5, Urban:
{CTfork @4(=Mountains..Tidewater|?)[
 {CTleaf..,(Maybe 50:50,mState[0.93,0.04,0.04]),..},
 {CTleaf..,(Maybe 50:50,mState[0.70,0.19,0.11]),..},
 {CTleaf..,(Maybe 50:50,mState[0.38,0.02,0.6 ]),..},
 {CTleaf..,(Maybe 50:50,mState[0.73,0.2 ,0.07]),..}
]},

-- @6, HrsNt:
{CTfork @1(<|>=62.0|?)[
 {CTleaf ...,(Maybe 50:50,N( 8.7, 7.6)(+-0.5)),..},
 {CTleaf ...,(Maybe 50:50,N(21.4,26.3)(+-0.5)),..},
 {CTleaf ...,(Maybe 50:50,N(20.0,...1-case...),..}
]},

-- @7, DistIPP:
{CTfork @6(<|>=1.0|?)[
 {CTleaf ...,(Maybe 50:50,N( ...no-cases... ),...},
 {CTleaf ...,(Maybe 50:50,N(0.59,0.6)(+-0.2)),...},
 {CTleaf ...,(Maybe 50:50,N(1.52,2.8)(+-0.2)),...}
]}]

network: 115.1 nits, data: 5396.6 nits
null:   5935.6 nits (@0..@7)
```

Fig. 6. Trees in the Nodes of the Lost Person Network 1.

allowing ordered (`Ord`) splits on `Topography` and `Urban`, in place of `Bounded Enum` splits; the conclusions were broadly similar.

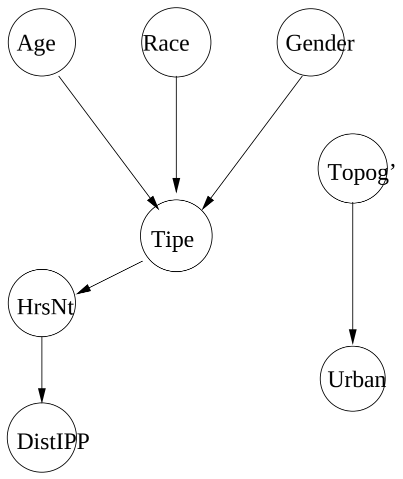

Fig. 7. Lost Person Network 2.

When `Tipe` was allowed to split according to `setSplits [[Alzheimers], [Child]]` (section 3.3), the implicit complement being `[Despondent...]`, the link from `Age` to `HrsNt` was replaced by a link from `Tipe` (which itself depends strongly on `Age`) for a saving of 6 nits on the model against a loss of 3.7 nits on the data (figure 7). However this small net gain should be taken with a big pinch of salt and may well be due to the pattern of missing data as much as anything.

As a final example, modelling all 15 variables gave the network shown in figure 8; Tipe has been duplicated for ease of drawing. The extra variables, 8 to 14, are: `TrackOffset` (continuous); `Health = Well | Hurt | Dead`; `Outcome = Find | Suspended | ... | Invalid`; `FindRes = Ground | Air | ... | Dog`; `FindLoc = Brush | Woods | ...`; `HrsFind` (continuous); `HrsTo` (continuous).

3.8 Discussion

Weka [31] which is based on Java is perhaps the system closest to the present work. Weka's Bayesian networks "assume that all variables are discrete" [8] p.22 and "a limitation of the current classes is that they assume that there are no missing values" [8] p.23. In Weka, continuous variables must be discretised first and the way this is done may affect the outcome. This is unnecessary for modelling and, for splitting, is part of the network optimisation when using our classification trees (section 3.2). Missing-ness was built into our models (section 3.6).

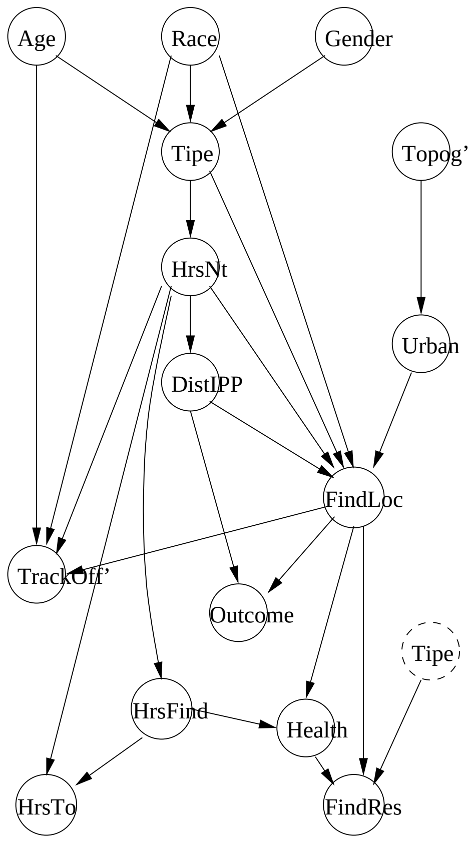

Fig. 8. Lost Person Network for all 15 Variables.

There are distant similarities between IP and inductive logic programming (ILP): There has been some interest in the use of complexity-based measures in ILP [10, 21] but this aspect of ILP is less developed than work on MML. The programmer is involved in design of the search algorithm in IP to a greater extent than in ILP, typically in designing new models and estimators; it is infeasible to have a very general search over too large a class of computable statistical model.

A model in IP, particularly one that is used as a component of other models (figure 9), must be able to handle extreme data sets. For example a Bayesian network may contain several trees and each tree may contain several leaf distributions. One or more of those leaf distributions may be given a sub-set of data that is "unusual" – perhaps consisting of just a single item. MML insists that every model effects a valid, decodable message (in principle) so there can be no understating of a model's complexity. A (sub-) model must guarantee this, or at the very least raise an exception if it cannot. This principle keeps us "honest" and ensures that the top-level model's complexity is valid.

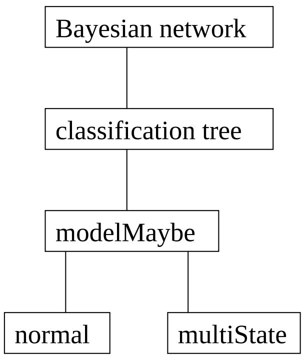

Fig. 9. Model Layers.

4 Conclusions

Inductive programming (IP) uses the compositional abilities of functional programming, Haskell and MML. Haskell's features have a number of advantages in inductive inference. Mapping a data set, such as lost persons, onto the Haskell type system is a useful exercise in getting to know the data, very precisely; a person doing data analysis will inhabit this space for some time. The need to define a variable's properties, e.g. `Ord` or not, automatically suggests what is possible, such as to split `Topography` as discrete or as ordered (section 3.2). These things cannot be forgotten; the type and class system brings them to your attention.

The IP code shown is standard Haskell-98 but other experiments [4] do show that some Haskell type extensions can be useful in some other problems. In-built support for wide tuples, `(,)`, would make it easier to deal with large multi-variate data sets, although template Haskell is a possible solution.

High-order functions, such as `estModelMaybe` (section 3.6), are invaluable in creating new ways of using *arbitrary* statistical models. The polymorphic type system ensures that the uses are both general and type safe. Haskell's type inference algorithm often finds a more general type for a function than its programmer did and this is also the case with statistical models and estimators. There is potential for an extensive library of operators on statistical models and their estimators.

Lazy evaluation means, for example, that only models of selected variables of lost persons (section 3.2) are evaluated. Selections are made once at the top level, most of the algorithms do not "consider" the matter at all.

Computing model complexity by minimum message length (section 2.1) is a good match with the compositional style of functional programming. The reader may hardly have noticed any explicit Message length calculations are handled by `modelMaybe` (section 3.6) and other functions, and are combined in the complexity of the Bayesian network (section 3.1, figure 9) and its classification trees (figure 8) to inform the search.

A specific model can be created quickly for a new problem thanks to Haskell's expressive power. Of course it cannot yet be claimed that the types and classes created are the best possible designs for a compositional denotational semantics of for statistical models. For example, a case can be made for specifying the notion of a *data set*; perhaps data traversal, data measurement accuracy and data weights should be wrapped up in suitable types and classes. Only more experience, and time, will let us settle on the best trade-off between generality, usability and efficiency, but experience to date is positive.

The Bayesian network estimator, `estNetwork`, and associated classes `Project` and `Splits2` [4] (section 3.5) took one day to create. The lost person (section 3.2) application came along some time later and it took one and a half days to create a working model, *including* how to handle missing data (section 3.6) which had previously been in the 'must think about that one day' category. Any amount of further time can be spent playing with the data once a model and a program exist, although there is a fine line between data exploration and fishing.

Acknowledgments. I want to thank Charles Twardy, Ann Nicholson and Kevin Korb for generous discussions of Bayesian networks. Charles Twardy also discussed missing people and he coined the term 'inductive programming'. Chris Wallace (1933–2004) always gave much help and was an inspiration; he is sadly missed. The Department of Computer Science at the University of Wales Aberystwyth, and the Department of Computer Science and the Programming Language and Systems Research Group at the University of York were very hospitable during my visits in 2004.
[A paper based on this work has since appeared as:
L. Allison, *A Programming Paradigm for Machine Learning, with a Case Study of Bayesian Networks*, ACSC2006, pp. 103-111, January 2006.]

References

1. L. Allison. Inductive inference 1. Technical Report 2003/148, School of Computer Science and Software Engineering, Monash University, December 2003.
2. L. Allison. Types and classes of machine learning and data mining. *26th Australasian Computer Science Conference (ACSC), Adelaide*, pages 207–215, February 2003.
3. L. Allison. The types of models. *2nd Hawaii Int. Conf. Statistics and Related Fields (HICS-2)*, June 2003.
4. L. Allison. Inductive inference 1.1. Technical Report 2004/153, School of Computer Science and Software Engineering, Monash University, May 2004.
5. L. Allison. Models for machine learning and data mining in functional programming. *J. Functional Programming*, pages 15–32, January 2005. Online 7/2004. doi:10.1017/S0956796804005301.
6. L. Allison, D. Powell, and T. I. Dix. Compression and approximate matching. *BCS Computer J.*, 42(1):1–10, 1999.
7. T. Bayes. An essay towards solving a problem in the doctrine of chances. *Phil. Trans. of the Royal Soc. of London*, 53:370–418, 1763. (Reprinted in *Biometrika* 45(3/4), pp. 296–315, 1958).

8. R. R. Bouckaert. Bayesian networks in Weka. Technical Report 14/2004, Comp. Sci. Dept.. U. of Waikato, September 2004.
9. J. Comley and D. Dowe. General Bayesian networks and asymmetric languages. *2nd Hawaii Int. Conf. Statistics and Related Fields (HICS-2)*, June 2003.
10. D. Conklin and I. H. Witten. Complexity-based induction. *Machine Learning*, 16(3):203–225, 1994.
11. M. J. Crawley. *Statistical Computing - an Introduction to Data Analysis using S-Plus.* Wiley, 2002.
12. G. E. Farr and C. S. Wallace. The complexity of strict minimum message length inference. *BCS Computer J.*, 45(3):285–292, 2002.
13. N. Friedman and M. Goldszmidt. Learning Bayesian networks with local structure. *UAI'96*, pages 252–262, 1996.
14. M. P. Georgeff and C. S. Wallace. A general selection criterion for inductive inference. *European Conf. on Artificial Intelligence (ECAI84), Pisa*, pages 473–482, September 1984.
15. R. J. Koester. Virginia dataset on lost-person behaviour. 2001. Author's site http://www.dbs-sar.com/.
16. K. B. Korb and A. E. Nicholson. *Bayesian Artificial Intelligence.* Chapman and Hall / CRC, 2004.
17. S. Peyton-Jones et al. Report on the programming language Haskell-98. 1999. Available at http://www.haskell.org/.
18. D. R. Powell, L. Allison, and T. I. Dix. Modelling alignment for non-random sequences. *Proc. 17th ACS Australian Joint Conf. on Artificial Intelligence (AI2004)*, pages 203–214, 2004. LNCS/LNAI vol.3339.
19. C. E. Shannon. A mathematical theory of communication. *Bell Syst. Technical Jrnl.*, 27:379–423 and 623–656, 1948.
20. T. Sheard and S. Peyton-Jones. Template meta-programming for Haskell. *Proc. of the workshop on Haskell*, pages 1–16, 2002.
21. A. Srinivasan, S. Muggleton, and M. Bain. The justification of logical theories based on data compression. *Machine Intelligence*, 13:87–121, 1994.
22. C. R. Twardy. Sar*bayes*: Predicting lost person behavior. 2002. Presented to National Association of Search and Rescue (NASAR 2002), Charlotte, NC (Available at http://sarbayes.org/nasar.pdf).
23. C. R. Twardy and L. Hope. Missing data on missing persons. *personnal communication*, 2004.
24. A. van Deursen, P. Lint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.
25. Various. Cran: The comprehensive R archive network. *http://lib.stat.cmu.edu/R/CRAN/*, 2004.
26. C. S. Wallace. *Statistical and Inductive Inference by Minimum Message Length.* Springer-Verlag, 2005. isbn:038723795X.
27. C. S. Wallace and D. M. Boulton. An information measure for classification. *BCS Comput. J.*, 11(2):185–194, 1968.
28. C. S. Wallace and P. R. Freeman. Estimation and inference by compact coding. *J. Royal Statistical Society series B.*, 49(3):240–265, 1987.
29. C. S. Wallace and M. P. Georgeff. A general objective for inductive inference. Technical Report 32, Department of Computer Science, Monash University, 1983.
30. C. S. Wallace and J. D. Patrick. Coding decision trees. *Machine Learning*, 11:7–22, 1993.

31. I. H. Witten and E. Frank. Nuts and bolts: Machine learning algorithms in Java. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.*, pages 265–320, 1999.