# A Programming Paradigm for Machine Learning, with a Case Study of Bayesian Networks

**Lloyd Allison**

Clayton School of Information Technology,
Monash University,
Victoria,
Australia 3800.
tel:(+61) 3 9905 5205,
Email: `Lloyd.Allison@infotech.monash.edu.au`
`http://www.csse.monash.edu.au/~lloyd/tildeFP/II/`

## Abstract

Inductive programming is a new machine learning paradigm which combines functional programming for writing statistical models and information theory to prevent overfitting. Type-classes specify general properties that models must have. Many statistical models, estimators and operators have polymorphic types. Useful operators combine models, and estimators, to form new ones; Functional programming's compositional style of programming is a great advantage in this domain. Complementing this, information theory provides 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 are built from partitioning functions and models on data-spaces. Trees, and hence networks, are general as a natural consequence of the method. Discrete and continuous variables, and missing values are handled by the networks. Finally the Bayesian networks are applied to a challenging data set on lost persons.

Keywords: inductive inference, functional programming, Haskell, minimum length encoding, statistical models, Bayesian networks.

## 1 Introduction

The paper describes *inductive programming* (IP) a new 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 data, infer a *general* model from the data; the data are invariably noisy. IP uses functional programming 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. The resulting stand-alone programs are often hard to modify and to 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 problems in general-purpose computing, the chances of having a ready-made program that already solves some new problem is small. Things are 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 in general computing. One could devise a special purpose programming language for inductive inference and examples exist, sometimes as a "scripting" language as distinct from the main "implementation" language in a data analysis platform such as R (CRAN 2004) and S-Plus (Crawley 2002). But such scripting languages are often interpreted and lack compile-time type checking. Instead IP uses an existing general purpose functional programming language that is compiled and has a strong type system – Haskell (Peyton-Jones et al 1999). Haskell is a good choice (Allison 2005) for the domain because it is expressive and has a powerful system of polymorphic types and type-classes; it is good programming language technology. Functional programming encourages the *composition* of functions, and polymorphic types lead to general solutions; this 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 (Allison 2003) created basic but useful statistical models, estimators and functions. The present paper shows how they can be extended, composed and 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 and functions on them 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. William of Occam argued long ago that an explanation should be kept simple unless necessity dictates otherwise. A computer program doing inductive inference must address model complexity in some way. 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. Later we will see basic models used within models of missing data which are used within classification trees which in turn are used within Bayesian networks. With its compositional nature, minimum message length (MML) (section 2.1) inference (Wallace & Boulton 1968, Wallace 2005) is a natural partner for functional programming in machine learning.

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 them, and how can they be transformed and

M; D|M

Receiver ← Transmitter

Figure 1: Message Paradigm



Figure 2: Classes and Conversions

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 functional programming, or as an embedded language (van Deursen, Lint & Visser 2000). The Haskell code produced could also be seen a rapid prototype for other data analysis platforms.

The next section covers background material. After that inductive programming (IP) is illustrated by a case study of Bayesian networks. The Bayesian networks are then applied to a data set of lost persons (Koester 2001). 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 form a case study; the main aim of the paper is to show how a new statistical model can be programmed quickly to suit a new problem. It explores IP's expressiveness not the statistical performance of any particular model(s). If IP and some other system have equivalent models then those models will, in principle, behave roughly equivalently. Rather the point is to show how IP can be used to create a *new* model to suit a new inference problem, as opposed to "massaging" the problem to suit some existing model.

All code shown is Haskell-98 (Peyton-Jones et al 1999) in the interests of standardization and has been compiled under the Glasgow Haskell Compiler, ghc, version 6.0.1.

## 2 Background

For completeness, this section briefly introduces MML and IP's main type-classes.

### 2.1 MML

Minimum message length (MML) (Wallace & Boulton 1968, Wallace 2005) builds on Shannon's mathematical theory of communication (1948), hence 'message', and on Bayes's theorem (Bayes 1763):

```
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,
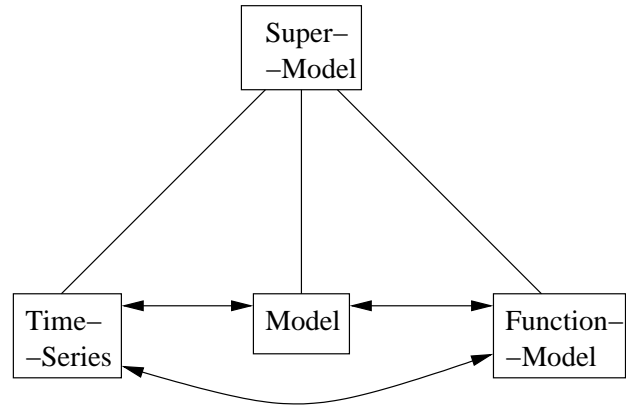
and its fit to the data, D|M: A simple model is cheap to state but may not fit the data well. A complex model may fit the data better but is more expensive to state (Georgeff & Wallace 1984). In some simple cases MML is equivalent to maximum aposteriori (MAP) estimation but this is not true in general (Wallace & Freeman 1987, Farr & Wallace 2002). For example, if one or more continuous parameters are involved they must be stated to *finite*, optimal precision, and MML shows how to do this. Note that a one-part message can be very slightly more efficient in transmitting D but it does not offer an *explanation* of D; it does not state an answer to an inference question.

MML (Wallace & Boulton 1968) is related to the minimum description length (MDL) principle (Rissanen 1978). The former aims to select a parameterized model – the parameters being stated to optimal precision – and embraces explicit priors. The latter aims to select a model-class and favours universal distributions and implicit priors. MML and MDL have been featured in the Journal of the Royal Statistical Society (Wallace & Freeman 1987, Rissanen 1987) and in a special issue of the Computer Journal on Kolmogorov complexity (Gammerman & Vovk 1999). Oliver and Baxter (1994, p. 24) made a direct comparison and concluded that only MML (Wallace & Freeman 1987) had all the desirable properties of invariance under non-linear transformations of parameters, of applicability to large and small samples (not only asymptotic), and of making a definite inference.

Strict MML (SMML) relies on the design of a full optimal code book. Unfortunately SMML is infeasible for most inference problems (Farr & Wallace 2002). Fortunately there are efficient, accurate MML approximations for many useful problems and models (Wallace 2005).

MML is a natural *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 subtheories 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" (Wallace & Georgeff 1983). 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 combined models of some specific types (e.g. Allison et al (1999) and Powell et al (2004))

```
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
 ...
```

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

Figure 3: Classes of Statistical Model

but its full *programming* potential has only recently started to be studied (Allison 2003). A functional programming language with a parametric polymorphic type system is a sound foundation for such a study.

## 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* and yet have the set clean, orthogonal and built on a small foundation. For simplicity, we also want a small collection of just their essential properties. Here statistical model is taken to cover things that assign explicit probabilities to data. Haskell type-classes (figures 2, 3) Model, FunctionModel and TimeSeries were previously defined (Allison 2003, Allison 2005) for basic models (distributions), function-models (regressions) and time-series models; the first two are used in the following case study.

A basic Model, mdl (figure 3), 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 two-part message length, msg (section 2.1), for a data set. We are only concerned with the most important properties here; a statistical model might be able to do several other things.

Note that in Haskell t->u denotes the function type with input type t and output type u. A data set over a data-space ds has the type [ds], that is 'list of ds.' For example, the pr operator of class Model (figure 3) has the type (mdl dataSpace) -> dataSpace -> Probability. That is, given a model over the dataSpace and a datum from the dataSpace return the probability of the datum.

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 sub-classes must return its own prior probability and message length, msg1, and that it must be able to form mixtures; it must also be in the standard class Show so that we can print the answers to inference problems.
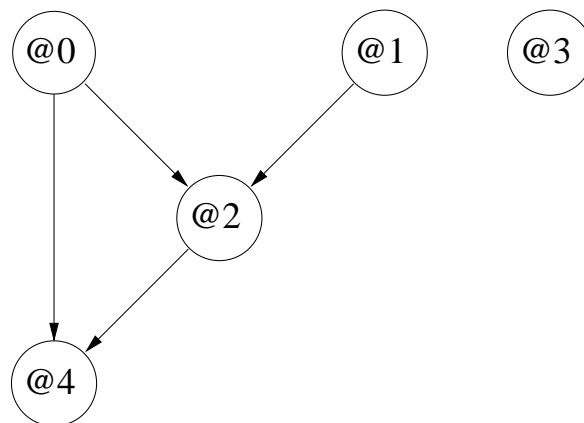


Figure 4: Example Network.

Some 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 m1, a model of ds1, and fm, a function-model from ds1 to ds2. There are related 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, integer, normal and multi-variate distributions, mixture models, Markov models, finite function-models (conditional probability tables) and classification trees, have been defined and made instances of the appropriate classes. 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 (Korb & Nicholson 2004) is a good tool to investigate relationships among the variables of a data set. A Bayesian network is a directed acyclic graph. 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. Creating and applying an estimator of the structure and parameters of a Bayesian network forms our case study to illustrate IP – a network is a non-trivial tool and implementing one provides a good test of a system's expressive power. A Bayesian network is in class Model (section 2.2) and can assign a probability to a data tuple; belief updating was not required by the application and has not been implemented. Figure 4 shows an example Bayesian network in which variable 2 is a child of variables 0 and 1 and is a parent of 4, variable 3 has no relationship to the other variables, and so on. It happens that variables 0 and 4 are continuous and variables 1, 2 and 3 are discrete.

Friedman and Goldszmidt (1996) first suggested using decision-trees (classification trees), in place of the full conditional probability tables (CPTs) often used within the nodes of networks over discrete vari-

ables; Comley and Dowe (2003) 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 created a rather general classification tree algorithm (Allison 2003, Allison 2005). The tree's type is an instance of class `FunctionModel`. Such a tree can test arbitrary variables – discrete, continuous, multi-variate, sequence – from its input space and can have arbitrary distributions over its ouptut space, or even function-models (regressions), in the leaves. These possibilities follow naturally from IP's exploitation of Haskell's polymorphic type system. The classification tree is reused here as the basis of our new Bayesian networks; also see section 3.7. Each classification tree consists of at least one leaf-node, `CTleaf`, and possibly also fork-nodes, `CTfork`. These tree-nodes are *not to be confused* with the network's nodes; there is one tree per network node. The classification tree type is an instance of the class `FunctionModel` (section 2.2). A fork tests a parent (input) variable value. A leaf models the appropriate child (output) variable. Typically the multi-state distribution, `mState`, models a discrete variable, and the normal distribution models a continuous variable but other distributions can be used if desired because the tree estimator is parameterized by the leaf estimator. MML gives a trade-off between the complexity of a tree and its fit to the data and this is used to control the search. Note that when used within a node in a Bayesian network, 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 it shows the composition of statistical models: Multi-state and normal distributions within models of missing data within 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. The point is to investigate how easy it is to adapt IP to a new task. This is important because 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 (2001) lost person data set has been examined in CSSE, Monash (Twardy 2002, Twardy & Hope 2004). Here it provides an application of the Bayesian network case study. There are 363 records, and 15 variables, numbered 0-14. Approximately half of the variable values are missing overall. Attention is sometimes restricted to the first eight variables; one aim is to predict distance travelled, `DistIPP` variable 7, from variables zero to six. In general, workers want to have an "explanation" of the data; the structure and parameters of a network are a good start.

### 3.2 Describing the Data

The first step in the application is to define the variable types in the lost person data set; in Haskell this is done quite naturally as:

```
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, Bounded,...)
data Urban = Rural | Suburban | Urban
            deriving (Ord, Enum, Bounded,...)
type HrsNt   = Double   -- hours notified
type DistIPP = Double   -- distance
...

type MissingPerson =
            (Maybe Tipe, Maybe Age, ... )
```

The Haskell keyword '`deriving`' directs the compiler to add a new data type, for example `Topography`, to standard Haskell classes such as `Ord` (ordered), `Enum` (enumerated) and `Bounded`.

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; also see section 3.6.

A datum, a lost person, is a tuple of the component variables. Haskell's standard Prelude (Peyton-Jones et al 1999) 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 in principle be automated in template Haskell (Sheard & Peyton-Jones 2002), say.

### 3.3 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 that the multi-state distribution and its estimator are polymorphic, being applicable to any bounded enumerated data-space (type).

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

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.

### 3.4 Partitioning Data Spaces: Class Splits

A classification tree, as used in a node of a Bayesian network, 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` (Allison 2003) 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 uses a simple zero-lookahead algorithm in the search to balance tree complexity (`msg1`) against fit to the data (`msg2`).

A multi-variate input space is, by default, split by splitting on one of its component variables. To implement this, the ways of splitting the components are interleaved for consideration in turn.

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 (Wallace & Patrick 1993).

A discrete, `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 also `Ord` (ordered), as can be seen from their definitions, 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 alternatively  = splitsOrd

instance Splits Urban where
  splits = splitsBE
  -- or alternatively  = splitsOrd
```

Yet another alternative was also implemented and tested for lost persons: `Tipe` has seven values and high-arity, un-ordered, discrete types like `Tipe` can cause difficulties to function-models because of the large number of cases and the few data in some or all of them. If some cases 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 a few lines.

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

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

If 'k' sub-sets are specified, their complement is taken to be the (k+1)st. Note that the programmer decides how to group the values in `Tipe` here. In principle a program could search through the possibilities but it would, of course, add to the overall search time.

It is a simple matter in IP to implement extensions, such as `setSplits`, to models to suit a problem and its data.

## 3.5 Selecting Sub-Spaces: Projections

In a typical application of a classification tree, the input variables and the output variable are fixed. But here, in a Bayesian network, the selection of parent (input) and child (output) variables must be under program control on a node by node basis; this property made the case study particularly interesting from the IP point of view. 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 (section 3.7) does not "care" what types the data and sub-estimators have, provided only that they are consistent. An instance, type `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

```
estNetwork perm estMV dataSet =
 let
  n = (length . selAll) (estMV [])
  search _  []     = []  -- done
  search ps (c:cs) =
   --parents ps, children c:cs
   let
    opFlag  = ints2flags [c] n  --child
    ipFlags = ints2flags ps  n  --parents
    cTree = estCTree
            (estAndSelect estMV opFlag)
            (splitSelect ipFlags)
            dataSet dataSet
   in cTree : (search (c:ps) cs)

  trees  = search [] perm       --network
  msgLen = sum (map msg1 trees) --total msg1
  nlP datum = sum
    (map(\t -> condNlPr t datum datum) trees)
 in
    MnlPr msgLen nlP         --return a Model
          (\() -> "Net:"++(show trees))
```

Figure 5: Network estimator.

case of a `Model` this behaviour is to return zero information, probability one, for non-selected variables, i.e. 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 3.4, class `Splits` exists for partitioning data-spaces – discrete, continuous, multi-variate or whatever other data-spaces are made instances of it. A new class `Splits2`, inspired by `Project`, was defined (it could perhaps be folded into class `Splits`) to allow splitting on only selected variables:

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

## 3.6 Handling Missing Data

The lost person data set is difficult 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 already has the ideal type to represent possibly missing values: `Maybe`. New operators were implemented to extend arbitrary statistical models to cover possibly missing values. Rather than cleaning the data – deleting data with missing values – or imputing (replacing) missing values, missing-ness 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`; it takes a message length, a negative log likelihood function, and a description which shows the model.

The related high-order 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
```

This is not the same as just coding missing-ness as a "special" value because it is not estimated with the given definition of m1.

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

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

In addition to modelling, missing values also affect splits, that is partitions of the data (section 3.4). A simple strategy is for the variable to be split as for the underlying type but with an extra option for missing (Nothing) cases. Other splitting 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 that is sufficient for the application.

## 3.7 Mixed Bayesian Networks and the Lost Person Network

The function, estNetwork (figure 5), 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 (Korb & Nicholson 2004). It is sufficient for this application because a plausible ordering of the variables is fairly obvious but, in principle, it would be possible to search over permutations. Such a search would have to be heuristic if there were many variables, and information theory does suggest some heuristics, but the simple algorithm does not do this and the permutation is currently taken to be common knowledge.

Internally estNetwork uses the estimator for classification trees (Allison 2003), estCTree, to do much of the work. The remainder consists of organising selector flags (section 3.5) corresponding to the allowed parents for the child in the current node. Note that the dataSet seems to be passed to estCTree twice, as both the input and output variables – its third and fourth parameters. But its first and second parameters use straightforward auxiliary functions ints2flags, estAndSelect and splitSelect, to flag the child (output) to be predicted by the leaf estimator and the parents (input) to be used for splitting as appropriate at each node in the network.

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 probably depends on one
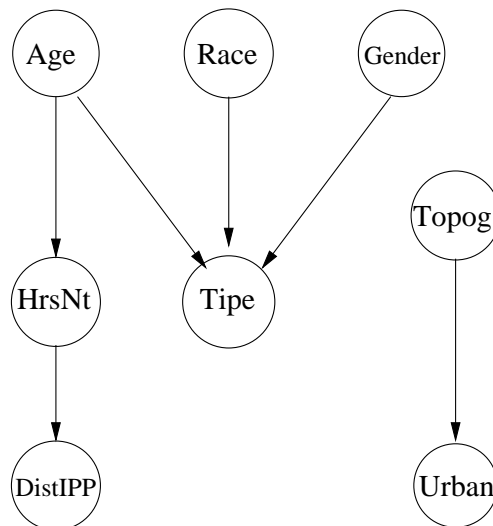


Figure 6: Lost Person Network 1.

or more of 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   --model

nullModel = estMissingPerson dataSet
```

Figure 6 shows the first network inferred for variables 0 to 7; the node parameters are inferred but not shown.

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

When Tipe was allowed to split according to setSplits [[Alzheimers], [Child]] (section 3.4), 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; variable 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,
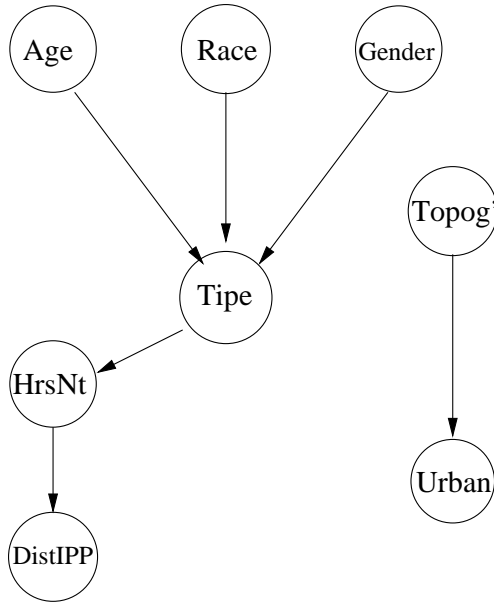FindRes = Ground | Air | Local | Law | Dog,

Figure 7: Lost Person Network 2.

```
FindLoc = Brush | Woods | Field | Water |
Linear | Thing,
HrsFind (continuous),
HrsTo (continuous).
These extra variables can all be missing.
```

## 3.8 Comparisons

Weka (Witten & Frank 1996) which is based on Java is perhaps the system closest to the present work. Weka's Bayesian networks "assume that all variables are discrete" (Bouckaert 2004) p. 22 and "a limitation of the current classes is that they assume that there are no missing values" (Bouckaert 2004) p. 23. In Weka, continuous variables must be discretized first and how this is done may affect the final result. Discretization is unnecessary in IP for modelling and, for splitting, is part of the network optimization as a by-product of using our classification trees (section 3.2). Missing-ness was built into the model when necessary (section 3.6).

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 (Conklin & Witten 1994, Srinivasan, Muggleton & Bain 1994) but this aspect of ILP is less developed than work on MML. The programmer is involved in the design of the search algorithm (section 3.7) 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 models.

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.
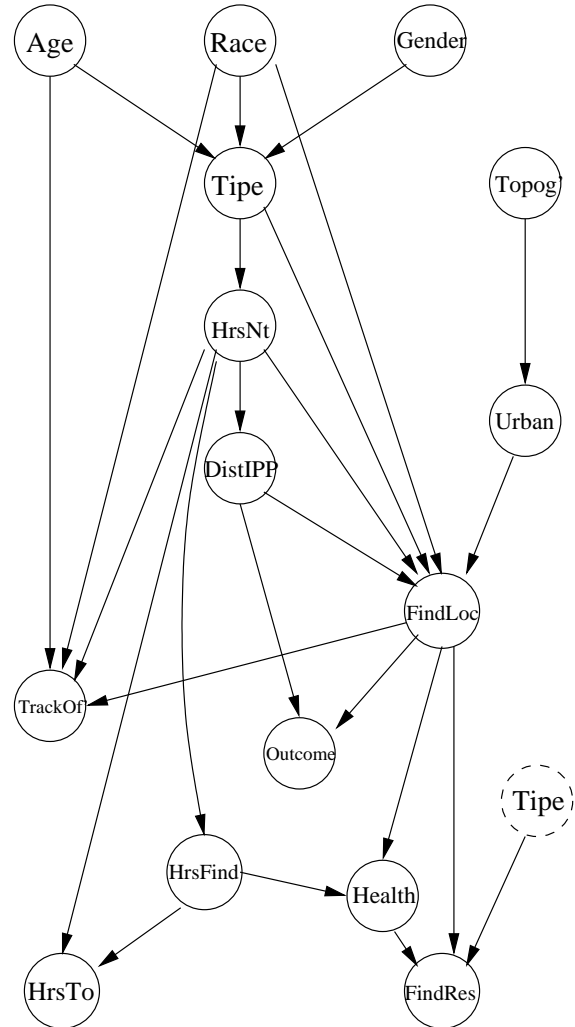


Figure 8: All 15 Variables.

## 4 Conclusions

Inductive programming (IP) uses the compositional abilities of functional programming, Haskell and minimum message length (MML) inference. 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 data-analyst will work in 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 whether to split `Topography` as discrete or as ordered data (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 (Allison 2004) 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 (Sheard & Peyton-Jones 2002) 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 can also be the case with statistical models and their estimators. There is potential for an extensive library of operators on statistical
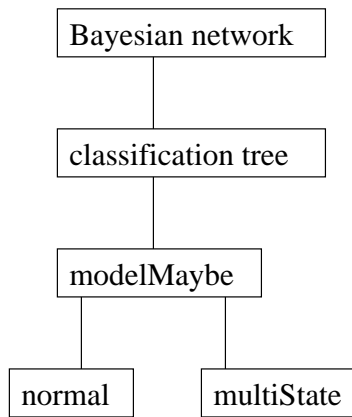
Figure 9: Model Layers

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 but they are handled by `modelMaybe` (section 3.6) and other functions, and are combined in the complexity of the Bayesian network (section 3.1 and 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 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` (section 3.5) took one day to create. The lost person application (section 3.2) came along some weeks 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.

## 4.1 Acknowledgments.

It is a pleasure 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.

## References

Allison, L. (2003), Types and classes of machine learning and data mining, *in* 26th Australasian Computer Science Conference (ACSC), pp. 207–215.

Allison, L. (2004), Inductive inference 1.1, TR 2004/153, School of Computer Science and Software Engineering, Monash University. http://www.csse.monash.edu.au/~lloyd/tildeFP/II/

Allison, L. (2005), 'Models for machine learning and data mining in functional programming', *J. Functional Programming* **15**(1), pp. 15–32, doi:10.1017/S0956796804005301.

Allison, L., Powell, D. & Dix, T. I. (1999), 'Compression and approximate matching', *BCS Computer J.* **42**(1), pp. 1–10.

Baxter, R. A. & Oliver, J. J. (1994), MDL and MML: Similarities and differences, TR 207, Department of Computer Science, Monash University. (Amended 1995.)

Bayes, T. (1763), 'An essay towards solving a problem in the doctrine of chances', *Phil. Trans. of the Royal Soc. of London* **53**, pp. 370–418, and reprinted in *Biometrika* **45**(3/4), pp. 296–315, 1958.

Bouckaert, R. R. (2004), Bayesian networks in Weka, TR 14/2004, Comp. Sci. Dept.. U. of Waikato.

Comley, J. & Dowe, D. (2003), General Bayesian networks and asymmetric languages, *in* 2nd Hawaii Int. Conf. on Statistics and Related Fields (HICS-2).

Conklin, D. & Witten, I. H. (1994), Complexity-based induction, *Machine Learning* **16**(3), pp. 203–225.

Crawley, M. J. (2002), *Statistical Computing – an Introduction to Data Analysis using S-Plus*, Wiley.

Farr, G. E. & Wallace, C. S. (2002), 'The complexity of strict minimum message length inference', *BCS Computer J.* **45**(3), pp. 285–292.

Friedman, N. & Goldszmidt, M. (1996), Learning Bayesian networks with local structure, *in* Uncertainty in A.I., pp. 252–262.

Gammerman, A. & Vovk, V. (eds) (1999), Special Issue on Kolmogorov Complexity, *BCS Computer J.* **42**(4).

Georgeff, M. P. & Wallace, C. S. (1984), A general selection criterion for inductive inference, *in* European Conf. on Artificial Intelligence (ECAI84), Pisa, pp. 473–482.

Koester, R. J. (2001), 'Virginia dataset on lost-person behaviour', author's site http://www.dbs-sar.com/.

Korb, K. B. & Nicholson, A. E. (2004), *Bayesian Artificial Intelligence*, Chapman and Hall / CRC.

Peyton-Jones, S. et al, (1999), *Report on the Programming Language Haskell-98*, Available at http://www.haskell.org/.

Powell, D. R., Allison, L. & Dix, T. I. (2004), Modelling alignment for non-random sequences, *in* 17th ACS Australian Joint Conf. on Artificial Intelligence (AI2004), Springer-Verlag, LNCS/LNAI Vol. 3339, pp. 203–214.

R, various authors, (2004), 'CRAN: The Comprehensive R Archive Network', http://lib.stat.cmu.edu/R/CRAN/.

Rissanen, J. (1978), 'Modeling by shortest data description', *Automatica* **14**, pp. 465–471.

Rissanen, J. (1987), 'Stochastic complexity', *J. Royal Statistical Society series B.* **49**(3), pp. 223–239 and 252–265.

Shannon, C. E. (1948), 'A mathematical theory of communication', Bell Syst. Technical Jrnl. **27** pp. 379–423 and pp. 623–656.

Sheard, T. & Peyton-Jones, S. (2002), Template meta-programming for Haskell, *in* Proc. of the Workshop on Haskell, ACM, pp. 1–16.

Srinivasan, A., Muggleton, S. & Bain, M. (1994), 'The justification of logical theories based on data compression', *Machine Intelligence* **13**, pp. 87–121.

Twardy, C. R. (2002), 'SAR*bayes*: Predicting lost person behavior', presented to National Association of Search and Rescue (NASAR), available at http://sarbayes.org/nasar.pdf.

Twardy, C. R. & Hope, L. (2004), 'Missing data on missing persons', School of Computer Science and Software Engineering, Monash University, personal communication.

van Deursen, A., Lint, P. & Visser, J. (2000), Domain-specific languages: An annotated bibliography, *in* ACM SIGPLAN Notices **35**(6), pp. 26–36.

Wallace, C. S. (2005), *Statistical and Inductive Inference by Minimum Message Length*, Springer-Verlag.

Wallace, C. S. & Boulton, D. M. (1968), 'An information measure for classification', *BCS Computer J.* **11**(2), pp. 185–194.

Wallace, C. S. & Freeman, P. R. (1987), 'Estimation and inference by compact coding', *J. Royal Statistical Society series B.* **49**(3), pp. 240–265.

Wallace, C. S. & Georgeff, M. P. (1983), A general objective for inductive inference, TR 32, Dept. of Computer Science, Monash University.

Wallace, C. S. & Patrick, J. D. (1993), 'Coding decision trees', *Machine Learning* **11**, pp. 7–22.

Witten, I. H. & Frank E. (1999), Nuts and bolts: Machine learning algorithms in Java, *in* Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations, Morgan Kaufmann, pp. 265–320.