Loadable Modules: Tilted Probabilities

***Ects* source code**

Russell Davidson

Table of Contents

Chapter 1

Tilted Probabilities

1. Introduction

In bootstrapping with resampling, it is sometimes desirable not to resample from a set of objects with equal probabilities for each object, but rather to have different probabilities assigned to the different objects. There are various naïve methods of achieving this, but the most efficient way I know of is sketched in Knuth (1998), section 3.4.1A, where it is ascribed to A. J. Walker.

The problem can be stated as follows. Consider the discrete probability distribution with atoms at $0, 1, \ldots, n-1$, with probability π_i associated with the outcome i. We require of course that the sum of the π_i should be equal to one. If we make drawings from this distribution, and associate the outcome i with the object i, $i = 0, 1, \ldots, n-1$, then we achieve the desired weighted resampling. We wish to implement the discrete distribution as efficiently as possible.

2. The Algorithm

Let π_i be the desired probability of drawing index i, $i = 0, \ldots, n-1$. Define $P_i = \sum_{j=0}^{i} \pi_j$, and let $P_{-1} = 0$. A naive algorithm would draw a random number U from U(0,1), and select index i if $P_{i-1} \le U < P_i$. This works because the probability of choosing index i is $P_i - P_{i-1} = \pi_i$. But determining i by this algorithm once U is drawn can involve far too many comparisons for the algorithm to be efficient.

The idea of the efficient algorithm we use is first to select an index i from $0, \ldots, n-1$ with *equal* probability weights for each index. The index actually selected as the drawing is then either i or an index y_i specified in a table. Index i is chosen rather than y_i with probability p_i, where p_i is specified in another table. The two tables are set up in such a way that each possible drawing occurs with the desired probability.

An essential feature of this scheme is that the first selection limits the number of possibilities for the final selection to at most two. This allows the choice between i and y_i to be made efficiently: A second random number U' can be generated from U(0,1), and then i is chosen if $0 \le U' \le p_i$, and y_i is chosen otherwise. In practice, we do not really need to generate two random numbers U and U'. We draw one random number V, and set $i = \lfloor nV \rfloor$, thereby making i a drawing from the uniform distribution on $0, \ldots, n-1$. Then $U' = V - i$ is a drawing from U(0,1).

Each index is chosen at the first step with probability $1/n$. Conditional on choosing i, i is kept at the second stage with probability p_i, and so i is chosen in this way with unconditional probability p_i/n. Conditional on any other first choice k, i can be chosen only if $y_k = i$, and then only with probability $1 - p_k$. Thus the total unconditional probability of choosing i is given by

$$\pi_i = n^{-1}\Big(p_i + \sum_{k=1}^{n}(1-p_k)\,\mathrm{I}(y_k = i)\Big). \tag{1}$$

Here, $\mathrm{I}(\cdot)$ is the indicator function. Observe that, if $p_i = 1$, the value of y_i is irrelevant, since if i is chosen at the first stage, it is always kept at the second.

The tables giving the y_i and the p_i are constructed in an iterative fashion. Throughout the successive iterations, a number of things are kept invariant, so that we can guarantee that, at the end of the iterative loop, the equations (1) are satisfied. The loop makes use of a set of nonnegative numbers q_i, $i = 0, \ldots, n-1$, which are initialised so that $q_i = \pi_i$. At intermediate stages, the indices are partitioned into a subset I, containing those indices for which the table entries p_i and y_i have not yet been computed, and a complementary subset J. For any $i \in J$, p_i and y_i have been computed, and are not changed subsequently. For $i \in I$, the relation that constitutes the loop invariant is as follows:

$$\pi_i = q_i\,\mathrm{I}(i \in I) + n^{-1}\Big(p_i\,\mathrm{I}(i \in J) + \sum_{k \in J}(1-p_k)\,\mathrm{I}(y_k = i)\Big). \tag{2}$$

Before starting the loop, the set J is empty and $\pi_i = q_i$ for all i, so that (2) holds initially. Similarly, when I is empty and the tables are complete, we see that (2) is identical with (1), as required. An algorithm that maintains (2) at each iterative step while filling J and emptying I therefore gives a correct set of tables.

If we sum the equations (2) over all values of i, we find that

$$1 = \sum_{i=0}^{n-1}\pi_i = \sum_{i \in I} q_i + n^{-1}\sum_{i \in J} p_i + n^{-1}\sum_{i=0}^{n-1}\sum_{k \in J}(1-p_k)\,\mathrm{I}(y_k = i). \tag{3}$$

On interchanging the order of the sums over i and k in the last term above, we see that that term becomes

$$n^{-1}\sum_{k \in J}(1-p_k)\sum_{i=0}^{n-1}\mathrm{I}(y_k = i).$$

But y_k takes on exactly one value between 0 and $n-1$, and so $\sum_i \mathrm{I}(y_k = i) = 1$. Thus (3) becomes

$$1 = \sum_{i \in I} q_i + n^{-1}\sum_{k \in J} 1 = \sum_{i \in I} q_i + n^{-1}(\#J),$$

where $\#J$ is the number of elements in the subset J. Then, since $\#J = n - \#I$, we conclude that $(\#I)^{-1}\sum_{i \in I} q_i = 1/n$. This means that the average of the elements q_i for $i \in I$ is $1/n$.

We distinguish two cases. If all the q_i for $i \in I$ are equal, they are all equal to the average of $1/n$. In this case, we can simply set $p_i = 1$ for all $i \in I$, transfer all the indices i to the subset J thereby emptying I, and end the loop. We note that the y_i for $i \in I$ are irrelevant. To see that this maintains the required invariant relations (2), observe that, before we perform the operation, (2) for all $i \in J$ says that

$$\pi_i = n^{-1}\Big(p_i + \sum_{k \in J}(1-p_k)\,\mathrm{I}(y_k = i)\Big).$$

After the operation, all the indices previously in I are now in J and so can potentially contribute to the sum over k. However we set $p_k = 1$ for all the indices transferred, and so their contributions are zero. The invariant is thus maintained for $i \in J$. For $i \in I$, (2) before the operation is

$$\pi_i = q_i + n^{-1}\sum_{k \in J}(1-p_k)\,\mathrm{I}(y_k = i).$$

After the operation, (2) becomes

$$\pi_i = n^{-1}\Big(1 + \sum_{k=0}^{n-1}(1-p_k)\,\mathrm{I}(y_k = i)\Big).$$

As before, the sum over k is unchanged because all the new terms are zero, and so, since $q_i = 1/n$, the invariant is maintained. This means that finding that all the q_i for $i \in I$ are equal to $1/n$ can serve as the stopping condition for the iterative loop. In fact, the stopping condition is simply that the smallest q_i for $i \in I$ is equal to $1/n$, since, if the smallest is equal to the average, all are equal to the average.

At the start of each iteration, the indices in I are sorted in increasing order of the q_i. At this point, we check whether the stopping condition is satisfied. After the sort, the smallest q_i for $i \in I$ is indexed by the first element of the set I. If it is equal to $1/n$, we set $p_i = 1$ for all $i \in I$ and transfer all elements of I to J, as discussed above.

If the values of the q_i for $i \in I$ are not all equal, we transfer one element of I to J as follows. The index j for which the table entries p_j and y_j are to be computed in this iteration is chosen to correspond with the smallest of the q_i for $i \in I$. We set the conditional probability p_j equal to n times the unconditional probability q_j. For y_j we take the index $y \in I$ for which q_y is the greatest of the values of the q_i for $i \in I$. The index y_j is the last element of the set I, and so it is different from j.

The rest of the iteration is devoted to maintaining the invariant relations. Since index y_j can now be chosen at the second step if j is chosen at the first, we reduce the probability mass that remains to be attributed to y_j by the amount now associated with j. Unconditionally, this amount is $(1-p_j)/n = 1/n - q_j$. Note that, as the greatest of a set of unequal elements, q_y is strictly greater than the average $1/n$. Thus q_y remains nonnegative after this operation. Finally, we transfer the index j from I to J.

The invariants

It remains to check that these operations do indeed preserve the invariants. The right-hand side of (2) for $i = j$ before p_j and y_j are appended to the tables is equal to

$$q_j + \sum_{k\in J} n^{-1}(1-p_k)\,\mathrm{I}(y_k = j).$$

At the end of the iteration, when $j \in J$, the right-hand side becomes

$$n^{-1}p_j + \sum_{k\in J} n^{-1}(1-p_k)\,\mathrm{I}(y_k = j).$$

But $p_j = nq_j$, and the sum over k is the same for both iterations, since $y_j \neq j$. Thus we conclude that, if (2) holds for $i = j$ at the beginning of the iteration, it does so as well at the end.

The index y_j belongs to I both at the beginning and at the end of the iteration. At the beginning, the right-hand side of (2) for $i = y_j$ is

$$q_{y_j} + \sum_{k\in J} n^{-1}(1-p_k)\,\mathrm{I}(y_k = y_j).$$

At the end of the iteration, $(1-nq_j)/n$ has been subtracted from q_{y_j}. Index j now belongs to J, and so it contributes $n^{-1}(1-p_j)$ to the sum over k. What is subtracted is thus equal to what is added, and so, if (2) holds for $i = y_j$ at the beginning of the iteration, it does so as well at the end.

Consider an index $i \in I$ other than either j or y_j. Then $i \in I$ at the beginning of the iteration if and only if $i \in I$ at the end. Neither p_i nor q_i is changed during the iteration. The terms in the sum over k on the right-hand side of (2) that are present at the beginning are still there, unchanged, at the end, since, once they are computed, p_k and y_k are never subsequently altered. The index i belongs to J at the end, but does not contribute to the sum over k because $y_j \neq i$. Thus the right-hand side of (2) for the i we consider is the same at the beginning and the end. Consequently, (2) holds for all $i \in I$ both at the beginning and at the end of each iteration.

Now let $i \in J$. The value of the sum over k in the right-hand side of (2) is unaltered by the iteration, since j, the new element of J, is such that $y_j \in I$, and so $y_j \neq i$ for any $i \in J$. We have proved that the invariant relation is preserved by the iterative procedure, and that the procedure terminates after at most n iterations.

3. Coding the Algorithm

The class `tiltprobs` in namespace `Tilt` is used to implement the algorithm described in the preceding section. It is declared as follows. The vectors `p_` and `y_` represent the two tables used by the algorithm.

⟨*tiltprobs declaration*⟩≡

```
class tiltprobs
{
   std::vector<double> p_;
   std::vector<size_t> y_;
 public:
   tiltprobs(const std::vector<double>&);
   const std::vector<double>& p() const { return p_; }
   const std::vector<size_t>& y() const { return y_; }
   size_t operator()() const;
};
```

The tables are set up by the constructor, which is supplied with a reference to a vector containing the desired probabilities π_i. In the code below, the set of indices `I` initially contains every index from 0 to $n-1$. The sorting is performed using as the sorting criterion a function object constructed using the Boost lambda library. Note that no check is made of whether the π_i are all nonnegative and sum to one. This is thus a precondition for the constructor.

⟨*tiltprobs constructor*⟩≡

```
Tilt::tiltprobs::tiltprobs(const std::vector<double>& p) :
   p_(p), y_(p.size())
{
   size_t n = p_.size();
   std::vector<size_t> I;
   for (size_t i = 0; i < n; ++i) I.push_back(i);
   std::vector<double> q(p_);
   double eqprob = double(1.0)/n;

   using namespace boost::lambda;
   while (I.size())
   {
      std::sort(I.begin(), I.end(), var(q)[_1] < var(q)[_2]);
      size_t j = I[0];
      double pr = n*q[j];
      if (std::abs(pr-1.0) < 1E-8) break;
      p_[j] = pr;
      size_t y = I.back();
      y_[j] = y;
      q[y] += q[j]-eqprob;
      I.erase(I.begin());
   }
   std::for_each(I.begin(), I.end(), var(p_)[_1] = 1.0);
}
```

The operator() of the class returns drawings from the distribution. Recall that, first, an index i is chosen with probability $1/n$, and then retained as the drawing with probability p_i, the drawing otherwise being y_i.

⟨*tiltprobs operator()*⟩≡

```
size_t Tilt::tiltprobs::operator()() const
{
   double u = (*rng)->uniform();
   u *= p_.size();
   size_t i = size_t(std::floor(u));
   double v = u-i;
   return v < p_[i] ? i : y_[i];
}
```

The pointer rng can be initialised to point to a pointer to whatever random number generator is to be used. It should be an instance of the class Random::RNG.

⟨*Tilt::rng*⟩≡

```
Random::RNG** rng;
```

Source files

The file tilt.h contains the declaration of tiltprobs.

⟨*tilt.h*⟩≡

```
#ifndef _TILT_TILT_H
#define _TILT_TILT_H

#include <vector>
```

```
namespace Tilt
{
   ⟨tiltprobs declaration⟩
}

#endif
```

The code to be compiled is in tilt.cc.

⟨*tilt.cc*⟩≡

```
#include <cmath>
#include <boost/lambda/lambda.hpp>
#include <algorithm>
#include <rng/random.h>
#include "tilt.h"

namespace Tilt { ⟨Tilt::rng⟩ }

⟨tiltprobs constructor⟩
⟨tiltprobs operator()⟩
```

4. The *Ects* Module

The module allows ***Ects*** to construct any number of objects of type tiltprobs. The command that is used for this purpose is settilt. The command is implemented by objects of class settilt in namespace EctsTilt, declared as follows.

⟨*settilt declaration*⟩≡

```
class settilt : public Framework::command
{
   std::string tname_;
   Grammar::Unit<double> pi_;
   void execute() const;
 public:
   settilt(const std::string&);
};
```

The syntax of the command is

    settilt <name> <pi>

where *name* is the name by which the object that is to be constructed can be referred to, and *pi* is an expression, to be evaluated by the rules of mat, the elements of which are the probabilities to be assigned to the outcomes $0, 1, \ldots, n-1$.

The settilt constructor saves the supplied name, objecting if it doesn't find one, and then the expression that is to be evaluated later.

⟨*settilt constructor*⟩≡

```
EctsTilt::settilt::settilt(const std::string& st)
{
   std::istringstream is(st);
   is >> tname_;
   if (tname_.empty())
      throw std::runtime_error(msg(NoName, tilttexts));
   pi_ = matgetunit(Framework::swallow(is>>std::ws,'='));
}
```

The `execute` function evaluates the probabilities, complaining if it finds none, and then checks that none of the probabilities is negative, and that they sum to one. If these checks fail, exceptions are thrown with suitable error messages. If they succeed, a smart pointer to a new `tiltprobs` object is inserted in a table.

⟨*settilt execute*⟩≡

```
void EctsTilt::settilt::execute() const
{
   Matfns::mptr m = mateval(pi_);
   size_t n = m->size();
   if (!n) throw std::runtime_error(msg(NoProbs, tilttexts));
   double minel = m->min();
   if (minel < 0) throw std::runtime_error(msg(NegProb, tilttexts));
   double sumel = m->sum();
   if (std::abs(sumel-1.0) > 1E-12)
      throw std::runtime_error(msg(BadSum, tilttexts));
   TiltTable.replace(tname_,
      boost::shared_ptr<Tilt::tiltprobs>(new Tilt::tiltprobs(
         std::vector<double>(&((*m)(0,0)), &((*m)(n))))));
}
```

The table, called `TiltTable`, is of type `Data::AssocVecSmartPtrTable`.

⟨*TiltTable*⟩≡

```
Data::AssocVecSmartPtrTable<Tilt::tiltprobs> TiltTable;
```

Since this is a module, we need a function that returns a pointer to a newly constructed object of type `settilt` using the `forcmd` argument used for such purposes. This function is `make_settilt`.

⟨*make_settilt*⟩≡

```
Framework::command* make_settilt(Framework::forcmd arg)
{ return new settilt(arg.st); }
```

Generating drawings

The ***Ects*** function `tilt` is made available by the module for use with `set` or `gen`. The function takes a single argument, but this argument is not a

matrix expression, but rather the name of the previously created `tiltprobs` object from which a drawing is to be extracted. Thus we need a very special mechanism for “evaluating” this argument.

The class `settilteval`, derived from `Argeval::argeval` with the usual template arguments, is defined for use with `set`. It is declared as follows. The constructor sets things up so as to accept exactly one argument.

⟨*settilteval declaration*⟩≡

```
class settilteval : public Argeval::argeval<double, Matfns::mptr>
{
   Matfns::arglist Do_eval(const Grammar::Unit<double>&) const;
 public:
    settilteval() : Argeval::argeval<double, Matfns::mptr>(1,1) {}
};
```

The work is done by the private member function `Do_eval`. The argument is a reference to a `unit`, which should have a single argument, which appears to be an identifier, with name the name of the `tiltprobs` object. After having checked that the function indeed has one argument, we try to locate the smart pointer to the appropriate `tiltprobs` object in the table. If nothing is found, we throw an exception. Otherwise, we call the `operator()` of the object to get a scalar drawing that we can return enclosed in a matrix smart pointer pushed as one argument of the argument list.

⟨*settilteval Do_eval*⟩≡

```
Matfns::arglist EctsTilt::settilteval::Do_eval(const Grammar::Unit<double>& u)
   const
{
   checkargs(u);
   const boost::shared_ptr<Tilt::tiltprobs>& tp =
      TiltTable.find(u.arguments()[0].token().text());
   if (!tp.get()) throw std::runtime_error(msg(NoTilt, tilttexts));
   Matfns::arglist arg;
   arg.push_back(Ects::newmatrix((*tp)()));
   return arg;
}
```

We need a constructed object of this class, here named `setev`.

⟨*setev*⟩≡

```
settilteval setev;
```

The declaration of the class `gentilteval` is exactly like that of `settilteval`. The `Do_eval` function is however coded so as to return a vector of independent drawings for the declared sample. The object `genev` is constructed as an instance of the class.

⟨*gentilteval and genev*⟩≡

```
class gentilteval : public Argeval::argeval<double, Matfns::mptr>
{
   Matfns::arglist Do_eval(const Grammar::Unit<double>&) const;
 public:
    gentilteval() : Argeval::argeval<double, Matfns::mptr>(1,1) {}
};

gentilteval genev;
```

The code of the `Do_eval` function starts just like that for `settilteval`. Then a matrix is constructed with enough elements to reach the end of the sample. The elements from the start of the sample to the end are then filled with drawings from the `tiltprobs` object.

⟨*gentilteval Do_eval*⟩≡

```
Matfns::arglist EctsTilt::gentilteval::Do_eval(const Grammar::Unit<double>& u)
   const
{
   checkargs(u);
   const boost::shared_ptr<Tilt::tiltprobs>& tp =
      TiltTable.find(u.arguments()[0].token().text());
   if (!tp.get()) throw std::runtime_error(msg(NoTilt, tilttexts));
   Matfns::arglist arg;
   Matrices::Matrix* ans = new Matrices::Matrix(*smplend,1);
   for (size_t i = *smplstart; i < *smplend; ++i)
      (*ans)(i) = (*tp)();
   arg.push_back(Matfns::mptr(ans));
   return arg;
}
```

Notice that these `argeval` objects have done *all* the required work. Thus the actual function that normally would combine the arguments to give the final answer should just return its first argument. That is what is done by the function `retarg`.

⟨*retarg*⟩≡

```
Matfns::mptr retarg(const Matfns::arglist& a)
{ return a[0]; }
```

The C functions

Every module must have two C functions that constitute its interaction with ***Ects***. The more important is `init`, which takes an argument, supplied by ***Ects***, which points to a structure containing information necessary for the module. Here, we need the variables `smplstart` and `smplend`, and things for evaluating expressions as with `mat`. We set up the `settilt` command, and the `tilt` function for `set` and `gen`. Additionally, we initialise the pointer

`Tilt::rng` to point to the pointer to the current ***Ects*** random number generator.

⟨*Tilt init*⟩≡

```
void init(Modules::module_host* bridge)
{
   EctsTilt::smplstart = bridge->smplstart;
   EctsTilt::smplend = bridge->smplend;
   EctsTilt::mateval = bridge->matevaluate;
   EctsTilt::matgetunit = bridge->matgetunit;
   Tilt::rng = bridge->rangen;

   bridge->cmdfn("settilt", EctsTilt::make_settilt);

   bridge->setectsfn("tilt", EctsTilt::retarg, &EctsTilt::setev);
   bridge->genectsfn("tilt", EctsTilt::retarg, &EctsTilt::genev);
}
```

The lesser of the C functions is `banner`. It returns a reference to one of the module text strings.

⟨*Tilt banner*⟩≡

```
const char* banner()
{
   static std::string st = msg(EctsTilt::notice, EctsTilt::tilttexts);
   return st.c_str();
}
```

The text strings

The module has to set aside a buffer into which the text strings can be inserted from a file. This buffer is called `tilttexts`. It is declared here, along with the `enum` listing the names by which the text strings are identified.

⟨*Tilt texts*⟩≡

```
const char tilttexts[500] = "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!000460";

enum { notice = 1, NoName, NoProbs, NegProb, BadSum, NoTilt, };
```

The text strings used by the module are in the separate file `tilt.txt`.

⟨*tilt.txt*⟩≡

```
Texts for module Tilt
%%%

Ects 4 module Tilt, April 2007

Creates objects that return indices with specified
unequal probabilities
```

```
Copyright (c) Russell Davidson, April 2008.


%%%
No name given for tilted probabilities object
%%%
Could not find any probabilities to initialise object
%%%
At least one negative probability specified
%%%
Specified probabilities do not sum to one
%%%
No object found with specified name
%%%
```

5. The Code File and the Makefile

The code for the interface between ***Ects*** and the Tilt material is in the file ectstilt.cc.

⟨*ectstilt.cc*⟩≡

```
#include "tilt.h"
#include <formod.h>
#include <framework/smarttable.h>
#include <framework/utility.h>
#include <framework/text.h>
#include <boost/smart_ptr.hpp>
#include <sstream>
#include <stdexcept>

extern "C"
{
   const char* banner();
   void init(Modules::module_host*);
}

namespace Tilt { extern Random::RNG** rng; }

namespace EctsTilt
{
   Modules::eval mateval;
   Modules::create_unit matgetunit;
   const size_t* smplstart, *smplend;

   ⟨TiltTable⟩
   ⟨settilt declaration⟩
   ⟨make_settilt⟩
   ⟨settilteval declaration⟩
   ⟨setev⟩
   ⟨gentilteval and genev⟩
   ⟨retarg⟩
```

```
   ⟨Tilt texts⟩
}

using Framework::Text::msg;

⟨settilt constructor⟩
⟨settilt execute⟩
⟨settilteval Do_eval⟩
⟨gentilteval Do_eval⟩

namespace Framework
{
   namespace Text { size_t buflen = 460; }
}

⟨Tilt banner⟩
⟨Tilt init⟩
```

Finally, the Makefile has no surprises.

⟨*tilt Makefile*⟩≡

```
MODNAME=tilt
CC = g++
CFLAGS = -c -O3 -Wall
INCLUDE = -I $(HOME)/ects4/include -I $(HOME)/ects4/web/src
INSTALLDIR = $(HOME)/ects4/lib
OBJS = tilt.lo ectstilt.lo

%.lo: %.cc
        libtool --mode=compile $(CC) $(CFLAGS) $(INCLUDE) $<

all: $(MODNAME).la

$(MODNAME).la: $(OBJS)
        libtool --mode=link g++ -o $@ -module -rpath $(INSTALLDIR) \
-avoid-version -no-undefined $^

$(MODNAME).lo: $(MODNAME).cc $(MODNAME).h
ects$(MODNAME).lo: ects$(MODNAME).cc $(MODNAME).h

install:
        setmsg .libs/$(MODNAME).so ! $(MODNAME).txt
        libtool --mode=install install $(MODNAME).la \
$(INSTALLDIR)/$(MODNAME).la

clean:
        rm -rf *.lo $(MODNAME).la *.bak .libs
```

References

Knuth, D. E. (1998). *The Art of Computer Programming*, Vol 2, Third Edition, Addison-Wesley.

General Index

Type Index

Index of Code Segments