Model Classes

Lloyd Allison
School of Computer Science and Software Engineering,
Monash University,
Clayton,
Victoria,
Australia 3168.
http://www.csse.monash.edu.au/~lloyd/
Technical Report 2002/125 [*]

4 November 2002

Abstract

Statistical models are used, explicitly or implicitly, in artificial intelligence, data mining, machine learning and statistics (of course), and in specific application areas. Much associated research is about devising a new kind of model, finding a better optimization algorithm for a model, or applying a model to a problem area. This paper instead examines the semantics of statistical models – what they are precisely, how they behave *in general*, and how they can be combined. The programming language Haskell-98, with its polymorphic type system, is the tool for the exercise so one of the products is a running program. Data types and type classes, that allow models to be manipulated in a type-safe yet flexible way are developed for probability distributions, function-models (regressions), time-series, estimators, unsupervised classification, mixture models, supervised classification, and classification trees. It is shown that there are short and convenient conversion functions between various kinds of models and between their estimators. The result is a collection of useful tools in Haskell and, perhaps more importantly, the start of a theory of programming with models.

Keywords: Classes, data mining, types, functional programming, Haskell, machine learning, minimum message length, MML, MMLFP, types.

[*] peer communications and specific requests only (1st draft 14/10/2002).

"... 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 1972 p432).

1 Introduction

Statistical models appear in artificial intelligence, data mining, machine learning and statistics (of course), and in many application areas. Sometimes the appearance is explicit and sometimes it is implicit, e.g. the "weights" and "costs" used in some algorithms for aligning DNA and protein sequences in bioinformatics can be *normalized* to reveal the underlying statistical model (Allison 1993). For our purposes, probability distributions, statistical models, model classes, hypotheses and theories are much the same, except in ranging from the modest to the ambitious. Common activities are fitting the parameters of a model to data, inferring a model from data, selecting a model class (class as in statistics), and comparing models.

In this paper, the behaviour of statistical models is examined and made precise in a programming sense. Types and classes, and operations on instances of them are defined allowing statistical models to be combined in flexible ways while also allowing these combinations to be rigorously type-checked for validity. This can be viewed as a first step towards a semantics of statistical models, as denotational semantics (Milne and Strachey 1976) provided for programming languages. The functional programming language Haskell-98 (Peyton-Jones *et al.* 1999) is used as the meta-language for the exercise.

General inspiration is taken from the field of functional programming (FP) which achieves great expressive power from treating functions as first-class values with all that follows from that decision. It is hoped that treating statistical models as first-class values will bring similar benefits to the use of models in machine learning and data mining, and to programming with models. Using models as *parameters* of other models has been very useful in specific cases (Allison *et al.* 1999) and we seek to formalize and generalize this. Polymorphic types (Milner 1978), as found in Standard ML (SML), Haskell and other functional programming languages, give the precision of strong static typing with the flexibility and generality of type parameters. It turns out that many models and associated operations are naturally polymorphic. Haskell's type classes here allow the behaviour of different kinds of models to be specified in general but precise terms.

Much research in machine learning and data mining consists of developing and evaluating some new kind of statistical model, or a new algorithm to infer a specific kind of model for a certain range of problem, e.g. mixture models for unsupervised classification (clustering) of multivariate data, or support vector machines (SVMs) for supervised classification, and so on. In contrast, here we are mainly concerned with questions such as "what is the behaviour of classification functions in general?", classification trees (say) being just one way of implementating such functions. The former activity often leads to stand-alone programs that use shell-scripts and intermediate files if and when they cooperate on data exchange, and that are difficult to use as building blocks in any sophisticated way. There are some more general data-mining *platforms*, such as R (see CRAN), S-Plus (Venables and Ripley 1999) and Weka (Witten and Frank 2000), which allow data analysis techniques to cooperate and which support the addition of new techniques, but any typing is typically dynamic and ad-hoc, compromising speed, or type-safety, or both. Here in contrast static, polymorphic types and operations (Strachey 1967) are developed; many operations on statistical models are naturally polymorphic. As the formal study of syntax led to good notation for syntax, tools to process syntax definitions (parser generators), and better syntax for programming languages, so it is hoped that the present study will have beneficial effects on the definition and use of statistical models with computers.

2 FP: Haskell, Types and Classes

Haskell (Hudak *et al.* 1992), and specifically Haskell-98 (Peyton Jones *et al.* 1999), is used here as the programming language, i.e. the *meta-language*, to describe our types and classes of statistical models. Haskell is a pure functional programming (FP) language having a polymorphic type system (Milner 1978) of the general kind first implemented in Standard ML (SML). The most important feature is that a structured data type can take one or more *type variables* as parameters. For example, `[t]`, that is `List` of `t`, represents the type of *all* `Lists` – `Lists` over any element type, `t`. The type variable, `t`, can be *instantiated* in many ways, hence *polymorphic*, e.g. `["MML", "FP"] :: [String]`, and `[factorial, fibonacci] :: [Int → Int]`, where → denotes a function type and `::` denotes "has the type".

Haskell has a system of type *classes*. A type class is defined by the values and operations (functions) associated with it – resembling a Java interface. A data type can be an *instance* of, i.e. can belong to, one or more type classes. For example, `List` is an instance of the classes `Show`, that is printable things, and `Functor`.

Haskell also has a *type inference* algorithm which automatically infers the most general types of most expressions in a program without the programmer having to explicitly give types. Type systems of this kind have most of the advantages of both dynamically and statically typed languages.

The standard library function `map` makes a good example. It is also known as "apply to all" because it applies a function `f` to all the elements, if any, of a list. It is defined "by cases" below; case one is for the empty list `[]`, and case two is for a non-empty list with head `x` and tail `xs`. The result list is empty if the input list is empty. Otherwise, `f` is applied to the first element, `x`, and `map f` is applied to the rest of the list; the list constructor, "`:`", constructs the result list.

```
map f []     = []
map f (x:xs) = (f x):(map f xs)
```

The programmer need not specify any types for `map`. The compiler's type-inference algorithm can detect that the function `f` must be applicable to the elements of the list, i.e. if `f::t→u` then `x::t`, `xs::[t]` and `map :: (t→u) → [t] → [u]`. Thus `map odd [1, 2, 3]` and `map isDigit ['a', '2', 'c']` are valid and `map isDigit [1, 2, 3]` gives a type error because the elements of the list are numbers not characters.

The programmer can define types in various ways, algebraic *data types* being used later to define types of statistical models.

Anonymous functions are defined using "\" to represent `lambda`, as in the lambda-calculus, and "→" to separate the formal parameter from the result expression, e.g. `(\x→x+1)` is the successor function.

Finally, Haskell is a *lazy* functional language; a value is only evaluated when (if) it is needed. Laziness allows infinite structures to be defined, provided that only finite parts are evaluated, and this simplifies many definitions. For example the list of *all* positive integers can be defined and computed with – provided only a finite part is used.

```
posInts = 1 : (map (\x -> x+1) posInts)
```

3 (Basic) Models

The term *statistical model* appears in many varied, flexible, creative and often contradictory ways in everyday usage depending on the context. We cannot hope to match this flexibility in programming which must necessarily be more restricted on one hand and more precise on the other. This section examines the most basic kind of statistical models including, but not limited to, common probability distributions.

The most basic operation that (many) models support is assigning a probability to a datum from a model's data space. A model might be able to do many other things but it must at least be able to do that. The Haskell code below specifies that a data type, `mdl`, is in the class `Model` (i.e. is a `Model`) if `mdl` has a type parameter, `dataSpace`, and if an operator, `pr`, is defined which takes a value of type `(mdl dataSpace)` and a datum of type `dataSpace` and produces a probability.

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

In order to create some *values* that are `Models`, one or more data types must be defined to implement them. For example, a value of a suitable `ModelType` is specified by a constructor, `MPr`, which takes two parameters. The first parameter is the message length (a measure of complexity) of the `Model` itself; message lengths are discussed in the next section. The second parameter is the well-known *likelihood* function which returns the probability of a datum given the `Model`.

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

Another constructor for `ModelType` is introduced later. `ModelType` is made an instance of class `Model` by defining the necessary function(s), in this case use the likelihood function to calculate a probability:

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

Thus if `n01 :: ModelType Float` is defined to be `(normal 0 1)`, that is a normal distribution with mean 0 and standard deviation 1 then `pr n01 0 = 0.399`. It is natural to write "`n01` is a `Model` of `Float`" although it is more correct to write "`n01` has the data type `(ModelType Float)` and the data type `ModelType` is an instance of the class `Model`".

4 MMLFP = MML + FP

Overfitting is a well known problem in inductive inference. A more complex model (e.g. a cubic polynomial) tends to fit training data better than a simpler model (e.g. a quadratic) but this may not generalise to test data. Various ways of taking the complexity of a model into account have been proposed as a cure for overfitting. The author's inclination is to use *minimum message length* (MML) inference (Wallace and Boulton 1968, Wallace and Freeman 1987). Just as the use of Haskell is well motivated but somewhat arbitrary, so the adoption of the particular framework, MML, is not central to the theme of this paper, although it does fit in very well, and the work has been done in that framework so it is briefly introduced here.

The MML criterion is to use the length of a *two-part message* to select between competing hypotheses (probability distributions, models, model classes, theories). It relies on Bayes theorem, that for a hypothesis H and data D

```
pr(H & D) = pr(H).pr(D|H) = pr(D).pr(H|D)
```

and on Shannon's mathematical theory of communication (1948), particularly that for an event, E, of probability pr(E) we have

```
msg(E) = - log(pr(E))
```

where msg(E) is the *message length* of a code-word for E in an optimal code. From Bayes and Shannon we get

```
msg(H & D) = msg(H) + msg(D|H) = msg(D) + msg(H|D)
```

In many problems pr(D) is unknown, but it is often possible to give a reasonable estimate for the *prior* probability of H, p(H), and for the likelihood, pr(D|H), and hence their negative logs.

The first part of the message, msg(H), describes an instance of a hypothesis, including any parameters stated to *optimum accuracy*; it corresponds to a header in terms of file compression. The second part of the message, msg(D|H), states the data given the hypothesis. The difference between message lengths under two hypotheses gives the posterior negative log odds ratio of the hypotheses.

MML is *invariant* under monotonic transformations of parameters and is *consistent*. Although strict MML (SMML) inference is NP-hard for most interesting model-spaces (Farr and Wallace 2002), efficient MML approximations and inference algorithms exist (Wallace and Freeman 1987) for many important practical problems.

MML is well suited to the composition of statistical models (as are some but not all other criteria) because the information content of data, model and submodels are all measured in the same units: "[It is possible] to use [message] length to select among competing sub-theories at some low level of abstraction, which in turn can form the basis (i.e., the 'data') for theories at a higher level of abstraction. There is no guarantee that such an approach will lead to the best global theory, but it is reasonable to expect in most natural domains that the resulting global theory will at least be near-optimal." (Wallace and Georgeff 1983).

The prior probabilities of complex models and the likelihoods of large data values are typically very small, perhaps small enough to cause underflow. As a practical matter, so that the resulting programs can run on non-trivial data, it is often better to work with negative log probabilities – message lengths – rather than probabilities themselves. Class `Model` is revised:

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

  msg2 :: (mdl dataSpace) -> dataSpace -> MessageLength

  msg  :: (mdl dataSpace) -> dataSpace -> MessageLength
  ...

  msg  m d = (msg1 m) + (msg2 m d)
```

The name `msg2` is used to indicate the *second* part of a two-part message, that is the part for a datum given the `Model`. The first part of the message, `msg1`, is defined later. The total message length, `msg`, for `Model` and data, is the sum of both parts.

Note that message lengths can be combined, without conversion to probabilities, by using `logPlus` where `logPlus(-log(p1), -log(p2)) = -log(p1+p2)`. Note that `logPlus(m,m) == m-1`.

5 Some Models and Operations

It is now possible to define `normal`, the function which given a mean and a standard deviation produces a fully-parameterized normal `Model` (probability distribution). Here this happens to be done using the second constructor, `MMsg`, of `ModelType`. `MMsg` takes the message length (complexity) of the `Model` and the function for the negative log likelihood, i.e. the message length of a datum given the `Model`:

```
normal m s =
  let constPart = (log(2 * pi)) / 2 + (log s)
      nll x = (constPart + (((x-m)/s)**2)/2) / log 2
  in MMsg 0 nll   -- Model of Float
```

The `Model`'s complexity is zero in this case because the parameters are taken to be constants, common knowledge. A `Model` of `Float` produces a probability density rather than a probability as such, but fortunately the measurement-accuracy of the data "passes through" the MML calculations under reasonable conditions, and the distinction can be ignored.

A list of probabilities can be used to form a `Model` of `Int`:

```
probs2model ps = MPr 0 (\n -> ps !! n )
```

(Note, the operator `!!` selects the nth element of a List.) For example,

```
fairDice = probs2model [1/6,1/6,1/6,1/6,1/6,1/6]
```

is the `Model` of rolls for a fair, computer-science dice, 0..5. A list of frequencies can also be used to form a `Model` of `Int`:

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

The calculation for the complexity of the `Model`, i.e. `part1`, and the MML estimator, as used in `p`, are given in Wallace and Boulton (1968). The precise details do not matter here, but the complexity increases (the uncertainty region for the parameter estimate shrinks) as the given frequencies (i) increase, or (ii) become more biased, or (iii) both of the above.

A `Model` of a discrete, enumerated (`Enum`), `Bounded` data space can be formed from a `Model` of the appropriate subrange of `Int`:

```
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


data Throw = H | T  deriving ...

fairCoin = modelInt2model H (probs2model [0.5,0.5])
```

`Throw` is the data type of one throw of a coin and `fairCoin` is a `Model` of `Throw`. Note that an example value, `H` above, is given to inform the type checker of the data space. (`Enum` and `Bounded` are standard classes in Haskell.)

A `Model` of a discrete, enumerated (`Enum`), `Bounded` type can be inferred given a sample (training data set) from a data space by counting the frequencies of the values in the data set:

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

The values in the data set are counted and their frequencies used to produce a `Model` of Int. This is turned into a `Model` of `dataSpace` where `(dataSet !! 0)` is used as the example value from `dataSpace` merely to inform the type inference algorithm. Note that, thanks to laziness, the example value need not even exist provided that its type can be inferred as though it does exist.

Given two `Model`s, `m1` and `m2`, a *bivariate* `Model` can be formed:

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

It is sometimes useful to deal with *weighted* data and to have an estimator that works from a data set together with a corresponding series of weights. Weights allow a repetitive data set to be compacted. They are also needed later in mixture modelling as described later. Given (weighted) estimator functions for a `Model` of `dataSpace1` and for a `Model` of `dataSpace2`, an estimator for a bivariate `Model` of `(dataSpace1, dataSpace2)` is given by:

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

The bivariate data are separated (`unzip`) and each column is fed, with the weights in this case, to the appropriate estimator and finally the bivariate `Model` is formed.

Obviously trivariate operations and so on can be created. It is also straightforward to form similar operations on n-ary lists of `Models` of the same data space, and their estimators.

6 FunctionModels

The basic `Models` of the previous section certainly do not cover everything that can be meant by the term statistical model. Each member of another large subset of statistical models has an input (independent, exogenous) space (variables, attributes) and an output (dependent, endogenous) space: A type, `fm`, is a `FunctionModel` if it has two type parameters, one an input space, `inSpace` and the other an output space `opSpace`, and if there are functions `condModel` and `condPr` defined. Function `condModel`, named for "conditional `Model`", maps an `(fm inSpace opSpace)`, `m`, and an `inSpace` value, `i`, onto a conditional (dependent) `Model` of `opSpace`. Function `condPr`, named for conditional probability, when also given an `opSpace` value, `o`, returns its probability, `pr(o|i,m)`:

```
class FunctionModel fm where
 condModel :: fm inSpace opSpace -> inSpace -> ModelType opSpace
 condPr :: fm inSpace opSpace -> inSpace -> opSpace -> Probability
 condPr m i o = pr (condModel m i) o
  ...
```

The type `FunctionModelType` can be made an instance of `FunctionModel` class:

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

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

A value of `FunctionModelType` is created by the constructor, `FM`, given a message length (i.e. its complexity) and a function that maps from `inSpace` to `Model opSpace`.

7 TimeSeries

The time series is another important kind of statistical model. A type, `tsm`, is in the `TimeSeries` class if it has a type parameter `dataSpace` and if there are functions `predictors` and `prs` defined on `tsm`. The function `predictors` takes a `(tsm dataSpace)` and a list of `dataSpace` values and produces a list of predictions. The predictions are probabilistic so each one is a `Model` of `dataSpace`.

Note that the list of predictions is one longer than the list of `dataSpace` because the former includes a prediction for the next element *after* the end of the latter list. In addition, function `prs` produces a list of probabilities from a data series.

```
class TimeSeries tsm where
  predictors :: (tsm dataSpace)
                       -> [dataSpace] -> [ModelType dataSpace]
  prs :: (tsm dataSpace) -> [dataSpace] -> [Probability]
  msg2s :: (tsm dataSpace) -> [dataSpace] -> [MessageLength]

  prs tsm dataSeries =
    map (\(m,d) -> pr m d)
      (zip (predictors tsm dataSeries) dataSeries)
  ...
```

The default version of `prs`, above, combines (`zip`) the predictions and data values and takes the probabilities (`pr`) of the latter given the former; the last prediction is ignored.

One natural way to define an instance of a statistical model that is in the `TimeSeries` class is to give a message length and a predictor function. The predictor function takes a *context* of past data values and produces a `Model` over the next element.

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

It is efficient to have the context in reverse order, from most recent to least recent, because of the way that the predictor function is scanned along the input data series when `TimeSeriesType` is made an instance of `TimeSeries`:

```
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 []
  ...
```

Cheating is impossible because the predictor function, `f`, is only shown the *previous* elements of the data series and cannot look ahead. Note that the first prediction relies on the empty context, and that there is the extra prediction for the element that would extend the data series.

Many useful `TimeSeries`, such as Markov models of order k, only examine the k most recent values in the context and it is convenient to have these values first to hand for that reason also.

8 A Parade of Models

So far classes of (basic) `Models`, `FunctionModels` and `TimeSeries` have been defined. These do not exhaust all the possibilities of statistical models by any means but they are a good start. There are some properties shared by these three classes, and perhaps others of their kind. These shared properties naturally belong to some super class of all statistical models which can only have one possible name: `SuperModel`!

The most important properties of a `SuperModel`, in the MML framework, relate to prior probability and message length. Any instance of `SuperModel` must at least define either `prior` or `msg1`. The name `msg1` is used to indicate the first-part of a message, i.e. due to the instance of `SuperModel` itself:

```
class SuperModel sMdl where
  prior :: sMdl -> Probability
  msg1  :: sMdl -> MessageLength
  ...
  prior sm = 2 ** (-msg1 sm)
  msg1  sm = - logBase 2 (prior sm)
  ...
```

8.1 On Name Calling

Every program tells a story and it is important that good words are used to name the values, types and classes. It could be argued that `Model` is the natural name for the super class, but there do not seem to be any useful pure `SuperModels`. It seems that all *useful* `SuperModels` have another job – as a (basic) `Model`, `FunctionModel`, `TimeSeries` or member of some other subclass to be defined. Thus `Model` is too valuable a word to waste on the super class.

`FunctionModel` includes regression models as in polynomial regression etc.. Regression is therefore a plausible alternative name for what has been called `FunctionModel` here, but it is currently felt that regression carries too much other baggage with it to be used for that purpose.

8.2 Changing Clothes

There are some natural functions between classes `Model`, `FunctionModel` and `TimeSeries` (Figure 1).

A `Model` can easily be turned into a `TimeSeries` by ignoring the context and producing the same `Model` for every element in the data series:

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

The above conversion is so natural that the idea of forcing `Model` to be a subclass of `TimeSeries` has been toyed with, but currently it is optional for an instance of `Model` to be an instance of `TimeSeries`. e.g. For `ModelType`:

```
instance TimeSeries ModelType where predictors m dataSeries
    = map (\_ -> m) ((error "") : dataSeries)
```

A `TimeSeries` of `dataSpace` can be turned into a `Model` of `[dataSpace]` (note the square brackets for list):

```
timeSeries2model1 mdlLen tsm =
 MMsg (msg1 tsm) (\dataSeries -> foldl (+)
```

```
          (msg2 mdlLen (length dataSeries))
          (msg2s tsm dataSeries))

timeSeries2model tsm = timeSeries2model1 someModelOfInt tsm
```

i.e. The message length of the data series given the new `Model` is the sum of the message lengths of the elements of the data series under the old `TimeSeries` plus a term for stating the length of the data series. Note that the probability of a long sequence, e.g. a complete chromosome (Stern *et al.* 2000) would likely cause underflow, so it is better to work with message lengths, `msg2s` , rather than probabilities. There is no mathematical difficulty in finding a `Model` for the length of a data series, there being many candidate probability distributions for non-negative integers, but the choice of the right one for a particular problem can be surprisingly tricky (e.g. Allison and Yee 1990). On the other hand, the length may even be common knowledge in some cases and thus of zero cost.

A `FunctionModel` of the right kind, one that embodies a predictor function, i.e. a `FunctionModel` of `[dataSpace] dataSpace`, can be turned into a `TimeSeries` of `dataSpace`:

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

A `TimeSeries` of some `dataSpace` can be converted into a `FunctionModel` of series, `[dataSpace]`, and the `dataSpace`, i.e. that makes a prediction for the next element:

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

A `Model` of `opSpace` can be turned into a `FunctionModel` of `inSpace opSpace` by "ignoring" the input space attributes:

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

And a `FunctionModel` of `inSpace opSpace` can be turned into a `Model` of the product space, `(inSpace, opSpace)`:

```
functionModel2model fm =
  MMsg (msg1 fm) (\(i, o) -> condMsg2 fm i o)
```

i.e. The input variables (attributes), `i`, are assumed to be common knowledge, having zero cost – which is the case in supervised problems.

9 Mixtures

A mixture of two or more component `Models` behaves as a weighted average of the components. For example a fifty-fifty mixture of a fair coin and a biased coin is a slightly biased coin. A large family of methods for unsupervised classification (clustering, numerical taxonomy) infer multivariate mixture Models. To form a mixture of `Models` requires a list of K `Models`, by a small stretch of Haskell notation, a `[Model]` of length K, and a list of K weights or probabilities, i.e. `[Probability]`, that sum to one. More generally, such

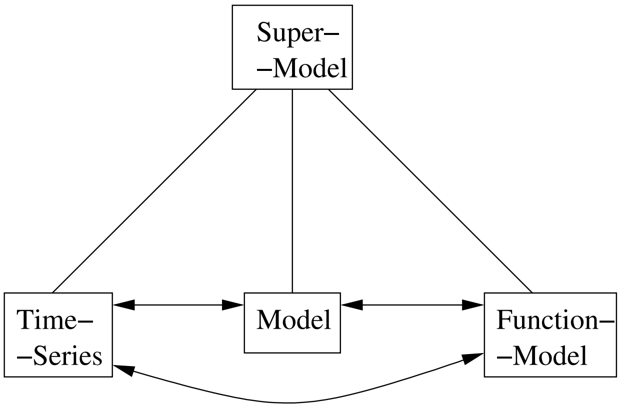

Figure 1: Classes.

a list of probabilities is really a `Model` of `0..K-1`. When focusing on clustering, one might be tempted to make the function `mixture` operation a requirement of class `Model`, e.g. `mixture :: (mdl Int)` $\to$ `[mdl dataSpace]` $\to$ `mdl dataSpace`, or similar. But there are also mixtures of `FunctionModels` and `TimeSeries`, collectively `SuperModels`, so the definition of the latter is revised:

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

Note that the text before the "=>" specifies that `mx` must be a `Mixture` and that `(mx sMdl)` must be a `SuperModel`. In that case `mixture` takes a `(mx sMdl)` and produces a `sMdl`.

A class, `Mixture`, and a data type, `MixtureType`, are useful:

```
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
  mixer      (Mix m _ ) = m
  components (Mix _ es) = es

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

Note that the message length (`msg1`) of a `Mixture` is the sum of that of the `mixer Model` and of the component `SuperModels`.

The mixture of a collection of `Models` of `dataSpace` is itself a `Model` of `dataSpace` by adding the following to the appropriate instance declaration:

```
instance SuperModel (ModelType dataSpace) where
  ...
  mixture mx =
    let m2 datum = calculate weighted avg'
             ...
    in MMsg (msg1 mx) m2
```

Similarly a mixture of `TimeSeries` of `dataSpace` is itself a `TimeSeries` of `dataSpace`:

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

Note that the mixture of `TimeSeries` operates through the mixture of predicted `Models` for each position in the data series.

And a mixture of `FunctionModels` of `inSpace opSpace` is a `FunctionModel` of `inSpace 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
```

It is defined through the mixture of conditional `Models` for a given input value.

9.1 Mixture Modelling

The preceding sections have discussed various kinds of statistical models with rarely a mention of an instance of any particular model (this is not unknown for a "theory"). Here an example of a particular inference problem and an inference algorithm for it are presented. The problem is *mixture modelling* (unsupervised classification, clustering, numerical taxonomy), that is: Given univariate or multivariate data, infer a mixture `Model` that best describes the data (Wallace and Boulton 1968, Figueiredo and Jain 2002). In general neither the number of component `Models` in the answer nor the parameter values of the components are known in advance. This problem provided possibly the first application of information theoretic inference to a practical problem: Wallace and Boulton (1968) developed the MML theories of the multi-state distribution, the normal distribution and multivariate combinations of them, and embodied the results in a Fortran program for mixture modelling, "Snob". Snobbery, its family of descendants in various languages, is alive and well. (In unsupervised classification the components of a mixture are often called classes but that word is already taken here by the programming language use.)

In the present framework, an estimator is required for a mixture `Model` given a list of estimators and given a data set.

```
estMixture ests dataSet =
  let ...
  in ...
```

For simplicity only, the number of components is fixed here as the length of the list of estimators, `ests`; Snob's full algorithm can also split and merge classes and uses extra heuristics in its search for the optimal mixture. Although the number of components is fixed, the total message lengths for one, two, three, ... etc. components, up to some moderate limit, can be compared to perform a more general search. Note that the components of a mixture can be different kinds of `Model`, e.g. geometric and Poisson distributions, provided that they all have the same data space.

The search problem involves a chicken (mixture) and an egg (memberships). Given a mixture, the probability that a given datum is a *member of* the `c`-th component, that is `Model m`, of the mixture is proportional to the probability of component `c` times the probability of the datum given that `Model`:

```
(pr mixer c) * (pr m datum)
```

Normalizing these values gives the probabilities that the datum belongs to component 0, component 1, etc.. The normalized values are also called the *(fractional) memberships* of the datum.

Given the fractional memberships of all the data, a new mixture can be estimated: The weight of a particular component in the mixture becomes the sum of the fractional memberships of the data for that component, i.e. the number of data that it owns. The fractional memberships also form weights for estimation of the parameters of a component `Model`, being fed, with the data set, into the estimator for that component. (Total assignment of a datum to a single component leads to biased parameter estimates in general; fractional assignment avoids this failing.)

The algorithm consists of an *expectation maximisation* loop, fitting new memberships to the current mixture, then fitting a new mixture to the current memberships, repeatedly:

```
cycle    mx = fitMixture (memberships mx)
cycles 0 mx = mx
cycles n mx = cycles (n-1) (cycle mx)
```

The process is started by allocating *random* fractional memberships of data items across the components. Figure 2 shows the final estimator algorithm to find a mixture `Model`.

The function `estMixture` (Figure 2), can be used to estimate a mixture `Model` of a given number of component `Models`, for *any* type of data, given (i) the estimators for those components and (ii) a corresponding training data set. e.g. To infer simple properties of calls in a game of two-up:

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

      randomMemberships
         = ... not hard

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

      fitMixture mems
         = Mix (freqs2model (map (foldl (+) 0) mems))
               (fit ests mems)

      cycles  ... as discussed

  in mixture (cycles some_value
             (fitMixture randomMemberships))
```

Figure 2: Estimate a Mixture Model

```
twoUpData = [(H,H), (T,H), ... etc. ...]

est2coins = estBivariateWeighted( estMultiStateWeighted,
                                  estMultiStateWeighted )

twoCoins = estMixture [est2coins, est2coins] twoUpData
```

The result is a two-component mixture `Model` of pairs of `Throws`. A type error will result if an attempt is made to use `twoCoins` with any other sort of data, although nothing in the definition of `estMixture` limits that function to that kind of data.

If for example, `(H,H)` and `(T,T)` are over-represented and `(T,H)` and `(H,T)` are under-represented in `twoUpData` then the best two-component `Model` fits the data better than a one component `Model` under the MML criterion, and the conclusion is that something fishy is going on in the game.

It can be said that `estMixture` itself only really estimates the weights of the components and passes off the responsibility of estimating the parameters of the components to other estimator functions. This is true, but the estimator for the multistate `Model` (distribution) has been seen to be quite simple, and that for the normal `Model` (distribution) is well known and not difficult. Passing off responsibility like this is also a good thing! If a more complex multi-variate estimator embodying, say, factor analysis (Wallace and Freeman 1992) is developed then it can be used with `estMixture` without change to the latter.

A nice feature of `estMixture` is that the only reference to a component, `m`, of the mixture being a `Model` is the ability to apply `(pr m datum)`. Because a `FunctionModel` of `inSpace opSpace` is turned into a `Model` of `(inSpace, opSpace)` by `functionModel2model`, `estMixture` can also, in effect, infer mixtures of `FunctionModels` for no extra effort. A `TimeSeries` of `dataSpace` can also be turned into a `Model` of `([dataSpace], dataSpace)` and hence `estMixture` can in principle also infer mixtures of `TimeSeries`. In the previous example `twoCoins` is a memory-less `TimeSeries` of `Throw` (because `ModelType` is also an instance of `TimeSeries`), but `TimeSeries` that examine the context are of course possible.

10 Classification (Decision) Trees

Classification trees, also known as decision trees, are used in *supervised classification*. The problem is to learn a classification function, from an input space, `inSpace`, to an output space, `opSpace`, given examples, i.e. learn a `FunctionModel` of `inSpace opSpace`. The best known classification tree program is C4.5 (Quinlan 1992) and its relatives. In that program `opSpace` must be an enumerated, bounded, unordered type.

It is not hard to define a classification tree data type, `CTreeType`, in the present framework. A tree can be a leaf, `(CTleaf (ModelType opSpace))`. Note that the leaf `Model` can be of any `opSpace` at all – of `Float`, or of `(BloodPressure, BloodSugar)`, etc.. A tree can also be a fork with subtrees.

A traditional fork, `CTforkTrad`, contains a selector function on `inSpace` that will send a datum down one of the subtrees, together with the message length of the function for MML purposes, and a list of subtrees. For example, the function might test the *gender* attribute of a person and then use either the 0th or 1st subtree to predict the `opSpace` attribute(s). The selector function can be thought of as an extreme kind of `FunctionModel` of `inSpace Int` that allocates all of its probability to the index of a single subtree. This suggests a more general `CTfork`.

```
data CTreeType inSpace opSpace =
  CTleaf (ModelType opSpace) |
  CTforkTrad MessageLength (inSpace -> Int)
                [CTreeType inSpace opSpace] |
  CTfork (FunctionModelType inSpace Int)
                [CTreeType inSpace opSpace]
```

The `CTfork` contains a selector `FunctionModel` and a List of subtrees, sufficient information for what might be called a `FunctionMixture` of classification trees, that is the subtrees all predict the `opSpace` attribute(s) and the mixture of their results is controlled by the `FunctionModel`.

```
instance SuperModel (CTreeType inSpace opSpace) where
  msg1 (CTleaf leafModel) = (msg1 leafModel) + ...
  ... etc

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

```
condModel (CTfork fnMixer dts)     i = ... etc.
```

Note that the *structure* of a tree contributes towards its message length, its complexity. The details (Wallace and Patrick 1993) are interesting but they are not relevant here.

Classification trees can now be created (a search algorithm is not presented here) and compared under, say, the MML criterion. e.g. Suppose that Fred is suspected of using biased coins:

```
cTree = CTforkTrad 0
          (\ip -> if (name ip)=="Fred" then 0 else 1)
          [CTleaf biasedCoin, anotherSubTree]
```

Our `functionModel2model` can also be used to make a `FunctionModel` tree, that is a regression tree, e.g.

```
-- linear, FunctionModel of Float Float
-- i.e. y ~ a*x+b+(normal 0 epsilon)
linear a b epsilon = ...

leaf0 = CTleaf (functionModel2model (linear 1 2 1))

regTree = CTfork (...) [leaf0, ... ]
```

The general result is a `FunctionModel` of `ipSpace1 (ipSpace2, opSpace)`, which can be reconfigured into a `FunctionModel` of `inSpace opSpace` by the use of a small conversion function where `inSpace`, `inSpace1` and `inSpace2` are related at the programmer's discretion; e.g. `inSpace`, `inSpace1` and `inSpace2` can be identical, or `inSpace1` and `inSpace2` can be separate with `inSpace` being their product, and so on.

10.1 A Tree Estimator

An estimator, `estCTree`, for a classification tree needs a leaf `Model` estimator, `estLeafMdl`, methods to `split` a data set, and a training set consisting of inputs, `ipSet`, with corresponding outputs, `opSet`. A reasonable way of splitting data is to pass a function, `splits`, which returns a list of partitioning functions. Each partitioning function divides `ipSet` into at least two non-empty `parts`. The simplest possible tree consists of a single `leaf` node. Any other tree contains at least one fork node. The simplest, and fastest, 0-lookahead search investigates all `alternative` trees that contain exactly one fork node. The MML criterion is used to select between the competing trees. The best choice out of the single-leaf tree and the one-fork trees is selected. The search is complete if the single-leaf tree is best. Otherwise the search is continued recursively for each subtree for the appropriate *part* of the training data. Figure 3 shows the algorithm.

Conventionally, a K-way split is tried on a discrete input attribute of range K. Binary splits, at different *cut-points*, are tried on an input attribute of continuous values. The splits for multivariate input data include the splits for each attribute. A wider choice of splits can be searched, more slowly.

Note that simple conversion functions allow `estCTree` to estimate other kinds of tree. For example `estFunctionModel2estModel` converts an estimator for a `FunctionModel` into an estimator for a `Model` that can be used with `estCTree` to estimate a `FunctionModel`-tree (a regression tree), e.g.

```
estCTree  estLeafMdl splits  ipSet opSet =
  let
   search ipSet opSet =
    let
      leaf    = CTleaf leafMdl
      leafMdl = estLeafMdl opSet
      leafMsg = foldl (+) (msg1 leaf)
                  (map (msg2 leafMdl) opSet)

      partition ... = ...

      alternatives ... = ...
        ... choose best alternative...

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

       (t, _, _) -> t     -- single leaf wins, done

  in search ipSet opSet
```

Figure 3: Estimate a Tree

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

ft = estCTree (estFunctionModel2estModel someFnMdlEstimator)
       splits ipSet opSet
```

11 Files

The Haskell code of the previous sections is not intended for processing very large data sets, as it makes the assumption that the data fit into (virtual) memory, although that is not to say that the restriction could not be lifted.

The types and classes defined for statistical models are completely general about the data spaces they apply to, but a particular statistical model (e.g. `fairCoin`) is over a particular data space (e.g. `Throw`). If the data are of a new type and come from a file there are two possible approaches: The data could be translated into standard types – discrete values into `Int`s and continuous values into `Float`s, say. All analysis could then be done in terms of `Int`s, `Float`s and structures of them, and the results mapped back into terms of the original data. Alternatively, a small module could be compiled to define the types and to read and write the values specific to the data file and to evaluate their analysis; nothing else needs recompilation.

12 Conclusions

A step has been taken in designing a collection of classes, types, values and operators to define statistical models – "a model of modelling" perhaps? Already some surprising generalisations have popped out, for example in mixture modelling and in classification functions.

A reasonable spring-collection of kinds of statistical models has been shown – basic `Models` including probability distributions, `FunctionModels` including regressions, and `TimeSeries`, together with instances of these and useful estimators and operators on them and between them.

Haskell has a fair claim to being the reference functional programming language of today. It makes a good meta-language for the study of statistical models. One result is a theory that is rigorously type checked, and that runs. There is no reason why a Haskell program need be more than a constant factor slower than a corresponding C or Java program so the theory is usable, at least on moderate data sets. Haskell's type system is powerful and flexible. However type systems are still an active area of research and it is possible that some further type features would be useful in this kind of exercise. As a small example, a (system) `class Function` requiring the application operator (`f x` and `f $ x`) and having type $\rightarrow$ as an instance would be useful in allowing subclasses of `Function` to be created.

13 Acknowledgements

Many thanks to members of the Central Inductive Agency in Computer Science and Software Engineering at Monash, particularly Josh Comley, Leigh Fitzgibbon and Chris Wallace, who always provide much inspiration.

14 References

L.Allison (1993) Normalisation of affine gap costs used in optimal sequence alignment. J. Theor. Biol. 161(2) pp263-269.

L.Allison, D.Powell and T.I.Dix (1999) Compression and approximate matching. The Computer Journal 42(1) pp1-10.

L.Allison and C.N.Yee (1990). Minimum message length encoding and the comparison of macromolecules. Bulletin of Mathematical Biology 52(3) pp431-453.

CRAN: The comprehensive R archive network. http://lib.stat.cmu.edu/R/CRAN/ (current 2002)

G.E.Farr and C.S.Wallace (2002). The complexity of strict minimum message length inference. Computer Journal 45(3) pp285-292.

M.A.T.Figueiredo and A.K.Jain (2002). Unsupervised learning of finite mixture models. IEEE Trans. on Patt. Analysis and Machine Intell. 24 pp381-396.

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

N.K.Humphrey (1973). The illusion of beauty. Perception 2 pp429-439.

R.Milne and C.Strachey (1976). A Theory of Programming Language Semantics. (2 vols) Chapman Hall.

R.Milner (1978). A theory of type polymorphism in programming. J. Computer and Sys. Sci. 17 pp348-375.

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

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

C.E.Shannon (1948). A mathematical theory of communication, Bell Syst. Technical Jrnl. 27 pp379-423 and pp623-656.

L.Stern, L.Allison, R.L.Coppel and T.I.Dix (2000). Discovering patterns in Plasmodium falciparum genomic DNA. Molecular and Biochemical Parasitology 118(2) pp175-186.

C.Strachey (1967). Fundamental concepts of programming languages. Int. Summer School in Comp. Prog., Copenhagen, also in Higher-Order and Symbolic Computation 13(1-2) pp11-49 2000.

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

C.S.Wallace and D.M.Boulton (1968). An information measure for classification. Computer Journal 11(2) pp185-194.

C.S.Wallace and P.R.Freeman (1987). Estimation and inference by compact coding. Journal of the Royal Statistical Society series B. 49(3) pp240-265.

C.S.Wallace and P.R.Freeman (1992). Single-factor analysis by minimum message length estimation. Journal of the Royal Statistical Society series B. 54(1) pp195-209.

C.S.Wallace and M.P.Georgeff (1983). A general objective for inductive inference. TR 32, Department of Computer Science, Monash University. (An abridged version appeared as M.P.Georgeff and C.S.Wallace. A General Selection Criterion for Inductive Inference. European Conference on Artificial Intelligence (ECAI, ECAI84), Pisa pp473-482 September 1984.)

C.S.Wallace and J.D.Patrick (1993) Coding decision trees. Machine Learning 11 pp7-22.

I.H.Witten and E.Frank (2000). Nuts and bolts: Machine learning algorithms in Java. In Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations, pp265-320, Morgan Kaufmann.

TR 2002/125. First draft 14 October 2002. This version 4 November 2002. [http://www.csse.monash.edu.au/~lloyd/] L.A., CSSE, Monash University, Australia 3168.
11/2002: A related paper has been accepted for the Australian Computer Science Conference, ACSC2003, Adelaide, 4-7 February 2003, to be published through the Australian Computer Society (ACS) as the Australasian Computer Science Conference 2003 - Australian Computer Science Communications, Volume 25, Number 1, see http://www.csa.edu.au/acsw/acsw03/acsc/