JFP **15** (1): 15–32, 2005.
DOI: 10.1017/S0956796804005301 Printed in the United Kingdom

Models for machine learning and data mining in functional programming

LLOYD ALLISON

School of Computer Science and Software Engineering, Monash University, Clayton, Victoria, Australia 3800
(*e-mail:* lloyd@bruce.cs.monash.edu.au)

Abstract

The functional programming language Haskell and its type system are used to define and analyse the nature of some problems and tools in machine learning and data mining. Data types and type-classes for statistical models are developed that allow models to be manipulated in a precise, type-safe and flexible way. The statistical models considered include probability distributions, mixture models, function-models, time-series, and classification- and function-model-trees. The aim is to improve ways of designing and programming with models, not only of applying them.

Capsule Review

This paper uses Haskell as a language to describe a family of statistical models based on the MML principle. The generalization of the various models, and the natural mappings between them, are shown clearly by the use of Haskell classes, types and functions. It will be interesting to see how far this idea extends and how many more concepts in machine learning can also be expressed in this framework in the future.

1 Introduction

Much research in machine learning and data mining is typified by creating some new and reasonably general statistical model for a certain range of problems, or devising a new search algorithm for a statistical model, or applying such a model or algorithm to a new problem, or more than one of the above. This paper instead examines different kinds of statistical models and operations on them and between them, functional programming being used as the analytical tool. Haskell-98 (Peyton Jones *et al*., 1999) is used here to express a range of models including univariate and multivariate probability distributions, mixture models, function-models (regressions), and time-series. Types and classes (in the Haskell sense) of statistical models and functions of models are given and are, in effect, succinct definitions of what models are and how they behave. This shows promise towards codifying some of what is known about probability and statistics, and allowing statistical models to be analysed and broken down into building blocks which can be reused in forming new models.

There are two main aims for this work. The first perhaps more important aim is the getting of understanding, i.e. making what could be called a theory of statistical models and modelling – in the sense that `PreludeList` (Peyton Jones *et al.*, 1999, A.1 p105) is a theory or model of linked-lists and list-processing. (This understanding is also helping to guide the design of a Java data-mining platform (Comley *et al.*, 2003).) A second, long-term aim is to build a toolkit of machine learning and data mining tools, in Haskell. Haskell-98 is used in the interests of standardization although some experiments, not described here, have been carried out using type extensions. Functional programming gains great power from treating functions as first-class values, and from clean notions of type and class. There exist general data-mining platforms, such as R (see CRAN), S-Plus (Venables & Ripley, 1999) and Weka (Witten & Frank, 1999), that give some support to programming with models. In some there are two ways to program – in a compiled language, such as C, and in an interpreted "scripting" language where the type-checking in particular tends to be dynamic and ad hoc. Here Haskell is used for all purposes; if models are treated as first-class typed values it is hoped that the power of functional programming will rub-off on them.

Statistical models appear in statistics obviously, in artificial intelligence, machine learning and data mining, and in specific application areas. Unfortunately different, and often conflicting, terminology can be used in these various fields. This is a real problem. In everyday use, the term statistical model can cover almost anything from a simple probability distribution to a model of the planet's climate. We cannot match this flexibility in computer programming but try to be general while also being precise enough to write programs and have them pass the type checker, compile and run. Here, *statistical model* is often used in place of distribution, parameter estimate, model, model class, hypothesis, or theory. We consider statistical models that assign probabilities to data. Making an inference system where models can be combined requires some way of addressing the overfitting problem; the minimum message length (MML) framework discussed in section 2.1 provides sufficient conditions.

There are many possible ways to represent statistical models in Haskell but only some fit with the aims of the paper: A library of large, independent machine-learning tools, coded in Haskell or in some other language and called through an interface, would add little understanding over existing machine-learning platforms. Some completely new specification language for statistical models and having its compiler or interpreter implemented in Haskell would gain little, except perhaps quick development, from that choice compared to any other implementation language. Of more interest, some kind of denotational semantics of statistical models would be precise and would help understanding; by analogy Mosses's (1974) mathematical semantics of Algol-60 can be viewed as a theory of Algol-60 and of programming in it. And a Haskell-98 prelude for statistical models would provide strong help for programming with models, in Haskell-98. As noted, `PreludeList` is a kind of theory or semantics of linked-lists and list-processing, a theory having a heritage of 40 plus years with some of the operators appearing in APL (Iverson, 1962) and Lisp (McCarthy *et al.*, 1962). Statistical models are at least as complex as lists so it may be a little time before their own "best" programming theory is developed. However

it is worth making a start and this paper presents a step towards a theory or model or semantics or prelude of statistical models and of programming with them. The types and type classes of models and of their operators are of particular interest for understanding what models are and how they behave or could behave. Some algorithms that implement important operators are presented to support a general claim to realism.

In the following sections, first the most important properties of the most basic models are examined; these are discussed in the minimum message length (MML) framework. To study the realistic problem of unsupervised classification (also known as clustering), mixture modelling and an associated search algorithm are defined. Function-models, which have input (independent, exogenous) attributes (variables) and output (dependent, endogenous) attributes, are then examined. Time-series are treated next and some natural functions on and between the various kinds of model are defined. Classification trees, also known as decision trees, and as used widely in supervised classification, are brought in as examples of function-models and are used to illustrate some relationships between the different kinds of statistical models.

2 (Basic) models

Q: What is the difference between a hypothesis and a theory?
A: Think of a hypothesis as a card. A theory is a house made of hypotheses.
(From rec.humor.funny, attributed to Marilyn vos Savant.)

The terms *distribution*, *statistical model*, *model class* (class in the statistical sense), *hypothesis* and *theory* are taken to be equivalent here except perhaps for indicating a degree of scale or ambition. Similarly, *estimating* a model's parameters, *fitting* a model, *choosing* a model class and *inferring* a theory are taken to be equivalent to the same extent. It is theories not theorems that are being discussed. The former are inferred from observations, from data, and are necessarily falsifiable by new observations whereas the latter are true – unless errors were made. The most essential property of a statistical model seems to be that it makes predictions. Even for a good model, its predictions are not expected to be infallibly correct; they are probabilistic. The most basic property of the most basic kind of model is that it can calculate the probability, `pr`, of a datum:

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

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

instance Model ModelType where
  pr (MPr ml p) datum = p datum
  ...
```

Function `pr` takes a (basic) model over some data-space, i.e. a `(mdl dataSpace)`, and a datum from the data-space, and returns a probability. A `Model` might very well be able to do other things – such as generate sample data or print itself – but here it must at least have `pr`. The data type `ModelType` is an instance of the `Model` class (class as in programming terminology); a model value can be constructed by giving `MPr` a message length and a probability function; the *message length* is a measure of the model's complexity and is described in the next section. Thus if `fairCoin` is a model of a throw (`H|T`) of a coin, which seems to be the natural way to describe it, then `pr fairCoin H = 0.5`.

The accompanying code at the web site does implement (Allison, 2003) the ability of statistical models to print (`Show`) themselves. This is very useful but not very interesting so, for simplicity, references to printing are deleted in code fragments in this paper.

2.1 Minimum Message Length (MML)

It is well known that over-fitting is a potential problem in machine learning: If models of increasing complexities are entertained, some kind of stopping criterion is needed so that simple and complex models are treated fairly. The need is even greater if models can be combined, calling for a *compositional* methodology that can easily assess the complexity of a combination from its parts. Minimum message length (MML) inference (Wallace & Boulton, 1968) is a Bayesian methodology that satisfies these requirements and is used in the present work. It seems likely that some other methodologies, such as the Akaike information criterion (Akaike, 1973), could be used in a similar way. Once the methodology is chosen, its terms naturally tend to appear in the definition of each basic model and structured model. It is even conceivable that the methodology could be abstracted out, particularly if restricted to giving numerical *costs* (or scores) to models and to composing costs (or scores) in simple ways (+, *, min, max, etc.). However at least one methodology would still have to be instantiated for programs to be usable. The present work sits in the MML framework which is introduced below.

MML is consistent, invariant under monotonic transformations of parameters, and resistant to overfitting. MML relies on Bayes's theorem (1763) and on Shannon's (1948) mathematical theory of communication – hence "message". In particular, Bayes gives the joint probability of a model (hypothesis, theory), M, and data, D, in terms of the *prior* probability of M, Pr(M), and the *likelihood*, Pr(D|M), or in terms of the probability of D, Pr(D), and the *posterior* probability of M given D, Pr(M|D).

```
Bayes:
  Pr(M&D) = Pr(M) . Pr(D|M) = Pr(D) . Pr(M|D)
```

Shannon (1948) showed that if an event E has probability Pr(E) then it has a code-word of length $-\log(\Pr(E))$ in an optimal code. Combining Bayes and Shannon, we can consider a two-part message, sending first a model M and then data D given M. In principle one could equivalently send D and then M given D but such codes are hard to construct.

```
Shannon:
  msgLen(E) = -log(Pr(E))

so:
  msgLen(M&D) = msgLen(M) + msgLen(D|M)
              = msgLen(D) + msgLen(M|D)
```

It is, of course, arranged that M is an answer to some relevant inference problem concerning D. The message length of the model itself, msgLen(M), is a measure of its complexity as mentioned before in connection with `MPr`, the constructor for `ModelType`; the units of measurement for messages are *bits* for logs to base 2, *nits* (or nats) for natural logs, and Alan Turing's *bans* for logs to base 10 (Hodges, 1983).

The MML paradigm considers a transmitter and a receiver. The transmitter and receiver agree on algorithms and codes for transmitting models and data. The transmitter and the receiver are then separated and the transmitter is given data to send to the receiver. The transmitter should naturally use a code that is based on a good model, M, of the data, D, but must first send any parameters describing M that are not common knowledge so that the receiver is able to decode the message. Continuous-valued parameters must be stated to an optimum accuracy: If excessive precision is used, the length of the first part of the message increases by more than any saving in the second part. If precision is too low, the stated values are, on average, far from ideal and the consequent increase in the length of the second part exceeds any savings on the first part.

The length of the first part of the message is msgLen(M); its inclusion allows simple and complex models to be compared fairly. The second part of the message transmits the data given M. A complex model is likely to fit the data better than a simple model but is only preferred if the *total* message length is shorter. Note that the difference in message lengths under two alternative models gives their posterior −log odds ratio:

```
msgLen(M|D) - msgLen(M'|D)
  = msgLen(M)+msgLen(D|M) - (msgLen(M')+msgLen(D|M'))
```

Although strict-MML (SMML) inference is computationally intractable for most interesting model spaces (Farr & Wallace, 2002), there are efficient MML approximations for many problems and models of practical importance (e.g. Wallace & Boulton, 1968; Wallace & Freeman, 1987).

MML is well suited to the composition of models because the units of measurement for a model and for a data set given a model are the same (although its programming potential has not been fully realized previously). In general, better inferences can be obtained if all components and parameters of a model are optimised together, but the search may then be intractable. If a model is composed of sub-models which are optimised individually the total solution is not necessarily optimal but, as noted by Georgeff & Wallace (1984), it will usually be close to optimal under reasonable conditions and the search is much easier. Such composition of models has previously been useful in special cases (e.g. Allison *et al.*, 1999) and is generalized here by having models as genuine first-class, typed values

that can be used in other models. One of the main reasons for using Haskell in the present exercise is its ability to support the specification and composition of models in a compact and precise way through polymorphic types (Milner, 1978), type classes and high-order functions.

Probabilities of data and of models can be very small for large data-spaces and model spaces, possibly causing underflow. It is often better to compute in terms of negative log probabilities and message lengths which are quite manageable numbers; adding probabilities can be effected by using logPlus, where logPlus(−log(p1), −log(p2)) = −log(p1+p2). However any discussion is in terms of probabilities or their negative logs as is most convenient.

2.2 Example models

Other classes (in the Haskell sense) of statistical models are defined in section 3 in addition to the class `Model` already defined. They all share some common properties, including having a prior probability and a first-part message, `msg1`, which are attached to a super class – obviously the class `SuperModel`. Given that, and given message lengths, class `Model` is revised:

```
class Model mdl where
  pr   :: (mdl dataSpace) -> dataSpace -> Probability    -- & its...
  nlPr :: (mdl dataSpace) -> dataSpace -> MessageLength  -- neg' log

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

  pr   m datum = exp (- nlPr m datum)
  nlPr m datum = - log (pr m datum)

  msg  m ds = (msg1 m) + (msg2 m ds)          -- total m & ds
  msg2 m ds = foldl (+) 0 (map (nlPr m) ds)  -- ds ~ data set
```

An instance of class `Model` must at least define `pr`, or `nlPr`, or both. The two-part message length, `msg`, of a `Model` and a data set, is the sum of part one, `msg1`, and part two, `msg2`.

Perhaps the simplest example of a model is a uniform distribution over a range of discrete values:

```
uniform lo hi =
  MPr 0 (\_ -> 1 / fromIntegral((fromEnum upb) - (fromEnum lwb) + 1))
```

The complexity of `uniform lo hi` is zero because `lo` and `hi` are taken to be common knowledge here as defined by the data-space. The probability function always returns the inverse of the number of values in the data-space.

The following is a more interesting model – over non-negative integers. It is based on a universal code that was introduced for classification trees (Wallace & Patrick,

1993) and assigns code-words of 1, 3, 5, 5, 7,... bits to successive integers:

```
wallaceIntModel =       -- of non-negative Ints
 let catalans = ...           -- 1 1 2 5 14 ...
     cumulativeCatalans = scanl1 (+) catalans
 in MnlPr 0 (\n -> ((find n 0 cumulativeCatalans)*2+1)*log2)
```

Lazy evaluation aids in specifying the Catalan numbers used in the model's definition.

A continuous probability distribution defines a probability *density*. This gives a genuine probability when combined with the data measurement accuracy. Present practice is to require this accuracy to be given as a parameter of the function that creates such a distribution; a resulting model then fits straight into the current system. A conceivable, and more complex, alternative is to create one or more type classes specifically for continuous models; this is still under consideration. Note that data measurement accuracy does (should) affect estimation in some circumstances. For example, it is unreasonable to infer a standard deviation much less than the data measurement accuracy.

It can seem natural to try to define a special data type for some model and data that are being used a lot, e.g.

```
data BiasedCoin = BiasedCoin Float  -- i.e. Pr(H)
```

However this `BiasedCoin` cannot be made an instance of our current class `Model` because it has the wrong *kind*. Such definitions can be made under type-extensions such as multi-parameter classes and variations (Jones, 2000) but, as noted, we aim to stay within standard Haskell-98 if possible.

2.3 Estimators

An estimator takes a data set, i.e. a list of values, and returns a statistical model of some kind. For example, a multi-state model can be estimated from a data set (list) of an enumerated (`Enum`), bounded data-space: The frequencies of the values in the data set are counted and used to estimate the probabilities. It so happens that the MML estimator adds 0.5 to each frequency before normalizing to give probabilities, so values of zero or 1.0 are never estimated; justification for this and details of the calculation of the model's complexity, `part1`, were given by Wallace & Boulton (1968). (In contrast, the maximum-likelihood estimator normalizes the raw frequencies.) An example value, `dataSet!!0`, is used to convert the model of a range of `Ints` into a model of the enumerated, bounded data-space (type):

```
freqs2model fs =
  let total = foldl (+) 0 fs
      ...
      part1 = ...  -- model's complexity
      p n   = ...  -- probability function
  in MPr part1 p
```

```
modelInt2model egValue intModel =
  let fromE x = fromEnum( x `asTypeOf` egValue )
      toInt x = (fromE x) - (fromE minBound)
      p datum = pr intModel (toInt datum)
  in MPr (msg1 intModel) p

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

The above estimators can be applied to simple games of chance. For example, a model of a coin can be estimated from observations and used to make predictions about the game:

```
data Throw = H | T deriving ...

biasedCoin = estMultiState [H, T, T, ... ]
```

The model `biasedCoin` has the data-space `Throw` and cannot be used with any other type of data.

2.4 Operations on models

Given two models and their data-spaces, a *bivariate* model of the product data-space can be formed. Similarly, a bivariate estimator can be formed from two estimators for models. It is sometimes useful to deal with *weighted* data (e.g. section 2.5), and weighted estimators are used in this example:

```
bivariate (m1, m2) =
  let nlp (d1, d2) = (nlPr m1 d1) + (nlPr m2 d2)
  in MnlPr ((msg1 m1) + (msg1 m2)) nlp

estBivariateWeighted (est1, est2) dataSet weights =
  let (ds1, ds2) = unzip dataSet
  in bivariate (est1 ds1 weights, est2 ds2 weights)
```

It is possible to define a trivariate model and estimator, and so on, and also a n-variate model given a *list* of models of the same type. `bivariate` and the like assume an absence of correlation between attributes. Correlation might be explained by a model of d1, say, plus a function-model (see section 3.1) of d2 conditional on d1. In principle a factor model (e.g. Wallace & Freeman, 1992) could also be defined.

2.5 Mixture modelling

"... considered as a biological phenomenon, aesthetic preferences stem from a predisposition among animals and men to seek out experiences through which they may *learn to classify* the objects in the world about them. Beautiful 'structures' in nature or in art are those which facilitate the task of classification by presenting evidence of the 'taxonomic' relations between things in a way which is informative and easy to grasp." (Humphrey, 1973, p. 432).

A *mixture model* is a weighted average of a number of component models (e.g. Wallace & Boulton, 1968; Figueiredo & Jain, 2002). Mixture models are used in unsupervised classification, also known as clustering, where a component is sometimes called a class but that is too dangerous a word in this context. Mixtures can also be made of function-models and of time-series, as described later, so it is most generally an operation on `SuperModels`. It is convenient to define a `Mixture` class and a `MixtureType` data type. The weights in a mixture are, of course, equivalent to an integer model:

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

  prior sm = exp (- msg1 sm)
  msg1  sm = - logBase 2 (prior sm)

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 Mixture MixtureType where ...

instance (SuperModel elt) => SuperModel (MixtureType elt) where ...
```

For example, a mixture of a fair coin and a biased coin is a less biased coin. Note that it is possible for the components of a mixture to be different kinds of model, e.g. geometric and Poisson distributions, provided that they are all models of the same data-space. Their types must match.

The unsupervised classification problem is: Given univariate or multivariate data, find a mixture model that best describes the data. The number of components and the parameters of the components are not known in advance; they must be inferred.

For a component of a mixture model and a datum, the probability of the datum being a *member* of the component is proportional to the probability of the component times the probability of the datum given the component. The mixture's `mixer` gives the first factor and the component itself gives the second factor of the product. This computation allows data to be assigned (fractional) memberships of the components. Note that this method of *fractional assignment* of data to components gives an unbiased estimator whereas total assignment gives a biased one in general. A new mixture model can be estimated from the memberships: The weight of a component is proportional to the sum of the memberships of the data with respect to that component. The parameters of a component can be estimated from the data weighted by their memberships of the component. So, a mixture model can be estimated from memberships, and memberships can be estimated from a mixture,

```
estMixture ests dataSet =
  let
    memberships (Mix mixer components)
      = ... as discussed

    randomMemberships
      = ... straightforward

    fit [] [] = []
    fit (est:ests) (mem:mems)
      = (est dataSet mem) : (fit ests mems)

    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))
```

Fig. 1. Estimator for a mixture model.

and so on. The process is started by assigning the data *random* memberships and continues for several cycles. It must converge, possibly to a local optimum. This algorithm is an instance of the *expectation maximization* (EM) paradigm and is the most important part of the main loop in the "Snob" mixture modelling program (Wallace and Boulton 1968). That program has additional heuristics, not implemented or described here, to merge and split classes in the search for an optimal mixture. The basic EM loop described is sufficient for present purposes and given a list of estimators, one per component, yields an estimator for a mixture of models; see Figure 1. A search through increasing numbers of components, to some limit, will in principle find the best mixture.

As an example, consider a game played with a silver coin and a gold coin. A bivariate estimator, `est2coins`, can be formed from the multi-state estimator. This can be used with `estMixture` to fit a mixture model of two or more components to some observations of the game. Foul play is detected if a mixture model with two or more components describes the observations better than a one-component model, or if a one-component model shows a clear bias:

```
est2coins = estBivariateWeighted
              (estMultiStateWeighted, estMultiStateWeighted)

mix2  = estMixture [est2coins, est2coins] observations
```

It might be argued that `estMixture` only estimates the weights in the mixture model and that it relies on parameters to estimate the component models. This is

true, but the estimators for common models (distributions) such as the multistate and the normal are not difficult. Relying on parameters here is in fact a good thing because it allows new and improved estimators to be used with the existing code, e.g. perhaps a factor-analysis model.

3 More models

So far only the most basic kind of model has been examined. Next function-models and time-series are added. Taken together, these sub-classes of `SuperModel` describe not all but a good range of statistical models. It seems that there are no useful pure `SuperModels`; to be useful they need to have another job as a basic model, function-model, time-series or some other sub-class to be defined. For this reason the name `Model` seems too valuable to waste on the super class.

3.1 Function-models

A function-model describes the relationship between input (independent, exogenous) attributes (variables) and output (dependent, endogenous) attributes. This notion includes many regressions, polynomial fitting and supervised learning problems. A function-model makes a prediction of output attributes but it is *conditional* on the input attributes, i.e. a conditional model, `condModel`, of the inputs and the `FunctionModel`:

```
class FunctionModel fm where
  condModel :: (fm inSpace opSpace) -> inSpace -> ModelType opSpace
  condPr    :: (fm inSpace opSpace) -> inSpace -> opSpace
               -> Probability
  condNlPr  :: (fm inSpace opSpace) -> inSpace -> opSpace
               -> MessageLength

  condPr   fm i o = pr   (condModel fm i) o  -- Pr(o|fm,i)...
  condNlPr fm i o = nlPr (condModel fm i) o  -- & neg log

data FunctionModelType inSpace opSpace =
  FM MessageLength (inSpace -> ModelType opSpace)

instance SuperModel (FunctionModelType inSpace opSpace) where
  msg1 (FM mdlLen m) = mdlLen
  mixture mx =
    let condM inp = mixture (Mix (mixer mx)
                    (map (\f -> condModel f inp) (components mx)))
    in FM (msg1 mx) condM

instance FunctionModel FunctionModelType where
  condModel (FM mdlLen f) = f
```

The data type `FunctionModelType` is an instance of a `FunctionModel`; a value is specified by giving the message length and the conditional model function. A mixture of function-models is defined in terms of the mixture of the conditional models.

Functions of finite spaces form an important special case. If the input space is finite and if no correlation between cases is assumed, a separate model of the output space can be estimated for each input case. If the output space is itself finite, the multi-state model is the obvious candidate model of the output space. This yields conditional probability tables, for example, and can also form the basis of a function-model on lists of a finite element-type, and of length less than or equal to k:

```
estFiniteIpFunction estOpModel ipSeries opSeries = ...

estFiniteFunction ipSeries opSeries =
  estFiniteIpFunction estMultiState ipSeries opSeries

estFiniteListFunction k inputs outputs = ...
```

3.2 Time-series

A time-series is a statistical model of sequences of values or events. It makes a sequence of predictions, each one for a position given the preceding values which could literally depend on time or could, for example, come from a biological sequence (Stern *et al*., 2001). Each of the predictions is a model, conditional on the time-series and on the *context* of previous values, and this gives the types of `predictors`, `prs` (for probabilities) and `nlPrs` (negative log prs):

```
class TimeSeries tsm where
 predictors:: (tsm dataSpace) -> [dataSpace] -> [ModelType dataSpace]
 prs       :: (tsm dataSpace) -> [dataSpace] -> [Probability]
 nlPrs     :: (tsm dataSpace) -> [dataSpace] -> [MessageLength]
```

Note that `predictors` returns one more result than there are elements in the data series; the extra prediction is for the next element.

One natural way to create a time-series is to give its message length and its function for making a prediction from previous values; see `TimeSeriesType` and constructor `TSM` below. (Another natural way to create a `TimeSeries` type could be from a "stateful" function.)

```
data TimeSeriesType dataSpace =
  TSM MessageLength ([dataSpace] -> ModelType dataSpace)

instance SuperModel (TimeSeriesType dataSpace) where
  msg1 (TSM mdlLen m) = mdlLen
  mixture mx =
    let f context =
          mixture(Mix (mixer mx)
            (map (\(TSM _ ftsm) -> ftsm context) (components mx)))
    in TSM (msg1 mx) f
```

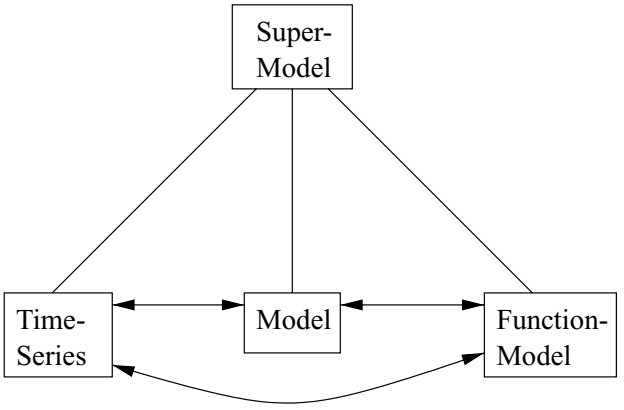

Fig. 2. Classes and conversion functions.

```
instance TimeSeries TimeSeriesType where
  predictors (TSM mdlLen f) dataSeries =
    let scan  []    context = [f context]
        scan (d:ds) context = (f context) : (scan ds (d:context))
    in scan dataSeries []
```

A mixture of two or more time-series is made via a mixture of the models predicted. Note that it is convenient to have the context of past values in reverse order, from most recent to least recent, because of the way that the values are scanned and because many time-series models pay more attention to recent values. For example, a Markov model of order k can be estimated by fitting a finite list function to contexts derived from a training data series:

```
estMarkov k dataSeries =
  let scan (d:ds) context = context : (scan ds (d:context))
      scan []     context = []
      contexts = scan dataSeries []
  in functionModel2timeSeries
       (estFiniteListFunction k contexts dataSeries)
```

Note that `functionModel2timeSeries` converts a function-model, of the appropriate type, into a time-series. More of these conversion functions are discussed in the next section.

3.3 Mappings between models

There are some natural conversion functions between (basic) models, function-models and time-series (see Figure 2). A model can be converted into a trivial function-model and into a time-series, by ignoring the input attributes and the context respectively. A time-series over a data-space, t, can be converted into a model of sequences (lists) of t, by summing the negative log probabilities of (i) the

elements of the list, and (ii) the length of the list using, say, `wallaceIntModel` of section 2.2 unless domain knowledge suggests a different choice. A time-series can be turned into a function-model of input space, [t], and of output space, t, by extracting the last prediction of the time-series on the list, that is by extracting the prediction for the next value; laziness is useful here. A function-model of an input space and an output space can be converted into a model of their product space by using its conditional prediction; this assumes that the values of the input attributes are common knowledge, of zero cost, which is the case in supervised problems. And, recalling the Markov model of the previous section, a function-model of input space, [t], and of output space t, can be converted into a time-series in the obvious way:

```
model2functionModel m = FM (msg1 m) (\ip -> m)

model2timeSeries m = TSM (msg1 m) (\context -> m)

timeSeries2model tsm = ...

timeSeries2functionModel tsm =
  FM (msg1 tsm) (\dataSeries -> last(predictors tsm dataSeries))

functionModel2model fm = MnlPr (msg1 fm) (\(i,o) -> condNlPr fm i o)

functionModel2timeSeries fm = TSM (msg1 fm) (condModel fm)
```

A function-model can be conditioned by a model of its input space to give a model of the product of its input and output spaces:

```
condition m fm =
  let nlp (i,o) = (nlPr m i) + (nlPr (condModel fm i) o)
  in MnlPr (msg1 m + msg1 fm) nlp
```

Conversion functions are more than curiosities. Their existence, and the existence of similar conversion functions on estimators, increases the generality of the whole collection of models, functions, types and classes. For example, the mixture modeller, `estMixture` (Figure 1), requires estimators for component models. Estimators for function-models can be converted into estimators for models and used with `estMixture` to infer mixtures of function-models, and so on.

3.4 Classification (decision) trees

The supervised classification problem is: Given a data set of items that have been classified, e.g. by an expert, learn a function-model to predict (model) the category of a new item. Classification trees, sometimes known as decision trees, are popular for this problem with C4.5 and its relatives (Quinlan, 1992) being perhaps the best known.

A classification tree of `CTreeType` consists of leaf and fork nodes. Each leaf, `CTleaf`, contains a model of the output attribute(s). Each fork node tests one (usually) input attribute in order to select a subtree. A traditional fork, `CTfork`, contains a selector or partitioning function with its message length, and a list of subtrees. It is convenient to create a data type `Splitter` for a partitioning function and some associated information. The function can be thought of as an "extreme" function-model of the input space and Ints, one that places *all* of its probability on the index of one subtree. This immediately suggests a more general kind of fork node, `CTforkFM`, which contains the pieces for a mixture, capable of blending predictions from the subtrees rather than just picking one of them. Note that it is likely that many tree estimators will return trees containing leaves and either only traditional or only general fork nodes. It *might* therefore be better to have two data types of trees, but having both kinds of fork in the one data type gives the potential for a combination to be used; the jury is still undecided. Also note that the structure of a tree is included as part of its message length calculations; the details (Wallace & Patrick, 1993) are interesting but not relevant to this paper.

```
data CTreeType inSpace opSpace =
       CTleaf (ModelType opSpace)                |
       CTfork MessageLength (Splitter inSpace)
              [CTreeType inSpace opSpace]        |
       CTforkFM (FunctionModelType inSpace Int)
                [CTreeType inSpace opSpace]

instance SuperModel (CTreeType inSpace opSpace) where ...

instance FunctionModel CTreeType where
  condModel (CTleaf leafModel) i = leafModel
  ... etc.
```

These classification trees are already more general than those of C4.5, allowing *any* kind of model, not just univariate discrete ones, in leaves. As another example, let `linear a b epsilon` be a linear function-model where y=a.x+b+(normal 0 epsilon), and `normal m s` is a normal model with mean m and standard-deviation s. Then `regTree`, below, is a function-model-tree, i.e. a regression-tree.

```
-- linear a b epsilon  is a FunctionModel of Float Float
linear a b epsilon = ...

regTree = CTfork (...) [leaf0, ... ]
leaf0 = CTleaf (functionModel2model (linear 1 2 1))
```

An estimator function for a classification tree requires an estimator function for leaf models, ways of partitioning or *splitting* the data set into parts to pass to subtrees, and a training set of inputs and outputs. (In this case alternative splits are provided by a function of the input set which returns a list, possibly empty, of partitioning functions. This seems to be a reasonable approach because data

```
estCTree  estLeafMdl splits  ipSet opSet =
 let search ipSet opSet =
      let leaf    = CTleaf leafMdl
          leafMdl = estLeafMdl opSet
          leafMsg = msg (functionModel2model leaf) (zip ipSet opSet)

          ...
          alternatives ... =
           let theTree  = CTfork ... ... leaves
               leaves   = map CTleaf leafMdls
               leafMdls = map estLeafMdl opParts
               ...
           in ...return the best...

      in case alternatives (splits ipSet) leafMsg leaf [ipSet] [opSet] of
           ((CTfork msgLen pf leaves), ipParts, opParts)
                -> CTfork msgLen pf (zipWith search ipParts opParts);
           (t, _, _) -> t   -- done

 in search ipSet opSet
```

Fig. 3. Estimator for classification tree.

sets shrink as the tree is descended.) The simplest tree consists of a single leaf, and gives a certain message length (`msg`, part one plus part two) to the data. The alternative is to split the data set in some way, i.e. a fork node. The simplest, *zero-lookahead* search first considers fork nodes with immediate leaves. The best alternative out of the single leaf and all possible one-fork trees is selected. If it is the single leaf the search is done, otherwise the search is repeated recursively for each subtree with the appropriate *part* of the training data (see Figure 3). It is usual to do an n-way split on an enumerated, bounded attribute of range n, and to try binary splits, at data-dependent cut-points, on a continuous-valued attribute. The splits on multivariate data include at least the splits on each attribute individually. The set of splits tried can in principle be generalized to include any other partitioning function that divides the data into at least two parts but the search problem gets worse, of course. The search selects the best alternative, subject to the restricted lookahead, on the basis of the complexity of the tree and its fit to the data.

It is even possible to make the `splits` parameter implicit: A class `Splits` having a partitioning operator on lists is defined. Float, any bounded enumerated type, and tuples `(,)` of `Splits` types can be added to the class. The jury is out as to whether this is the best method.

The tree estimator, `estCTree` (see Figure 3), also estimates a function-model-tree (regression-tree) with the aid of a simple conversion function; for example see tree `fmt` below. An estimator for a suitable leaf function-model is converted to an estimator for a corresponding model by `estFunctionModel2estModel` so that it

can be used with `estCTree`:

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

fmt = estCTree
      (estFunctionModel2estModel someFunctionModelEstimator)
      splits set1 set2
```

The data in `set1`, i.e. the inputs to the tree, and in `set2`, i.e. the input-output pairs for the function-model, can share common attributes or not as appropriate provided that care is taken to account, once, for all that is not common knowledge.

4 Conclusions

Types, classes, and functions have been defined for (basic) models (2), function-models (3.1), time-series (3.2), mixtures (2.5), classification trees (3.4), estimators, and mappings between models, function-models and time-series (3.3). These do not cover all the statistical models that are used in machine learning and data mining, but they make a good start. Basic but usable estimators for unsupervised classification, supervised classification, time-series and sequence analysis have been presented.

The models, functions and estimators given have static, polymorphic types as in any other Haskell program. It is a natural and useful consequence of using a programming language to give a treatment of statistical models that the "theory" or "model" compiles and runs. Polymorphic typing, type classes and the conversion functions defined above bring out the generality of models. The mere fact of having estimators and models defined in a compact, polymorphic and usable form has suggested unanticipated generalizations. For example, the relationship between classification trees and regression trees then became obvious.

The most valuable feature of Haskell-98 for this application is its type system. High-order functions follow closely for defining estimators, conversion functions, and functions that combine models to build new ones. Lazy evaluation is helpful in some algorithms and definitions, e.g. `wallaceIntModel` (2.2), and in some manipulations of time-series. Type-classes are very important but some question remains over whether or not it would be worth using type extensions outside the standard language.

Acknowledgements

Thanks to the members of the Central Inductive Agency in Computer Science and Software Engineering at Monash, particularly Chris Wallace, Leigh Fitzgibbon and Josh Comley, who have provided much inspiration.

References

Akaike, H. (1973) Information theory and an extension of the maximum likelihood principle. In: Petrov, B. N. and Csaki, F. (eds.), *Proc. 2nd Int. Symp. on Information Theory*, pp. 267–281.

Allison, L. (2003) Inductive Inference 1. *TR 2003/148*, School of Computer Science and Software Engineering, Monash University.

Allison, L., Powell, D. and Dix, T. I. (1999) Compression and approximate matching. *BCS Comput. J.*, **42**(1), 1–10.

Bayes, T. (1763) An essay towards solving a problem in the doctrine of chances. *Phil. Trans. Roy. Soc. Lond*, **53**, 370–418. (Reprinted in *Biometrika*, **45**(3/4), 296–315, 1958.)

Comley, J. W., Allison, L. and Fitzgibbon, L. J. (2003) Flexible decision trees in a general data-mining environment. *4th Int. Conf. on Intelligent Data Engineering and Automated Learning (IDEAL-2003): LNCS 2690*, pp. 761–767. Hong Kong, Springer-Verlag.

CRAN: The comprehensive R archive network. `http://lib.stat.cmu.edu/R/CRAN/` (current 2003)

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

Figueiredo, M. A. T. and Jain, A. K. (2002) Unsupervised learning of finite mixture models. *IEEE Trans. Patt. Anal. Machine Intell.* **24**(3), 381–396.

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

Hodges, A. (1983) *Alan Turing: The Enigma*. Simon and Schuster.

Humphrey, N. K. (1973) The illusion of beauty. *Perception*, **2**, 429–439.

Iverson, K. E. (1962) *A Programming Language*. Wiley.

Jones, M. P. (2000) Type classes with functional dependencies. *Proc. 9th European Symp. on Prog. (ESOP 2000): LNCS 1782*, pp. 230–244. Springer-Verlag.

McCarthy, J., Abrahams, P. W., Hart, T. P. and Levin, M. I. (1962) *The Lisp 1.5 Programmer's Manual*. MIT Press.

Milner, R. (1978) A theory of type polymorphism in programming. *J. Comput. & Sys. Sci.* **17**, 348–375.

Mosses, P. (1974) The mathematical semantics of Algol 60. *PRG-12*, Programming Research Group, Oxford University.

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

Quinlan, J. R. (1992) *C4.5: Programs for machine learning*. Morgan Kaufmann.

Shannon, C. E. (1948) A mathematical theory of communication, *Bell Syst. Tech. J.* **27**, 379–423, 623–656.

Stern, L., Allison, L., Coppel, R. L. and Dix, T. I. (2001) Discovering patterns in Plasmodium falciparum genomic DNA. *Molecular & Biochem. Parasitology*, **118**(2), 175–186.

Venables, W. N. and Ripley, B. D. (1999) *Modern Applied Statistics with S-PLUS*. 3rd edn. Springer.

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

Wallace, C. S. and Freeman, P. R. (1987) Estimation and inference by compact coding. *J. Roy. Stat. Soc. B.* **49**(3), 240–265.

Wallace, C. S. and Freeman, P. R. (1992) Single-factor analysis by minimum message length estimation. *J. Roy. Stat. Soc. B.* **54**(1), 195–209.

Wallace, C. S. and Patrick, J. D. (1993) Coding decision trees. *Machine Learning*, **11**(2), 7–22.

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