The Types of Models

Lloyd Allison,
School of Computer Science and Software Engineering,
Monash University,
Victoria,
Australia 3800.
http://www.csse.monash.edu.au/~lloyd/

15 March 2003

Abstract

The specification of various kinds of statistical model from machine learning and data mining is examined formally using the type and class system of the functional programming language Haskell as a meta-language. Types and classes (in the programming sense) of models, and operations on models, are defined; many are naturally *polymorphic*. Convenient conversion functions map between the classes of models and extend their range of usefulness. The result is a kind of theory of *programming with* models, not only of using them. The "theory" can run as an executable Haskell program or can throw light on the foundations of platforms for programming with statistical models. [1]

Keywords. Model, polymorphism, type, class.

1 Introduction

A good deal of research in machine learning and data mining develops a new statistical model, or devises a better algorithm to fit such a model to data, or applies such a model to some problem area. Sometimes a "classical" statistical model is involved, and sometimes a rather different kind of "model" is studied, e.g. an artificial neural network or a support vector machine. For want of a name we use the term *statistical model* to cover all and any of the above that deal in probabilities.

This paper develops something different. It examines what is a statistical model from a computer-programming point of view: How does a model behave, what can it do, what can be done to it, and what can it associate with? The functional programming (FP) language Haskell-98 [12] is used as the meta-language for the study. Various kinds of model are defined by specifying their types and classes (in the programming sense). Operators on classes of statistical model are defined and enhance the models' generality and range of application. Many of the operators are naturally polymorphic.

[1] Presented at the Second Hawaii International Conference on Statistics and Related Fields (HICS03), Honolulu, 2003 June 5-8.

There are some general data-mining platforms such as R [5], S-Plus [17] and Weka [22] that give some support to programming with models but their notion of *type* tends to be dynamic and *ad-hoc*. The present work brings the precision and generality of statically-checked (compile-time), polymorphic types to the problem.

We develop data types and classes for (basic) models including probability distributions, for function models including regressions, and for time series such as Markov-models. Example models and estimators are given. Conversion functions on models and their estimators allow them to be used more generally than otherwise. The result is a kind of "theory" of statistical models and of programming with them. It can be seen as a tool for investigating platforms for programming with models, and can also be run as an executable program in its own right.

2 Preliminaries

The paper calls on minimum message length (MML) inference, functional programming (FP) (hence MMLFP [2]) and Haskell so those topics are briefly introduced here for completeness.

2.1 Functional Programming

The origins of functional programming (FP) are in Church's lambda calculus [4]. Notably, functions are *first-class values*; i.e. they can be parameters and results of functions, and elements of data structures. Modern FP languages, such as Standard-ML [10] and Haskell [8, 12], typically have static, polymorphic type systems where a structured data type may have one or more *type parameters* which can be instantiated in many ways – hence *polymorphic*. For example `[t]→u` is the type of those functions whose parameter is a list, `[ ]`, of any element type, `t`, and whose result is of some type, `u`. E.g. `length :: [t]→Int`, that is function `length` is of type `[t]→Int`, can be applied to *any* list, and returns an integer. Another common FP feature is a *type inference* algorithm which automatically infers the types of most expressions in a program; the programmer rarely needs to give types explicitly. Lastly, Haskell has a system of *type classes*. A class is defined by the operators required on a data type that is an *instance* of the class. There are clear-cut divisions in Haskell: An expression evaluates to a value, a value has a type, and a type can be an instance of one or more classes.

FP gets great expressive power from the above features, enough power to provide the meta-language of denotational semantics [9] which can define the semantics of essentially all features found in common programming languages. The objective here is to start to define the semantics of statistical models. Data types and classes of models, and operations on them, are used to specify the behaviour of models. Many models and associated operations are naturally polymorphic. Types and classes are checked statically, by the compiler. The result is a rigorous "theory" that can be run as a program.

2.2 Minimum Message Length (MML)

Overfitting is a common problem in statistical inference; it really must be addressed somehow in an inference system and various methods have been proposed as cures. The choice of a particular method is orthogonal to the main points of this paper but the work was done within the framework of one such method, namely minimum message length (MML) inference [18, 19], and once a choice is made the method's features do appear and reappear so MML is briefly described. Both MML [18] and the minimum description length (MDL) principle [14] are Bayesian methods based on information theory. MML is invariant, consistent and resistant to overfitting.

The MML paradigm considers a *transmitter* and a *receiver*. Initially the transmitter and receiver agree on codes and algorithms to be used to transmit models (distributions, hypotheses, theories) and data. Naturally they design codes that give optimum compression in *expectation*. They are then separated and the transmitter is given data to send to the receiver. A *two part message* is used: The first part states a model, M, and the second part states the data, D, given M. It is, of course, arranged that M is a solution to an inference problem of interest; the transmitter is made to state an opinion. We have from Bayes theorem [3] and from Shannon's mathematical theory of communication (hence "message") [15]:

```
  Pr(M & D) =  Pr(M).Pr(D|M)  =  Pr(D).Pr(M|D)
 msg(M & D) = msg(M)+msg(D|M) = msg(D)+msg(M|D)
```

where `msg(E)=-log(Pr(E))` is the message length in an optimal code of an event, E. Almost invariably we use the results of Shannon, coding theory and data compression to calculate what the length of a message *would be* without actually doing the encoding and decoding.

The first part, `msg(M)`, of a two part message is a measure of the model's *complexity* and generally increases with the number of parameters of the model and with the precision to which they are stated. Its inclusion makes for a trade-off with the second part, `msg(M|D)`, i.e. the negative log likelihood, that is the fit of the model to the data. Anything that is not common knowledge must be included else the message will not be decodable; this keeps us "honest". Continuous-valued parameters must be stated to optimum, *finite* precision. The complexity of Strict MML (SMML) inference is, in general, NP-hard [7] but good practical MML approximations exist for many important inference problems [19].

3 Models

Here we define a Haskell class, `Model`, a broad sub-set of the collection of all statistical models. The most important property of a `Model` is its ability to assign a probability, `pr`, to a datum from its data space (sample space). Equivalently for MML purposes, a `Model` assigns a message length, `msg2` (for 2nd-part), to a datum. (A `Model` of a continuous data space instead assigns a probability-*density* which combines with the data measurement-accuracy to give a probability; in reasonable conditions this accuracy "passes through" the calculations and we will ignore the distinction here for simplicity.) A `Model` may be able to do other things, such as generate sample data, but pr seems to be most important. Class Model shares further properties with other classes to be defined, such as having a message length (complexity), msg1, of its own. Common properties are attached to a super-class, SuperModel; see Sect.4. The Haskell code below states that a data type, mdl, is in class Model if it has a type parameter, dataSpace, and if the required operations are defined. Note that mdl dataSpace is also *required* to be a SuperModel, by the condition SuperModel (mdl dataSpace) =>... in the type of msg:

```
class class Model mdl where
  pr   :: (mdl dataSpace) -> dataSpace -> Probability

  msg  :: SuperModel (mdl dataSpace) =>
          (mdl dataSpace) -> dataSpace -> MessageLength
  msg2 :: (mdl dataSpace) -> dataSpace -> MessageLength

  pr   m d = 2 ** (-msg2 m d)       -- Defaults; an
  msg2 m d = -(logBase 2 (pr m d))  -- instance must
  msg  m d = (msg1 m) + (msg2 m d)  -- define pr or msg2.
```

One or more data types such as ModelType can now be declared and made *instances* of class Model. ModelType gives a choice between two *constructor* functions – MPr and MMsg. A value of ModelType, i.e. an actual Model, is made by giving MPr its complexity and probability function, or equivalently by giving MMsg its complexity and message length function for data:

```
data ModelType dataSpace =
  MPr  MessageLength (dataSpace -> Probability)   |
  MMsg MessageLength (dataSpace -> MessageLength)

instance Model ModelType where
  msg2 (MPr  mdlLen p) d = - logBase 2 (p d)    -- in bits
  msg2 (MMsg mdlLen m) d = m d
```

Some Models have zero message length because they have no parameters, e.g. *universal* Models for integers [6, 14]. Wallace's integer model is an example of a universal Model of non-negative Ints, based on a code originally presented for classification-trees [21]: The *length* of a code word for an integer is always an odd number of bits. The "steps" where message lengths increase are always of 2-bits so this universal (model and) code is smoother than some others [14]; numbers of integers having code words of [1, 3, 5, 7, 9,...] bits are given by the Catalan numbers [1, 1, 2, 5, 14,...].

```
wallaceIntModel =
  let catalans = ...
      cumulativeCatalans = scanl1 (+) catalans
      find n posn (e:es) =
        if n < e then posn else find n (posn+1) es
  in MMsg 0 (\n -> (find n 0 cumulativeCatalans) * 2 + 1)
```

Budding Haskell programmers note that scanl1 (+) forms cumulative sums which are searched, above, by find. The pattern 'e:es' matches non-empty lists that start with e and continue with es. '\', for lambda, and '→' are used to define anonymous functions, e.g. \n→n+1 is the successor function. Lazy evaluation allows an infinite list, such as catalans, to be defined in Haskell, provided that not all of its elements are used.

A probability distribution over a range of Ints [0..n-1] can be estimated from frequencies of occurrence by a function such as freqs2model:

```
freqs2model fs =
  let total = foldl (+) 0 fs
      probs = ...       -- obvious
      part1 = ...
      p n = probs !! n
  in MPr part1 p
```

freqs2model returns a Model, given a list of n frequencies, fs. Note that foldl (+) 0 fs sums the elements of fs and that '!! n' selects the nth element of a list. Calculating the Model's complexity, i.e. part1, was specified by Wallace and Boulton [18]; the details are not relevant here. p is the probability function. Similarly, estimators can be given for other distributions such as the normal (Gaussian) and so on.

A simple example operator, modelInt2model, on Models converts a Model of Int into a Model of some other discrete data space of appropriate size. An example value, egValue, is used to inform the type checker.

```
modelInt2model egValue intModel =
  let toInt datum = ...
      p datum = pr intModel (toInt datum)
  in MPr (msg1 intModel) p
```

A *multi-state* distribution can now be estimated for an enumerated, bounded data space. Occurrences in a dataSet are counted and the frequencies used to form a Model of Int which is converted to the data space:

```
estMultiState dataSet =
  modelInt2model (dataSet !! 0) (freqs2model (count dataSet))
```

For example, myCoin=estMultiState [H,H,T,H,...] is a Model of throws of a coin. Throw is its data space. It cannot be accidentally used with any other type of data. E.g. The type checker accepts pr myCoin H and rejects pr myCoin True, say.

A bivariate Model can be formed from a pair of Models, m1 and m2, and their data spaces. For example, bivariate fairCoin wallaceIntModel is a Model of a throw *and* a non-negative integer.

```
bivariate (m1, m2) =
  let m (d1, d2) = (msg2 m1 d1) + (msg2 m2 d2)
  in MMsg ((msg1 m1) + (msg1 m2)) m
```

Operator bivariate assumes that the two attributes (variables) are independent; a more complex factor-Model [20] could be created. A bivariate *estimator* can also be formed from a pair of univariate estimators.

4 More Classes

There are many kinds of statistical model that are not covered by class `Model` (Sect.3). Two more important classes are `FunctionModel` and `TimeSeries`. These share some properties, notably `msg1`, with `Model`: They are all subclasses of `SuperModel` mentioned before. Other common properties are having a *prior* probability and being able to form a mixture. Forming a *mixture*, that is a weighted average as in mixture modelling, is an important operation not only on `Models` but also on `FunctionModels` and `TimeSeries`, and hence on `SuperModels`. A data type, `MixtureType`, contains a `Model` over the components and a list of components. Class `Mixture` specifies that an instance must deliver its `components` and a `Model`, `mixer`, which amounts to the "weights" over them:

```
class SuperModel sMdl where
  prior    :: sMdl -> Probability
  msg1     :: sMdl -> MessageLength
  mixture  :: (Mixture mx, SuperModel (mx sMdl)) =>
              mx sMdl -> sMdl
  ...

class Mixture mx where
  mixer      :: (SuperModel t) => mx t -> ModelType Int
  components :: (SuperModel t) => mx t -> [t]

data (SuperModel elt) =>
  MixtureType elt = Mix (ModelType Int) [elt]

instance (SuperModel elt) =>
  SuperModel (MixtureType elt) where
    msg1 (Mix m es) = foldl (+) (msg1 m) (map msg1 es)
```

Finally as promised earlier (Sect.3), we can turn a `Model`, or at least `ModelType`, into a `SuperModel`:

```
instance SuperModel (ModelType dataSpace)  where
  msg1 (MPr  mdlLen p) = mdlLen
  msg1 (MMsg mdlLen m) = mdlLen
  mixture mx = ...
```

A `mixture` of `Models` of a data space is itself a `Model` of the data space.

4.1 Function Models

A `FunctionModel` captures the relationship between input (independent, exogenous) attributes (variables) and output (dependent, endogenous) attributes, e.g. a linear-model of `x` fitting `y` with `linear a b epsilon`, i.e. `a*x+b` with noise from `normal 0 epsilon`. A `FunctionModel` produces a *conditional* `Model` of its output space, `opSpace`, given a value from its input space, `inSpace`. Conditional probabilities, `condPr`, and message lengths can be got from the conditional `Model`:

```
class FunctionModel fm where
  condModel :: (fm inSpace opSpace) ->
               inSpace -> ModelType opSpace
  condMsg2  :: (fm inSpace opSpace) ->
               inSpace -> opSpace -> MessageLength
  condPr    :: (fm inSpace opSpace)
               -> inSpace -> opSpace -> Probability
  ...
data FunctionModelType inSpace opSpace =
       FM MessageLength (inSpace -> ModelType opSpace)
```

`FunctionModelType` is made an instance of `FunctionModel` and `SuperModel` in the obvious way. Later (Sect.6) classification trees are made `FunctionModels`. A mixture of `FunctionModels` is also a `FunctionModel`.

As an example, a *finite* `FunctionModel` over finite input and output spaces can, assuming no correlation, be estimated by counting, using the estimator for a multi-state distribution (Sect.3) for each input *case*:

```
estFiniteFunction ipSeries opSeries =
  estFiniteIpFunction estMultiState ipSeries opSeries
```

4.2 Time-Series

A `TimeSeries` describes a data series, perhaps one dependent literally on time, or just a long sequence such as a biological sequence [16]. The `predictors` function of a `TimeSeries` returns a sequence (list) of predictions, i.e. a sequence of `Models`, for the *next* value given the *context* of preceding values at each position. One natural way to define a value of `TimeSeriesType` is to give its message length (complexity) and a function that maps from a context to a `Model`. The probabilities, `prs`, and message lengths, `msg2s`, *per* datum can be obtained from `predictors`.

```
class TimeSeries tsm where
  predictors :: (tsm dataSpace) ->
                [dataSpace] -> [ModelType dataSpace]
  msg2s      :: (tsm dataSpace) ->
                [dataSpace] -> [MessageLength]
  prs        :: (tsm dataSpace) ->
                [dataSpace] -> [Probability]
  ...
data TimeSeriesType dataSpace =
  TSM MessageLength ([dataSpace] -> ModelType dataSpace)
```

`TimeSeriesType` is an instance of `SuperModel` and of `TimeSeries` in the obvious way.

As an example, a Markov-model of order k can be estimated by using the estimator for a `FunctionModel` on discrete lists of length k. The data series is *scanned* to form a sequence of contexts. The contexts are the inputs for the `FunctionModel` and the latter is converted into the desired `TimeSeries` by `functionModel2timeSeries` (see Sect.4.3):

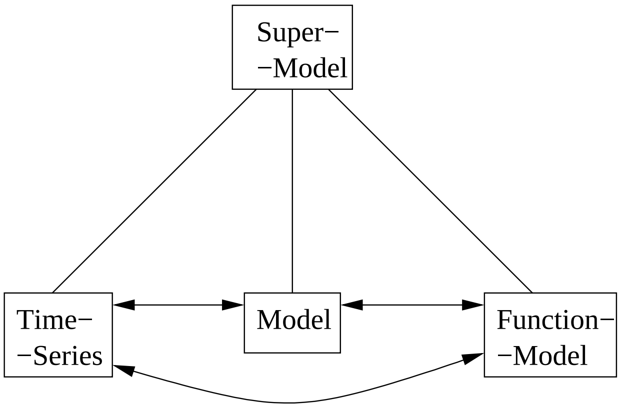

Figure 1: Classes.

```
estMarkov k dataSeries =
  let scan ... = ...
      contexts = scan dataSeries []
  in functionModel2timeSeries
       (estFiniteListFunction k contexts dataSeries)
```

4.3 Conversion Functions

Model, FunctionModel, TimeSeries and SuperModel cover a good range of statistical models including probability distributions, (unsupervised) mixture models, Markov-models, regressions, (supervised) classification functions, and so on. There are some useful conversion functions between the classes (Figure 1).

A Model of a dataSpace can be mapped trivially to a FunctionModel of some inSpace and dataSpace by always returning the given Model. In a similar way the Model can be mapped to a TimeSeries of dataSpace by ignoring the context of past values when making a prediction. We give the types of the functions although the programmer does not need to do this, rather the compiler infers the types in practice:

```
model2functionModel :: (Model mdl) =>
    mdl dataSpace -> FunctionModelType inSpace dataSpace

model2timeSeries :: (Model mdl) =>
    mdl dataSpace -> TimeSeriesType dataSpace
```

A FunctionModel of inSpace and opSpace can be turned into a Model of (inSpace, opSpace) by using condModel of the FunctionModel and effectively taking the input attributes as common knowledge – this is valid in some supervised learning problems. A FunctionModel of [dataSpace] and dataSpace can be turned into a TimeSeries of dataSpace by "applying" the FunctionModel to each prefix of the data series:

```
functionModel2model :: (FunctionModel fm) =>
    fm inSpace opSpace -> ModelType (inSpace, opSpace)

functionModel2timeSeries :: (FunctionModel fm) =>
    fm [dataSpace] dataSpace -> TimeSeriesType dataSpace
```

A `TimeSeries` of `dataSpace` can map to a `Model` of whole data series, that is of `[dataSpace]`: The message length of a data series includes a term for the length of the series, e.g. under `wallaceIntModel` (Sect.3), say, and a term for each value in the series. A `TimeSeries` of `dataSpace` can also be mapped onto a `FunctionModel` of `[dataSpace]` and `dataSpace` by using the prediction made by the `TimeSeries` in the context of the whole data series:

```
timeSeries2model :: (TimeSeries tsm) =>
                    tsm dataSpace -> ModelType [dataSpace]

timeSeries2functionModel tsm :: (TimeSeries tsm) =>
  tsm dataSpace -> FunctionModelType [dataSpace] dataSpace
```

A `FunctionModel`, `fm`, describes the relationship of its input space, `inSpace`, to its output space, `opSpace`. A `Model` of `(inSpace, opSpace)` can be formed if we are also given a `Model`, `mdl`, of `inSpace`:

```
conditionalize mdl fm =
  let p (ip, op) = (pr mdl ip) * (condPr fm ip op)
  in MPr ((msg1 mdl)+(msg1 fm)) p
```

The set of conversion functions on statistical models is mirrored in a set of similar functions on their estimators. They are more than interesting curiosities, enabling all models to be used in new ways, as illustrated below.

5 Unsupervised Classification++

Unsupervised classification, also known as clustering, provided one of the first applications of information-based machine learning: Given multivariate data, find a mixture model that best describes the data. The number of components of the `Model` and the components' parameters are *not* known in advance. *Snob* [18] used (and uses) MML to solve the problem of balancing the complexity of the mixture against its fit to the data.

An estimator for a mixture is easily expressed in our present system. For simplicity only, we consider an estimator, `estMixture`, for a *given* number of components, but in principle it can be used for a search through 1, 2, 3,... components to some reasonable limit. `estMixture` is given a list, `ests`, of *weighted* estimators, one per component (different kinds of distribution can be used), and data. In a simplification (and slight abuse) of the type notation we have, in spirit:

```
estMixture ::
  [[dataSpace] -> [Double] -> Model dataSpace]   -- estimators
  -> [dataSpace]                               -- training data
  -> Mixture (Model dataSpace)                        -- result
```

`estMixture` starts by allocating random fractional *memberships* of the data across the components of the mixture. A new mixture is fitted to the current memberships. New memberships are fitted to the current mixture, and so on. This leads to a typical expectation-maximization cycle.

```
estMixture ests dataSet =
 let
  memberships (Mix mixer components)  = ...
  randomMemberships = ...
  fit (est:ests) (mem:mems) = (est dataSet mem):(fit ests mems)
  fit [] [] = []
  fitMixture mems = Mix (freqs2model (map (foldl (+) 0) mems))
                        (fit ests mems)
  cycle    mx = fitMixture (memberships mx)
  cycles 0 mx = mx
  cycles n mx = cycles (n-1) (cycle mx)
 in mixture(cycles <some_Number> (fitMixture randomMemberships))
```

Note that the use of total-assignment would lead to biased estimates and that the use of fractional memberships with weighted estimators, as above, is unbiased.

In principle `estMixture` lets us find mixtures of *any* kind of distribution and data provided only that distribution(s) and data match. It can also be used with conversion functions (Sect. 4.3). E.g. An estimator for a `FunctionModel` of `inSpace` and `opSpace` can be converted into an estimator for a `Model` of `(inSpace, opSpace)` for use with `estMixture` which can therefore, in effect, infer mixtures of `FunctionModels`, and so on.

6 Supervised Classification++

The *supervised classification problem* is: Given data that have already been classified, i.e. corresponding elements from `inSpace` and from `opSpace`, infer a `FunctionModel` to describe the relation between `inSpace` and `opSpace`. In the simplest case `opSpace` is a finite space of *categories*. There are many implementations of classification functions, for example classification trees (sometimes called decision-trees) [13], classification graphs [11], and support vector machines (binary case) to name just three.

`CTreeType`, for classification-tree, is an example of a `FunctionModel`. A tree consists of leaves (`CTleaf`) and forks (`CTfork`). Each leaf holds a `Model` of the output space. Each fork holds a selector-function to test values of input attributes to select a subtree, the function's message length (complexity), and a list of subtrees. `CTreeType` is made an instance of `FunctionModel` (Sect.4.1):

```
data CTreeType inSpc opSpc =
  CTleaf (ModelType opSpc)  |
  CTfork MessageLength (inSpc->Int) [CTreeType inSpc opSpc]

instance FunctionModel CTreeType where
  condModel (CTleaf leafModel)   i = leafModel
  condModel (CTfork fnLen f dts) i = condModel (dts!!(f i)) i
```

Note that other kinds of fork, for example a fork that forms a *mixture* of the subtrees [2], are also possible.

Our classification-trees are already very general: The leaves can contain *arbitrary* `Models` – of discrete, continuous or multivariate output spaces.

An estimator, `estCTree`, of classification-trees easily fits into our system. It is given an estimator for leaf `Models`, a method of producing valid partitioning functions, and training inputs and outputs. A partitioning function tests the input attribute(s) and divides the training data into *parts*; such functions are easily created to split on discrete and continuous attributes. By a simplification of the type notation we have roughly:

```
estCTree :: ([opSpace]->Model opSpace)          -- est' a leaf
         -> ([inSpace]->[inSpace->PartNums])     -- partitions
         -> [inSpace] -> [opSpace]             -- training data
         -> CTreeType inSpace opSpace
```

An example search algorithm compares the simplest tree (1-leaf) with more complex trees (each of 1-fork, under 0-lookahead). The best is chosen on the basis of total message length. The search is over if the 1-leaf tree is best, otherwise it continues recursively on each subtree with its *appropriate part* of the data. A tree's structure is part of its message length (complexity); the details [21] are interesting but not relevant here.

Recalling (Sect.4.3) that an estimator for a `FunctionModel` of `inSpace opSpace` can be converted into one for a `Model` of `(inSpace, opSpace)`, we see that `estCTree` can also infer `FunctionModel`-trees, i.e. regression-trees:

```
estFunctionModel2estModel estFn ipOpPairs =
  functionModel2model (uncurry estFn (unzip ipOpPairs))

fnMdlTree = estCTree (estFunctionModel2estModel estFnMdl)
                     splitFns ipTrainSet opTrainSet
```

The details of `estFunctionModel2estModel` are unimportant; the point is that the little two-line function adapts our trees to a whole new problem. Combinations such as this show the generality of the system.

7 Conclusion

The semantics of a range of statistical models from machine learning and data mining has been defined, covering basic `Models` and probability distributions, `FunctionModels` such as conditional probability tables and classification functions, `TimeSeries` such as Markov-models, and mixtures. Data types and type classes have been given to make precise models' behaviour, operations on them, and functions between them. This *model of modelling* is expressed in the functional programming language Haskell-98 so it is statically type-checked and can be run.

Functional programming gained great expressive power from treating functions as first-class values; here we have treated models as first-class values. That is, models can be parameters of functions and of other models, results of functions and elements of data structures. This has previously been very useful in special cases [1] and we have generalized the idea. There is clear potential for a sophisticated library of types, classes and operations on many kinds of statistical model.

The Haskell notation is succinct and powerful. Its polymorphic types and type inference algorithm are invaluable as many (most?) models, estimators and operations on them are naturally polymorphic. One result of the work is an improved understanding of the behaviour of various kinds of models and their estimators. It can be considered a rapid-prototype for a data-mining platform.

The current code cannot be used for data-mining of *huge* data sets because it assumes that the data fit in memory, although that restriction could be removed in principle. On the other hand, the code is more than a mere toy and can solve real problems. For example, estimators have been given for supervised and unsupervised classification (Sect.5,6). Thanks to the class design and conversion functions, such estimators and their results are already more general than would otherwise be the case.

References

[1] Allison, L., Powell, D., Dix, T. I.: Compression and Approximate Matching. Computer Journal 42(1) (1999) 1–10

[2] Allison, L.: Types and Classes of Machine Learning and Data Mining. 26th Australasian Computer Science Conference (ACSC), Adelaide, ACS Series Conferences in Research and Practice in Information Technology V16 (2003) 207–215

[3] Bayes, T.: An Essay Towards Solving a Problem in the Doctrine of Chances. Phil. Trans. of the Royal Soc. of London 53 (1763) 370-418. Reprinted in Biometrika 45(3/4) (1958) 293–315

[4] Church, A.: The Calculi of Lambda Conversion. Princeton University Press (1941)

[5] The Comprehensive R Archive Network. http://lib.stat.cmu.edu/R/CRAN/ (current 2002)

[6] Elias, P.: Universal Codeword Sets and Representations of the Integers. IEEE Trans. Inform. Theory IT-21 (1975) 194–203

[7] Farr, G. E., Wallace, C. S.: The Complexity of Strict Minimum Message Length Inference. Computer Journal 45(3) (2002) 285–292

[8] Hudak, P. *et al*: Report on the Programming Language Haskell, version 1.2. Sigplan 27(5) (1992)

[9] Milne, R., Strachey, C.: A Theory of Programming Language Semantics. Chapman Hall, two volumes (1976)

[10] Milner, R., Tofte, M., Harper, R. M.: The Definition of Standard ML. MIT Press (1990)

[11] Oliver, J.: Decision Graphs - an Extension of Decision Trees. 4th Int. Conf. Artificial Intelligence and Statistics (1993) 343–350

[12] Peyton Jones, S. *et al*: Report on the Programming Language Haskell 98. http://www.haskell.org/ (1999-2003), also, Haskell 98 Language and Libraries, the Revised Report, Cambridge U. P. (2003)

[13] Quinlan, J. R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1992)

[14] Rissanen, J.: A Universal Prior for Integers and Estimation by Minimum Description Length. Annals of Statistics 11(2) (1983) 416–431

[15] Shannon, C. E.: A Mathematical Theory of Communication. Bell Syst. Technical Jrnl. 27 (1948) 379–423 and 623–656

[16] Stern, L., Allison, L., Coppel, R. L., Dix, T. I.: Discovering Patterns in Plasmodium Falciparum Genomic DNA. Molecular and Biochemical Parasitology 118(2) (2001) 175–186

[17] Venables, W. N., Ripley, B. D.: Modern Applied Statistics with S-PLUS. 3rd edn., Springer (1999)

[18] Wallace, C. S., Boulton, D. M.: An Information Measure for Classification. Computer Journal 11(2) (1968) 185–194

[19] Wallace, C. S., Freeman, P. R.: Estimation and Inference by Compact Coding. Journal of the Royal Statistical Society series B. 49(3) (1987) 240–265

[20] Wallace, C. S., Freeman, P. R.: Single-Factor Analysis by Minimum Message Length Estimation. J. Royal Stat. Soc. B 54(1) (1992) 195–209

[21] Wallace, C. S., Patrick, J. D.: Coding Decision Trees. Machine Learning 11 (1993) 7–22

[22] Witten, I. H., Frank, E.: Nuts and Bolts of Machine Learning Algorithms in Java. In Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations, Morgan Kaufmann (2000) 265–320