Bayesian Reasoning and Machine Learning

The BRML Matlab package

CHAPTER 1

Introduction

This document is in draft form and far from an exhaustive discussion of the package.

This document describes the main features of the BRML software package, in particular the object oriented version. There are various routines in the package for both probabilistic potential manipulation and inference, in addition to classical data analysis routines. Hopefully the data analysis routines should be largely self-explanatory and may be considered as somewhat stand-alone. The potential manipulation parts of the package interact more and are the focus of this short tutorial.

The package makes use of the package feature of Matlab (rather than being a toolbox). This means that the routines are not on the path and either need to be imported explicitly using the `import` function or called explicitly. See the Matlab Object Oriented Documentation for a description of packages and OO concepts.

There are a couple of benefits of using the package structure: it makes using code from different sources more straightforward, particularly for functions with conflicting names; overloading operators makes for a natural way to produce generic potential-independent inference routines. To extend the package then requires only the creation of a class for the new potential and basic operations in the class. The drawback is that not all these features are yet implemented in Octave (the free Matlab clone)[1].

Once the package has been downloaded and unzipped, simply type `setup` at the Matlab command prompt to set the paths and import the package into the workspace[2]. Typing `what brml` gives a listing of the package m-files and classes. Similarly, `help brml` lists the m-files and a one-line description of each m-file.

[1]For this reason at some point it would be great to have a Python version of the code.
[2]Note that one still needs to import the package within a function.

CHAPTER 2

Potential Classes

The package comes with a small number of potential classes namely `@potential`, `@array`, `@logconst`, `@const`, `@logarray`, `@GaussianMoment`, `@GaussianCanonical`.

2.1 @potential

The potential class itself contains only two properties: variables and table. All other potentials are subclasses of the potential class. Which methods are defined for each class can be obtained using the `methods` command. For example

```
>> methods array

Methods for class brml.array:

array           emppot          log             mrdivide        reshape         size
brml.logarray   evalpot         logscalar       mtimes          sample          sum
delta           exp             max             permute         setpot
disptable       exppot          maxN            plus            setstate
```

2.2 @array

The array potential class deals with manipulating discrete probability tables. The interaction with other classes can be found by typing `help array` and a description of the constructor is given from `help array.array`.

One can create an instance of the array by typing

```
>> p=array

p =

  brml.array
  Package: brml

  Properties:
    variables: []
        table: []

  Methods, Superclasses
```

To specify the variables we can write

```
p.variables=[1 2];
```

which states that the variables of the potential are 1 and 2 (in that order). To specify the table itself we can write for example

```
p.table=[0.05 0.3; 0.45 0.2];
```

`p` is now a brml.array instance with fully specified variables and table.

Equivalently we can just write

```
p=array([1 2],[0.05 0.3; 0.45 0.2])
```

The number of states of the variable is given by the size of the table. So for example

```
p=array([1 2],rand(2,3))
```

defines a potential on variables 1 and 2 with variable 1 having 2 discrete states and variable 2 having 3 discrete states.

The main manipulations we will require are marginalisation and multiplication. For marginalisation we use the `sumpot` command. For example

```
q=sumpot(p,1)
```

gives a new array potential `q` defined on variable 2. The `sumpot` function calls the overloaded `sum` operation so that `sum(p,1)` is also defined.

Let's define a simple belief network

$$p(x_1, x_2, x_3) = p_1(x_1|x_2)p_2(x_2, x_3)$$

```
p1=array([1 2],condp(rand(2,3)));
p2=array([2 3],condp(rand(3,4),[1 2]));
p=multpots({p1 p2});
```

This defines `p1` as a distribution[1] over variable 1 conditioned on variable 2 (the `condp` command by default normalises over the first dimension, otherwise over all dimensions specified in the second argument). `p2` is a distribution over variables 2 and 3. `p` is now a distribution over variables 1,2,3 with dimensions $2, 3, 4$ respectively. Naturally, we need the dimensions of the tables to match (here the number of states of variable 2 must be the same in `p1` and `p2`) for the multiplication to work. The `multpots` command multiples any number of potentials given in the argument cell.

The `*` operator is overloaded for `brml.array`, so that we can equivalently write

```
p=p1*p2;
```

for the above[2].

In addition the binary operators `+` and `/` are defined, so that one could write, for example

```
p=p1+p2; p=p1/p2;
```

[1]`condp(rand(2,2),[1 2])` is a handy way to specify a random table that sums to 1.
[2]The `prod` operation is not currently overloaded since the builtin `prod` command is frequently called in potential methods.

2.2.1 A note on cells versus arrays

We can also write `q=[p1 p2]` which defines a 1×2 `brml.array`. If `p1` and `p2` are not in the same class, Matlab seeks to convert `[p1 p2]` to 1×2 potential of the dominant class. Note that for an overloaded operator $f(\cdot)$, then for an $m \times n$ potential p, `f(p)` returns an $m \times n$ potential in which the function f is applied to each potential element.

Alternatively we can write a cell `r={p1 p2}`; in this case Matlab does not perform conversion and the cell entries can remain polymorphic. The brml package generally assumes that cells are used since this can avoid defining converters if no subsequent interaction between the elements of the cell is required. For example if we simply want to draw the belief network of set of potentials, we don't need to convert them. In this sense working with cells of potentials is more flexible than working with arrays of potentials.

Note that

```
log([p1 p2])
```

is defined since the `log` operator is overloaded for the array class. This returns a 1×2 array potential. However

```
log({p1 p2})
```

is not defined since Matlab does not distribute operators over cell elements. However, the potential method (*i.e.* not subclass specific) `logpot` is defined and returns a cell. In this case

```
logpot({p1 p2})
```

and

```
logpot([p1 p2])
```

both return cells.

2.3 @logarray

A `@logarray` potential is the same as an `@array` except that the entries are stored and manipulated in log form. This can be defined in the same way as `@array`. We can also convert an `@array` object to a `@logarray` object. For example

```
p=array([1 2],rand(2,3)); logp=logarray(p)
```

This defines `logp` to be an `@logarray` object. Note that this is not the same as

```
logpp=log(p)
```

which would define an `@array` object simply with the table entries the log of the entries of `p`.

The operations of the `@array` class are also defined for the `@logarray` class. For example

```
p=array(1,exp(1)); logp=logarray(p)
```

This defines a potential on variable 1 (which contains only a single state), with

```
logp =

  brml.logarray
  Package: brml

  Properties:
    variables: 1
        table: 1
```

Then

```
logpp=logp*logp

logpp =

  brml.logarray
  Package: brml

  Properties:
    variables: 1
        table: 2
```

This means that the elements of the `logp` table are not multiplied (which would give a table of 1) – rather they are added (which gives a table of 2). The `@logarray` operations are equivalent to first exponentiating the log table entries, then carrying out the operation as for the `@array` class, and then returning the log of the result. The usefulness of the `@logarray` class is that it can help avoid numerical over/underflow for example when running inference routines such as the sum-product algorithm on a large factor graph. The benefit of using classes is that we can simply reuse any existing algorithm defined for the `@array` class and covert to this to the `@logarray` class, and the result will be returned in log form, without requiring modification of the algorithm.

2.3.1 No change to high-level algorithms

For example `demoSumProdLogMess.m` demonstrates how we can call the standard `sumprodFG.m` code and get the results returned in log form, without requiring any modification of `sumprodFG.m`.

Similarly, we can call the junction tree code without modification and the results will be returned in log form.

2.3.2 Superiority

The `@logarray` is superior to the `@array` class since this will generally help improve numerical stability. This means that for our example above `q=[p logp]` returns a 1×2 `@logarray` in which `p` is converted to `@logarray`.

2.4 @const

```
>> c=const(2)

c =

  brml.const
  Package: brml

  Properties:
    variables: []
        table: 2
```

The `@const` defines a table with empty variables. It is inferior to the `@array` class so that for our example above `p*c` returns an `@array` on variable 1 with table value $2\exp(1)$. In general for a potential $\phi(\mathcal{X})$ on a collection of variables $\mathcal{X}$, and constant potential c, then $c\phi(\mathcal{X})$ returns a potential on variables $\mathcal{X}$ with value $c\phi(\mathcal{X})$.

2.5 @logconst

This behaves similarly to `@const` except that the operations are carried out in log space, in a similar manner to `@logarray`.

2.6 @GaussianMoment

In general the `@GaussianMoment` considers potentials of the form

$$\phi(\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N) = e^c \frac{1}{\sqrt{\det(2\pi\mathbf{\Sigma})}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathsf{T}} \mathbf{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu})\right)$$

where

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}^1 \\ \mathbf{x}^2 \\ \vdots \\ \mathbf{x}^N \end{pmatrix}$$

It's probably easiest to explain this with an example.

```
a=1; b=2; dim(a)=3; dim(b)=2; totaldim=sum(dim);
p=GaussianMoment;
p.variables=[a b];
p.table.mean=randn(totaldim,1);
tmp=randn(totaldim,totaldim); p.table.covariance=tmp*tmp';
p.table.dim=dim;
p.table.logprefactor=0;
```

This defines a Gaussian potential `p` on two variables a and b. The variable a has 3 dimensions and variable b has 2 dimensions. The above then defined a joint Gaussian potential with $5 = 3 + 2$ dimensional mean and covariance matrix:

```
p =

  brml.GaussianMoment
  Package: brml

  Properties:
    variables: [1 2]
        table: [1x1 struct]
```

Summing this potential over variable b gives a Gaussian potential with 3 dimensional mean and covariance:

```
>> pa=sumpot(p,b)

pa =

  brml.GaussianMoment
  Package: brml

  Properties:
    variables: 1
        table: [1x1 struct]
```

where the table is given by

```
>> pa.table

ans =

            mean: [3x1 double]
      covariance: [3x3 double]
             dim: 3
    logprefactor: 0
```

See `demoSumprodGaussMoment.m` which demonstrates how no change is required to the standard `sumprodFG.m` algorithm – since it knows how to sum and multiply `@GaussianMoment` potentials.

This class enables one to perform inference with linear-Gaussian systems at ease.

2.7 @GaussianCanonical

This class defines a Gaussian in canonical form

$$\phi(\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N) = e^d \exp\left(-\frac{1}{2}\mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x} + \mathbf{x}^\mathsf{T}\mathbf{b}\right)$$

where again the stacked vector is given by

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}^1 \\ \mathbf{x}^2 \\ \vdots \\ \mathbf{x}^N \end{pmatrix}$$

and **A** is called the 'precision' and plays the role of the inverse covariance, and **b** is called the 'inverse mean'. Formally, the translation between the moment and canonical representations is given by

$$\boldsymbol{\Sigma} = \mathbf{A}^{-1}, \quad \boldsymbol{\mu} = \mathbf{A}^{-1}\mathbf{b}, \quad c = d - \frac{1}{2}\mathbf{b}^\mathsf{T}\boldsymbol{\mu} - \frac{1}{2}\log\det(2\pi\mathbf{A})$$

The convertors are defined between the two representations so that we can write for example

```
>> pc=GaussianCanonical(p)

pc =

  brml.GaussianCanonical
  Package: brml

  Properties:
    variables: [1 2]
        table: [1x1 struct]

  Methods, Superclasses

>>
>> pc.table

ans =

    invcovariance: [5x5 double]
          invmean: [5x1 double]
     logprefactor: -7.3079
              dim: [3 2]
```

`demoSumprodGaussCanonLDS.m` gives an example of inference in the Linear Dynamical System using the Canonical Gaussian representation, calling the standard `sumprodFG.m` algorithm, comparing the results given by for example the RTS algorithm.

2.8 Extending the set of potential classes

For a potential type that is closed under marginalisation and multiplication it is straightforward to write a new class. See for example the directory `+brml\@GaussianMoment`. Along with the class definition and constructor (and possibly any convertors), the necessary methods are:

`mtimes.m` The '*' multiplication operator and specifies how to multiply two potentials $\phi_1 * \phi_2$ from the class. This should be in the form `newpot = mtimes(pot1,pot2)`

`size.m` Returns the dimensions of the continuous variables in the potential or the number of states for a discrete variable). This should be in the form `nstates = size(pot)`

`sum.m` Marginalises the potential $\phi(\mathcal{X})$ over a subset of variables $\mathcal{S} \subset \mathcal{X}$. This should be in the form `newpot = sum(pot,vars)`