Types and Classes of Machine Learning and Data Mining

Lloyd Allison

School of Computer Science and Software Engineering
Monash University,
Clayton, Victoria, Australia 3168.
`lloyd at bruce cs monash edu au`
`http://www.csse.monash.edu.au/~lloyd/`

Abstract

The notion of a statistical model, as inferred and used in statistics, machine learning and data mining, is examined from a semantic point of view. Data types and type-classes for models are developed that allow models to be manipulated in a type-safe yet flexible way. The programming language Haskell-98, with its system of polymorphic types and type-classes, is used as the meta-language for this exercise so one of the by-products is a running program.

Keywords: Classes, data mining, data types, functional programming, inductive inference, machine learning.

"... 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

Artificial intelligence, machine learning, data mining, and of course statistics, are to a greater or lesser extent concerned with probability distributions, statistical models, model classes, hypotheses and theories. Common activities are fitting model parameters to data, inferring models from data, selecting a model class, and comparing models.

In this paper, types and classes (in the programming language sense) are developed to describe, for want of a name, statistical models where *statistical model* is taken to include all of: Probability distribution, model, model class, hypothesis and theory. The primary aim is to make precise the *behaviour* of statistical models by defining types, classes and the operations on instances of them, thus allowing models to be combined in flexible ways while allowing these combinations to be rigorously checked for validity. This can be viewed as a first small step towards a semantics of statistical models, as denotational semantics (Milne & Strachey 1976) provided for programming languages.

As the formal study of syntax led to good notation for syntax (BNF), tools to process syntax definitions (e.g. parser generators), and better syntax for programming languages, it is hoped that the present study will have beneficial effects on the definition and use of statistical models with computers.

General inspiration is taken from the field of functional programming (FP) which achieves conciseness and generality from treating functions as first-class values and all that follows. It is hoped that treating statistical models as first-class values will bring similar benefits to machine learning and data mining. This has already been useful in special cases (Allison *et al.* 1999) and we want to see how far the idea can be taken. Another aim of the exercise is to provide a rapid prototype for parts of a data mining *platform*, CDMS, being developed locally.

Much research in machine learning and data mining consists of developing a new algorithm to infer a specific kind of statistical model for a certain range of problem, e.g. mixture models for unsupervised classification (clustering) of multivariate data, or classification trees (also known as decision trees) 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 being just one implementation of such functions. The former kind of activity typically leads to stand-alone programs that must use shell-scripts and intermediate files if they are to cooperate on data exchange, and that can hardly be used as building blocks in any sophisticated way. There are some more general data-mining platforms, such as R (see (CRAN 2002)), S-Plus (Venables & Ripley 1999) and Weka (Witten & Frank 2000), which allow data analysis techniques to cooperate and which support the addition of new techniques, but any typing is typically ad-hoc and dynamic, compromising speed, type-safety or generality, or all of these. Here in contrast, polymorphic types and operations (Strachey 1967) are developed for statistical models; many operations on models are naturally polymorphic.

2 FP: Haskell, Types and Classes

Haskell (Hudak *et al* 1992), specifically Haskell-98 (Peyton Jones *et al.* 1999), is used here as the programming language, i.e. the *meta-language*, to describe the 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 type variables as parameters. For example, `[t]`, that is List of `t`, represents the List type 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 "has the type", and "→" denotes a function type. "\" stands for lambda, as in the lambda-calculus (Church 1941), and can, with →, be used to define anonymous functions such as the successor function, `(\n->n+1)`.

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.

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 simplifies many definitions.

3 (Basic) Models

The terms model and statistical model appear in many varied, flexible, creative and even 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 the model's data space, i.e.

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

The Haskell code above 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.

In order to create some instances of `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 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 in Section 3.2. `ModelType` is made an instance of class `Model` by defining the necessary functions, 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 `Model` (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 in the class `Model`".

3.1 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 generalize 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 & Boulton 1968, Wallace & Freeman 1987). Just as the use of Haskell is well motivated but in principle arbitrary, the adoption of the particular framework, MML, is not central to the theme of this paper, although it does fit in rather well, and the work has been done in that framework so it is briefly introduced here. Oliver and Baxter (1994) discuss at length MML's relationship to alternative frameworks.

The MML criterion is to use the length of a two-part message to select between competing hypotheses (distributions, models, theories). It relies on Bayes theorem (Bayes 1763), 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), which gives the message length, *msg*, in an optimal code for an event E of probability pr(E)

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

Hence

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

In many problems pr(D) is unknown, but it is usually possible to give a reasonable estimate for the *prior*, p(H), and 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 & 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 & 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 the two-part message, the part for a datum given the `Model`. The first part of the message, `msg1`, is defined later in Section 6. The total message length, `msg`, for Model and data, is the sum of the parts.

Message lengths can be combined, without conversion to probabilities, by using `logPlus` which is equivalent to

```
logPlus msg1 msg2 =
 let (bigger, smaller)
       = if msg1 >= msg2
         then (msg1, msg2)
         else (msg2, msg1)
     diff = bigger - smaller
     eps  = 2 ** (-diff)
 in if diff > numSignificantBits
    then smaller
    else smaller - (logBase 2 (1+eps))
 -- NB. logPlus m m == m-1
```

3.2 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`:

```
data ModelType dataSpace =
  MPr  ...as before...   |
  MMsg MessageLength
    (dataSpace -> MessageLength)

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 we can ignore the distinction because the measurement-accuracy of the data passes through the MML calculations under reasonable conditions.

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

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

(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
```

Note that `foldl (+) 0` calculates the sum of a list. 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 estimates shrinks) as the given frequencies (i) increase, or (ii) become more biased, or 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

instance Enum Throw where ...

instance Bounded Throw where ...

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 type of the data space. (`Enum` and `Bounded` are standard classes in Haskell.)

A `Model` of a discrete, enumerated, `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 the data space merely to inform the type inference algorithm. On a small point, thanks to laziness the example value need not even exist provided that its type can be inferred as though it does exist.

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 useful later (Section 7.1) in mixture modelling.

Given two `Models`, `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
```

Given (weighted) estimator functions for a `Model` of `dataSpace1` and 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 dataSpace, and their *estimators*.

4 FunctionModels

The basic `Models` of Section 3 certainly do not cover everything that can be meant by "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 there are functions `condModel` and `condPr` defined. Function `condModel`, named for conditional `Model`, maps an `(fm inSpace opSpace)` and an `inSpace` value, `ip`, onto a conditional (dependent) `Model` of `opSpace`. Function `condPr`, named for conditional probability, when also given an `opSpace` value, `op`, returns its probability, `pr(op|ip,fm)`:

```
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
  ...
```

`FunctionModelType` is a data type that 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` of `opSpace`.

5 TimeSeries

The time series is another important kind of statistical model. A type, `tsm`, is in the `TimeSeries` class if there are functions `predictors` and `prs`. The function `predictors` takes a `(tsm dataSpace)` and a series (list) of `dataSpace` and produces a list of predictions for elements in the series. 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`) 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 []
  ...
```

Note that "`:`" is the list constructor and that `scan` is defined by cases, on the empty list `[]` and the non-empty list `d:ds`. Cheating on predictions is impossible because the predictor function, `f`, is only shown the previous elements of the series and cannot look ahead. Note, the first prediction relies on the empty context, and 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 of the data series and it is convenient to have these first to hand in the context for that reason also.

6 A Parade of Models

So far (basic) `Models`, `FunctionModels` and `TimeSeries` have been defined. These do not exhaust the possibilities of statistical models but they are a good start. There are some properties shared by these three classes, and 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 property of `SuperModel`, in the MML framework, is the message length.

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

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.

6.1 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, in the author's opinion.

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

6.2 Changing Clothes

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

A Model can 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 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 Model can also be turned into a FunctionModel by a similar trick:

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

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

```
timeSeries2model1 mdlLen tsm =
 MMsg (msg1 tsm + msg1 mdlLen)
      (\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 given 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.* 2001) 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

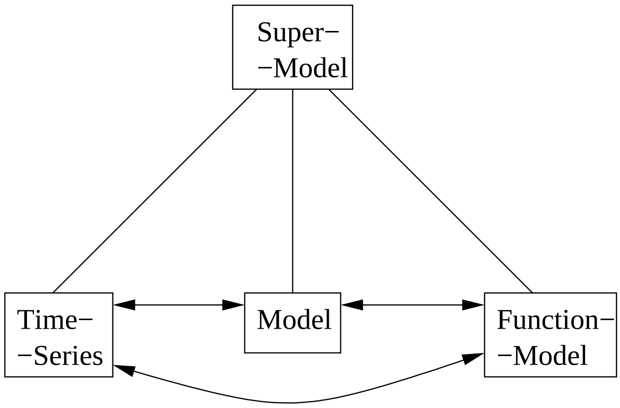

Figure 1: Classes and Functions.

problem can be surprisingly tricky (e.g. (Allison & Yee 1990)). On the other hand, the length may even be common knowledge in some cases and thus free.

We have seen that a Model of dataSpace can be turned into a TimeSeries of dataSpace and thence into a Model of [dataSpace]. One might ask, "*Is* a Model of dataSpace a Model of [dataSpace]?" and the answer is almost but formally no. The transformation from the former to the latter is straightforward but the distinction is necessary to keep the types legal, as can be seen by considering, say, an image to be a series of scan-lines, and a scan-line to be a series of pixels.

A TimeSeries of dataSpace can be turned into a FunctionModel of [dataSpace] dataSpace by returning the prediction given the whole data series as context:

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

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)
```

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.

As well as these, and other, functions on the various kinds of statistical model, there are similar functions on their estimators. Taken together they add to the generality of the system.

7 Mixtures

A mixture of two or more component Models behaves as a weighted average of the components. For example a 50:50 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, i.e. a [Model...] of length K, and a list of K weights or probabilities, i.e. [Probability], that sum to one. More generally, such a [Probability] is really a Model of 0..K-1. If focusing on clustering, one might be tempted to make the function mixture a requirement of class Model, mixture :: (mdl Int) → [mdl dataSpace] → 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 type, MixtureType, are useful. A Mixture must be able to produce a list of components and a mixer Model controlling them:

```
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 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.

7.1 Mixture Modelling

The preceding sections have discussed various kinds of statistical models with hardly a mention of an instance of any particular model. 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) (Wallace and Boulton 1968, Day 1969, Wolfe 1970), that is: Given univariate or multivariate data, infer a mixture Model that best describes the data. 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 an important, 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, Snob. Snobbery, its family of descendants in other 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), and there continues to be interest in MML and mixture modelling (Figueiredo & Jain 2002). MML allows simple and complex mixtures, of few and many components, to be compared fairly giving a built-in stopping criterion as an alternative to a external significance test (Wolfe 1970), say.

This paper is not about mixture modelling as such; the problem is simply used as a representative (and important) problem from statistics, machine learning and data mining; most data mining platforms have some sort of clustering ability. In the present framework, an estimator is required for a mixture Model given a list of estimators and a data set.

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

For simplicity only, the number of components is fixed as the length of the List of estimators, ests; Snob's full optimization 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 here, the total message lengths for one, two, three, ... etc. components, up to some moderate limit, could be compared to perform a more general search.

The search problem involves a chicken (mixture) and an egg (memberships). Given a mixture, the probability that a given datum belongs to the c-th component, 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)
```

```
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

Normalising these values gives the probabilities that the datum belongs to component 0, component 1, etc.. These 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 such a failing.)

The final algorithm is a typical expectation maximization loop (Baum & Eagon 1967, Dempster, Laird & Rubin 1977), fitting memberships to the mixture, then fitting a mixture to the memberships, repeatedly:

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

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

`estMixture`, 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:

```
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 sufficiently 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 is also a good thing! If a more complex multi-variate estimator performing, say, factor analysis (Wallace & 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)` to it. Recall that a `FunctionModel` of `inSpace opSpace` can be turned into a `Model` of `(inSpace, opSpace)` by `functionModel2model` (Section 6.2), and that an estimator of a `FunctionModel` can be treated analogously. This means that `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]`, 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.

8 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, `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`, of `(BloodPressure, BloodSugar)`, etc.; this is more general than C4.5. 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 of a person to decide whether to use the 0th or 1st subtree to predict the `opSpace` attribute(s). The selector function can be thought of as an extreme, all or nothing, kind of `FunctionModel` of `inSpace Int`. This also suggests a more general kind of fork, `CTfork`.

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

It might be useful to define a class `Function` and make type $\rightarrow$ an instance of it; a subclass of `Functions` having message lengths could then defined, allowing a more elegant version of `CTforkTrad`. Unfortunately `Function` would have to be a slightly special system class before the implicit application operator, "`f x`", could be overloaded.

A `CTfork` node 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.
```

Classification trees can now be created and compared under, say, the MML criterion (a search algorithm is given elsewhere (Allison 2002)). e.g. Fred is suspected of using biased coins:

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

Our `functionModel2model` conversion function (Section 6.2) 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 rejigged to `FunctionModel` of `inSpace opSpace` by the use of a trivial wrapper 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.

9 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. As a semantic study, or as a rapid-prototype of a data-mining platform, this is not important.

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 `Ints` and continuous values into `Floats`, say. All analysis could then be done in terms of `Ints`, `Floats` and structures of them, and the results mapped back into the 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.

10 Conclusions

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

A reasonable spring-collection of kinds of statistical models has been devised – basic `Models` and distributions, `FunctionModels` including regressions and classification functions, and `TimeSeries` such as Markov models, together with instances of these and useful operators on and between them and their estimators. These operators add to the generality of the collection.

Haskell has a fair claim to being the reference functional programming language of today. It does make a good meta-language for the study of statistical models. A 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 small constant factor slower than a corresponding C or Java program so the theory should be usable, at least on data sets of moderate size. Haskell's type system is powerful and flexible and has proved invaluable to the exercise. However type systems are still an active area of research and it is possible that some further type features might be useful in this kind of study.

11 Acknowledgements

Many thanks go 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.

References

Allison, L. (2002), Model Classes, TR 2002/125, School of Computer Science and Software Engineering, Monash University, Clayton, Victoria, Australia.

Allison, L., Powell, D. & Dix, T. I. (1999), Compression and approximate matching, Computer Journal **41**(1), pp. 1–10.

Allison, L. & Yee, C. N. (1990), Minimum message length encoding and the comparison of macromolecules, Bulletin of Mathematical Biology, **52**(3), pp. 431–453.

Baum, L. E., & Eagon, J. A. (1967), An inequality with applications to statistical estimation for probabilistic functions of Markov processes and to a model of ecology, Bulletin Amer. Math. Soc. **73**, pp. 360–363.

Bayes, T. (1763), An essay towards solving a problem in the doctrine of chances, Philosophical Transactions of the Royal Society of London **53**, pp. 370-418, reprinted in Biometrika **45**(3/4), pp. 293-315, 1958.

Church, A. (1941), The Calculi of Lambda conversion, Princeton University Press, Annals of Math. Studies **6**.

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

Day, N. E. (1969), Estimating the components of a mixture of normal distributions, Biometrika **56**(3), pp. 463–474.

Dempster, A. P., Laird, N. M. & Rubin, D. B. (1977), Maximum likelihood from incomplete data via the EM algorithm, Journal of the Royal Statistical Society series B **39**(1), pp. 1–38.

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

Figueiredo, M. A. T. & Jain, A. K. (2002), Unsupervised learning of finite mixture models, IEEE Transactions on Pattern Analysis and Machine Intelligence **24**, pp. 381–396.

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

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

Milne, R. & Strachey, S. (1976), *A Theory of Programming Language Semantics* (2 vols), Chapman Hall.

Milner, R. (1978), A theory of type polymorphism in programming. Journal of Computer and System Science, pp. 348–375.

Oliver, J. & Baxter, R. A. (1994), MML and Bayesianism: Similarities and differences, TR 206, Department of Computer Science, Monash University, Clayton, Victoria, Australia.

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 Systems Technical Journal **27**, pp. 379–423 and pp. 623–656.

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

Strachey, C. (1967), Fundamental concepts of programming languages, Int. Summer School in Computer Programming, Copenhagen, also in Higher-Order and Symbolic Computation **13**(1–2) pp. 11–49, 2000.

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

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

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

Wallace, C. S. & Freeman, P. R. (1992), Single-factor analysis by minimum message length estimation, Journal of the Royal Statistical Society series B. **54**(1) pp. 195–209.

Wallace, C. S. & Georgeff, M. P. (1983), A general objective for inductive inference, TR 32, Department of Computer Science, Monash University. (An abridged version appeared as M.P.Georgeff & C.S.Wallace. A general selection criterion for inductive inference, European Conference on Artificial Intelligence (ECAI), Pisa, pp. 473–482, September 1984.)

Witten, I. H. & Frank, E. (2000), 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.

Wolfe, J. H. (1970), Pattern clustering by multivariate mixture analysis, Multivariate Behavioural Research **5**, pp. 329–350.

This paper appeared in: 26th Australasian Computer Science Conference (ACSC2003), Adelaide 4-7 February 2003; Conferences in Research and Practice in Information Technology V**16**; Australian Computer Science Communications V**25**(1) pp207–215; copyright Australian Computer Society.