Elementary Probability and the `prob` Package

G. Jay Kerns

January 22, 2009

Contents

1 Preliminaries

This document is designed to get a person up and running doing elementary probability in R using the prob package. In addition to prob, you will want to install the combinat package in order to use a couple of functions, but other than that a base installation of R should be more than enough.

The prerequisites are minimal. The term "data frame" should ring a bell, along with some common tricks like using 1:6 to represent the integers from 1 to 6. That being said, please note that this document is not a treatise on probability. The reader should have a passing familiarity with the basic tenets of probability and some of the topics that are customarily studied to be able to follow along.

In the interest of space, most of the examples described below are uncharacteristically small and do not fully showcase the true computational power of R. Nevertheless, the goal is for these small examples to serve as a starting point from which users can investigate more complicated problems at their leisure.

2 Sample Spaces

The first step is to set up a *sample space*, or set of possible outcomes of an experiment. In the prob package, the most simple form of a sample space is represented by a *data frame*, that is, a rectangular collection of variables. Each row of the data frame corresponds to an outcome of the experiment.

This is the primary way we will represent a sample space, both due to its simplicity and to maximize compatibility with the R Commander by John Fox. However, this structure is not rich enough to describe some of the more interesting probabilistic applications we will encounter. To handle these we will need to consider the more general *list* data structure. See the last section for some remarks in this direction.

Note that the machinery doing the work to generate most of the sample spaces below is the expand.grid() function in the base package, also combn() in combinat and new but related permsn().

2.1 Some Standard Sample Spaces

The prob package has some functions to get one started. For example, consider the experiment of tossing a coin. The outcomes are H and T. We can set up the sample space quicky with the tosscoin() function:

```
> tosscoin(1)
```

```
  toss1
1     H
2     T
```

The number 1 tells tosscoin() that we only want to toss the coin once. We could toss it three times:

```
> tosscoin(3)
```

```
  toss1 toss2 toss3
1     H     H     H
2     T     H     H
3     H     T     H
4     T     T     H
5     H     H     T
6     T     H     T
7     H     T     T
8     T     T     T
```

As an alternative, we could consider the experiment of rolling a fair die:

```
> rolldie(1)

  X1
1  1
2  2
3  3
4  4
5  5
6  6
```

The `rolldie()` function defaults to a 6-sided die, but we can change it with the `nsides` argument. Typing `rolldie(3, nsides = 4)` would be for rolling a 4-sided die three times.

Perhaps we would like to draw one card from a standard set of playing cards (it is a long data frame):

```
> cards()

   rank    suit
1     2    Club
2     3    Club
3     4    Club
4     5    Club
5     6    Club
6     7    Club
7     8    Club
8     9    Club
9    10    Club
10    J    Club
11    Q    Club
12    K    Club
13    A    Club
14    2 Diamond
15    3 Diamond
16    4 Diamond
17    5 Diamond
18    6 Diamond
19    7 Diamond
20    8 Diamond
21    9 Diamond
22   10 Diamond
23    J Diamond
24    Q Diamond
25    K Diamond
26    A Diamond
27    2   Heart
28    3   Heart
29    4   Heart
30    5   Heart
31    6   Heart
32    7   Heart
33    8   Heart
34    9   Heart
35   10   Heart
```

```
36    J   Heart
37    Q   Heart
38    K   Heart
39    A   Heart
40    2   Spade
41    3   Spade
42    4   Spade
43    5   Spade
44    6   Spade
45    7   Spade
46    8   Spade
47    9   Spade
48   10   Spade
49    J   Spade
50    Q   Spade
51    K   Spade
52    A   Spade
```

The `cards()` function that we just used has arguments `jokers` (if you would like Jokers to be in the deck) and `makespace` which we will discuss later.

Additionally, the `roulette()` function gives the standard sample space for one spin on a roulette wheel. There are EU and USA versions available. I would appreciate hearing about any other game or sample spaces that may be of general interest.

2.2 Sampling from Urns

Perhaps the most fundamental of statistical experiments consists of drawing distinguishable objects from an urn. The `prob` package addresses this topic with the `urnsamples(x, size, replace, ordered)` function. The argument `x` represents the urn from which sampling is to be done. The `size` argument tells how large the sample will be. The `ordered` and `replace` arguments are logical and specify how sampling will be performed. We will discuss each in turn. In the interest of saving space, for this example let our urn simply contain three balls, labeled 1, 2, and 3, respectively. We are going to take a sample of size 2.

Ordered, With Replacement

If sampling is with replacement, then we can get any outcome $1, 2, 3$ on any draw. Further, by "ordered" we mean that we shall keep track of the order of the draws that we observe. We can accomplish this in `R` with

```
> urnsamples(1:3, size = 2, replace = TRUE, ordered = TRUE)

  X1 X2
1  1  1
2  2  1
3  3  1
4  1  2
5  2  2
6  3  2
7  1  3
8  2  3
9  3  3
```

Notice that rows 2 and 4 are identical, save for the order in which the numbers are shown. Further, note that every possible pair of the numbers 1 through 3 are listed. This experiment is equivalent to rolling a 3-sided die twice, which we could have accomplished with `rolldie(2, nsides = 3)`.

Ordered, Without Replacement

Here sampling is without replacement, so we may not observe the same number twice in any row. Order is still important, however, so we expect to see the outcomes `1, 2` and `2, 1` somewhere in our data frame as before.

```
> urnsamples(1:3, size = 2, replace = FALSE, ordered = TRUE)

  X1 X2
1  1  2
2  2  1
3  1  3
4  3  1
5  2  3
6  3  2
```

This is just as we expected. Notice that there are less rows in this answer, due to the restricted sampling procedure. If the numbers 1, 2, and 3 represented "Fred", "Mary", and "Sue", respectively, then this experiment would be equivalent to selecting two people of the three to serve as president and vice-president of a company, respectively, and the sample space lists all possible ways that this could be done.

Unordered, Without Replacement

Again, we may not observe the same outcome twice, but in this case, we will only keep those outcomes which (when jumbled) would not duplicate earlier ones.

```
> urnsamples(1:3, size = 2, replace = FALSE, ordered = FALSE)

  X1 X2
1  1  2
2  1  3
3  2  3
```

This experiment is equivalent to reaching in the urn, picking a pair, and looking to see what they are. This is the default setting of `urnsamples()`, so we would have received the same output by simply typing `urnsamples(1:3,2)`.

Unordered, With Replacement

The last possibility is perhaps the most interesting. We replace the balls after every draw, but we do not remember the order in which the draws come.

```
> urnsamples(1:3, size = 2, replace = TRUE, ordered = FALSE)

  X1 X2
1  1  1
2  1  2
3  1  3
4  2  2
5  2  3
6  3  3
```

We may interpret this experiment in a number of alternative ways. One way is to consider this as simply putting two 3-sided dice in a cup, shaking the cup, and looking inside as in a game of Liar's Dice, for instance. Each row of the sample space is a potential pair we could observe. Another equivalent view is to consider each outcome a separate way to distribute two identical golf balls

into three boxes labeled 1, 2, and 3. Regardless of the interpretation, `urnsamples()` lists every possible way that the experiment can conclude.

Note that the urn does not need to contain numbers; we could have just as easily taken our urn to be `x = c("Red", "Blue", "Green")`. But, there is an **important** point to mention before proceeding. Astute readers will notice that in our example, the balls in the urn were *distinguishable* in the sense that each had a unique label to distinguish it from the others in the urn. A natural question would be, "What happens if your urn has indistinguishable elements, for example, what if `x = c("Red", "Red", "Blue")`?" The answer is that `urnsamples()` behaves as if each ball in the urn is distinguishable, regardless of its actual contents. We may thus imagine that while there are two red balls in the urn, the balls are such that we can tell them apart (in principle) by looking closely enough at the imperfections on their surface.

In this way, when the `x` argument of `urnsamples()` has repeated elements, the resulting sample space may appear to be `ordered = TRUE` even when, in fact, the call to the function was `urnsamples(..., ordered = FALSE)`. Similar remarks apply for the `replace` argument. We investigate this issue further in the last section.

3 Counting Tools

The sample spaces we have seen so far have been relatively small, and we can visually study them without much trouble. However, it is VERY easy to generate sample spaces that are prohibitively large. And while `R` is wonderful and powerful and does almost everything except wash windows, even `R` has limits of which we should be mindful.

In many cases, we do not need to actually generate the sample spaces of interest; it suffices merely to count the number of outcomes. The `nsamp()` function will calculate the number of rows in a sample space made by `urnsamples()`, without actually devoting the memory resources necessary to generate the space. The arguments are: `n`, the number of (distinguishable) objects in the urn, `k`, the sample size, and `replace`, `ordered` as above.

In a probability course, one derives the formulas used in the respective scenarios. For our purposes, it is sufficient to merely list them in the following table. Note that $x! = x(x-1)(x-2)\cdots 3\cdot 2\cdot 1$ and $\binom{n}{k} = n!/[k!(n-k)!]$.

Values of `nsamp(n, k, replace, ordered)`

	`ordered = TRUE`	`ordered = FALSE`
`replace = TRUE`	n^k	$\frac{(n-1+k)!}{(n-1)!k!}$
`replace = FALSE`	$\frac{n!}{(n-k)!}$	$\binom{n}{k}$

Examples

We will compute the number of outcomes for each of the four `urnsamples()` examples that we saw in the last section. Recall that we took a sample of size two from an urn with three distinguishable elements.

```
> nsamp(n = 3, k = 2, replace = TRUE, ordered = TRUE)

[1] 9

> nsamp(n = 3, k = 2, replace = FALSE, ordered = TRUE)

[1] 6

> nsamp(n = 3, k = 2, replace = FALSE, ordered = FALSE)
```

```
[1] 3
```

```
> nsamp(n = 3, k = 2, replace = TRUE, ordered = FALSE)
```

```
[1] 6
```

Compare these answers with the length of the data frames generated above.

3.1 The Multiplication Principle

A benefit of `nsamp()` is that it is *vectorized*, so that entering vectors instead of numbers for `n`, `k`, `replace`, and `ordered` results in a vector of corresponding answers. This becomes particularly convenient when trying to demonstrate the Multiplication Principle for solving combinatorics problems.

Example

Question: There are 11 artists who each submit a portfolio containing 7 paintings for competition in an art exhibition. Unfortunately, the gallery director only has space in the winners' section to accomodate 12 paintings in a row equally spread over three consecutive walls. The director decides to give the first, second, and third place winners each a wall to display the work of their choice. The walls boast 31 separate lighting options apiece. How many displays are possible?

Answer: The judges will pick 3 (ranked) winners out of 11 (with `rep=FALSE`, `ord=TRUE`). Each artist will select 4 of his/her paintings from 7 for display in a row (`rep=FALSE`, `ord=TRUE`), and lastly, each of the 3 walls has 31 lighting possibilities (`rep=TRUE`, `ord=TRUE`). These three numbers can be calculated quickly with

```
> n = c(11, 7, 31)
> k = c(3, 4, 3)
> r = c(FALSE, FALSE, TRUE)
```

```
> x = nsamp(n, k, rep = r, ord = TRUE)
```

```
[1]   990   840 29791
```

(Notice that `ordered` is always `TRUE`; `nsamp()` will recycle `ordered` and `replace` to the appropriate length.) By the Multiplication Principle, the number of ways to complete the experiment is the product of the entries of `x`:

```
> prod(x)
```

```
[1] 24774195600
```

Compare this with the some standard ways to compute this in `R`:

```
> (11 * 10 * 9) * (7 * 6 * 5 * 4) * 31^3
```

```
[1] 24774195600
```

or alternatively

```
> prod(9:11) * prod(4:7) * 31^3
```

```
[1] 24774195600
```

or even

```
> prod(factorial(c(11, 7))/factorial(c(8, 3))) * 31^3
```

```
[1] 24774195600
```

As one can guess, in many of the standard counting problems there aren't much savings in the amount of typing; it is about the same using `nsamp()` versus `factorial()` and `choose()`. But the virtue of `nsamp()` lies in its collecting the relevant counting formulas in a one-stop shop. Ultimately, it is up to the user to choose the method that works best for him/herself.

4 Defining a Probability Space

Once a sample space is defined, the next step is to associate a probability model with it in order to be able to answer probabilistic questions. Formally speaking, a *probability space* is a triple $(S, \mathcal{B}, \mathbb{P})$, where S is a sample space, $\mathcal{B}$ is a sigma-algebra of subsets of S, and $\mathbb{P}$ is a probability measure defined on $\mathcal{B}$. However, for our purposes all of the sample spaces are finite, so we may take $\mathcal{B}$ to be the power set (the set of all subsets of S) and it suffices to specify $\mathbb{P}$ on the elements of S, the outcomes. The only requirement for $\mathbb{P}$ is that its values should be nonnegative and sum to 1.

The end result is that in the `prob` package, a probability space is an object of outcomes `S` and a vector of probabilities (called "`probs`") with entries that correspond to each outcome in `S`. When `S` is a data frame, we may simply add a column called `probs` to `S` and we will be finished; the probability space will simply be a data frame which we may call `space`. In the case that S is a list, we may combine the `outcomes` and `probs` into a larger list, `space`; it will have two components: `outcomes` and `probs`. The only requirement we place is that the entries of `probs` be nonnegative and `sum(probs)` is one.

To accomplish this in `R`, we may use the `probspace()` function. The general syntax is `probspace(x, probs)`, where `x` is a sample space of outcomes and `probs` is a vector (of the same length as the number of outcomes in `x`). The specific choice of `probs` depends on the context of the problem, and some examples follow to demonstrate some of the more common choices.

4.1 Examples

The Equally Likely Model

The equally likely model asserts that every outcome of the sample space has the same probability, thus, if a sample space has n outcomes, then `probs` would be a vector of length n with identical entries $1/n$. The quickest way to generate `probs` is with the `rep()` function. We will start with the experiment of rolling a die, so that $n = 6$. We will construct the sample space, generate the `probs` vector, and put them together with `probspace()`.

```
> outcomes = rolldie(1)

  X1
1  1
2  2
3  3
4  4
5  5
6  6

> p = rep(1/6, times = 6)

[1] 0.1666667 0.1666667 0.1666667 0.1666667 0.1666667 0.1666667

> probspace(outcomes, probs = p)

  X1     probs
1  1 0.1666667
2  2 0.1666667
3  3 0.1666667
4  4 0.1666667
5  5 0.1666667
6  6 0.1666667
```

The `probspace()` function is designed to save us some time in many of the most common situations. For example, due to the especial simplicity of the sample space in this case, we could have achieved the same result with simply (note the name change for the first column)

```
> probspace(1:6, probs = p)
```

```
  x     probs
1 1 0.1666667
2 2 0.1666667
3 3 0.1666667
4 4 0.1666667
5 5 0.1666667
6 6 0.1666667
```

Further, since the equally likely model plays such a fundamental role in the study of probability, the `probspace()` function will assume that the equally model is desired if no `probs` are specified. Thus, we get the same answer with only

```
> probspace(1:6)
```

```
  x     probs
1 1 0.1666667
2 2 0.1666667
3 3 0.1666667
4 4 0.1666667
5 5 0.1666667
6 6 0.1666667
```

And finally, since rolling dice is such a common experiment in probability classes, the `rolldie()` function has an additional logical argument `makespace` that will add a column of equally likely `probs` to the generated sample space:

```
> rolldie(1, makespace = TRUE)
```

```
  X1     probs
1  1 0.1666667
2  2 0.1666667
3  3 0.1666667
4  4 0.1666667
5  5 0.1666667
6  6 0.1666667
```

or just `rolldie(1:6,TRUE)`. Many of the other sample space functions (`tosscoin()`, `cards()`, `roulette()`, *etc.*) have similar `makespace` arguments. Check the documentation for details.

One sample space function that does NOT have a `makespace` option is the `urnsamples()` function. This was intentional. The reason is that under the varied sampling assumptions the outcomes in the respective sample spaces are NOT, in general, equally likely. It is important for the user to carefully consider the experiment to decide whether or not the outcomes are equally likely, and then use `probspace()` to assign the model.

An unbalanced coin

While the `makespace` argument to `tosscoin()` is useful to represent the tossing of a *fair* coin, it is not always appropriate. For example, suppose our coin is not perfectly balanced, for instance, maybe the "H" side is somewhat heavier such that the chances of a H appearing in a single toss is 0.70 instead of 0.5. We may set up the probability space with

```
> probspace(tosscoin(1), probs = c(0.7, 0.3))
```

```
  toss1 probs
1     H   0.7
2     T   0.3
```

The same procedure can be used to represent an unbalanced die, roulette wheel, *etc.*

4.2 Independent, Repeated Experiments

In some situations it is desired to repeat a certain experiment multiple times under identical conditions and in an independent manner. We have seen many examples of this already: tossing a coin repeatedly, rolling a die or dice, *etc.*

Example: An unbalanced coin continued

It was easy enough to set up the probability space for one toss, however, the situation becomes more complicated when there are multiple tosses involved. Clearly, the outcome HHH should not have the same probability as TTT, which should again not have the same probability as HTH.

At the same time, there is symmetry in the experiment in that the coin does not remember the face it shows from toss to toss, and it is easy enough to toss the coin in a similar way repeatedly.

The `iidspace()` function was designed specifically for this situation. It has three arguments: `x` which is a vector of outcomes, `ntrials` which is an integer telling how many times to repeat the experiment, and `probs` to specify the probabilities of the outcomes of `x` in a single trial. We may represent tossing our unbalanced coin three times with the following:

```
> iidspace(c("H", "T"), ntrials = 3, probs = c(0.7, 0.3))
```

```
  X1 X2 X3 probs
1  H  H  H 0.343
2  T  H  H 0.147
3  H  T  H 0.147
4  T  T  H 0.063
5  H  H  T 0.147
6  T  H  T 0.063
7  H  T  T 0.063
8  T  T  T 0.027
```

As expected, the outcome HHH has the largest probability, while TTT has the smallest. (Since the trials are independent, $\mathbb{P}(HHH) = 0.7^3$ and $\mathbb{P}(TTT) = 0.3^3$, *etc.*) Note that the result of the function call is a probability space, not a sample space (which we could construct already with the `tosscoin()` or `urnsamples()` functions). The same procedure could be used to represent an unbalanced die, or any experiment that can be represented with a vector of possible outcomes.

Note that `iidspace()` will assume `x` has equally likely outcomes if no `probs` argument is specified. Also note that the argument `x` is a *vector*, not a data frame. Trying something like `iidspace(tosscoin(1),...)` will give an error.

4.3 Words of Warning

It should be mentioned that while the splendour of `R` is uncontested, it, like everything else, has limits both with respect to the sample/probability spaces it can manage and with respect to the finite accuracy of the representation of most numbers (see the `R` FAQ 7.31). When playing around with probability, it is tempting to try to set up a probability space of tossing 100 coins or rolling 50 dice, and then attempt to answer some scintillating questions. (Bear in mind: rolling a die just 9 times has a sample space with over *10 million* outcomes.)

Alas! Even if there were enough RAM to barely hold the sample space (and there were enough time to wait for it to be generated), the infinitesimal probabilities that are associated with SO MANY outcomes make it difficult for the underlying machinery to handle reliably. In some cases, special algorithms need to be called just to give something that holds asymptotically. User beware.

5 Subsets and Set Algebra

A sample space contains all possible outcomes of an experiment. An *event* represents a subset of the sample space, that is, a certain subcollection of outcomes.

5.1 Finding Subsets

There are plenty of existing methods for finding subsets of data frames, so we will only mention some of them here in passing.

Given a data frame sample/probability space `S`, we may extract rows using the `[]` operator:

```
> S = tosscoin(2, makespace = TRUE)

  toss1 toss2 probs
1     H     H  0.25
2     T     H  0.25
3     H     T  0.25
4     T     T  0.25

> S[1:3, ]

  toss1 toss2 probs
1     H     H  0.25
2     T     H  0.25
3     H     T  0.25

> S[c(2, 4), ]

  toss1 toss2 probs
2     T     H  0.25
4     T     T  0.25
```

and so forth. We may also extract rows that satisfy a logical expression using the `subset()` function, for instance

```
> S = cards()

> subset(S, suit == "Heart")

   rank  suit
27    2 Heart
28    3 Heart
29    4 Heart
30    5 Heart
31    6 Heart
32    7 Heart
33    8 Heart
34    9 Heart
35   10 Heart
36    J Heart
37    Q Heart
38    K Heart
39    A Heart

> subset(S, rank %in% 7:9)

   rank    suit
6     7    Club
7     8    Club
8     9    Club
19    7 Diamond
20    8 Diamond
```

```
21    9 Diamond
32    7   Heart
33    8   Heart
34    9   Heart
45    7   Spade
46    8   Spade
47    9   Spade
```

We could continue indefinitely. Also note that mathematical expressions are allowed:

```
> subset(rolldie(3), X1 + X2 + X3 > 16)
```

```
    X1 X2 X3
180  6  6  5
210  6  5  6
215  5  6  6
216  6  6  6
```

5.2 Functions for Finding Subsets

It does not take long before the subsets of interest become more and more complicated to specify. Yet the main idea is that we have a particular logical condition that we would like to apply to each row. If the row satisfies the condition, then it should be in the subset and it should not be in the subset otherwise.

There are many ways to construct logical expressions, and there is an abundance of material that does a great job describing how to go about it. The ease with which the expression may be coded depends of course on the question being asked. The `prob` package contributes a few functions that help to answer questions typically encountered in an elementary probability course.

The function `isin()`

Experienced users of `R` will no doubt be familiar with the function `%in%`. It can be used in the following way.

```
> x = 1:10
> y = 3:7
> y %in% x
```

```
[1] TRUE TRUE TRUE TRUE TRUE
```

Notice that the value is a vector of length 5 which tests if each element of `y` is in `x`, in turn. But perhaps we would like to know whether the *whole* vector `y` is in `x`. We can do this with the `isin()` function.

```
> isin(x, y)
```

```
[1] TRUE
```

Of course, one may ask why we did not try something like `all(y %in% x)`, which would give a single result, `TRUE`. The reason is that the answers are different in the case that `y` has repeated values. Compare:

```
> x = 1:10
> y = c(3, 3, 7)

> all(y %in% x)
```

```
[1] TRUE

> isin(x, y)

[1] FALSE
```

The reason is of course that `x` contains the value 3, but it doesn't have two 3's. The difference becomes significant when rolling multiple dice, playing cards, *etc.* Note that there is an argument `ordered`, which tests whether the elements of `y` appear in `x` in the order in which they are specified in `y`. The consequences are

```
> isin(x, c(3, 4, 5), ordered = TRUE)

[1] TRUE

> isin(x, c(3, 5, 4), ordered = TRUE)

[1] FALSE
```

The connection to probability is that we are given a data frame sample space and we would like to find a subset of the space. A `data.frame` method has been written for `isin()` that simply applies the function to each row of the data frame. Certainly, one could extend the function to work for columns or entries instead of rows, but simplicity is a primary goal of the `prob` package. It is easy enough for experienced users to modify the code as desired to serve their purposes.

We can see the method in action with the following:

```
> S = rolldie(4)
> subset(S, isin(S, c(2, 2, 6), ordered = TRUE))

     X1 X2 X3 X4
188   2  2  6  1
404   2  2  6  2
620   2  2  6  3
836   2  2  6  4
1052  2  2  6  5
1088  2  2  1  6
1118  2  1  2  6
1123  1  2  2  6
1124  2  2  2  6
1125  3  2  2  6
1126  4  2  2  6
1127  5  2  2  6
1128  6  2  2  6
1130  2  3  2  6
1136  2  4  2  6
1142  2  5  2  6
1148  2  6  2  6
1160  2  2  3  6
1196  2  2  4  6
1232  2  2  5  6
1268  2  2  6  6
```

There are a few other functions written to find useful subsets, namely, `countrep()` and `isrep()`. Essentially these were written to test for (or count) a specific number of designated values in outcomes. See the documentation for details.

5.3 Set Union, Intersection, and Difference

Given subsets A and B, it is often useful to manipulate them in an algebraic fashion. To this end, we have three set operations at our disposal: union, intersection, and difference. Below is a table summarizing the pertinent information about these operations.

Name	Denoted	Defined by elements	Code
Union	$A \cup B$	in A or B or both	`union(A,B)`
Intersection	$A \cap B$	in both A and B	`intersect(A,B)`
Difference	$A \backslash B$	in A but not in B	`setdiff(A,B)`

Some examples follow.

```
> S = cards()
> A = subset(S, suit == "Heart")
> B = subset(S, rank %in% 7:9)
```

We can now do some set algebra:

```
> union(A, B)
```

```
   rank    suit
6     7    Club
7     8    Club
8     9    Club
19    7 Diamond
20    8 Diamond
21    9 Diamond
27    2   Heart
28    3   Heart
29    4   Heart
30    5   Heart
31    6   Heart
32    7   Heart
33    8   Heart
34    9   Heart
35   10   Heart
36    J   Heart
37    Q   Heart
38    K   Heart
39    A   Heart
45    7   Spade
46    8   Spade
47    9   Spade
```

```
> intersect(A, B)
```

```
   rank  suit
32    7 Heart
33    8 Heart
34    9 Heart
```

```
> setdiff(A, B)
```

```
   rank  suit
27    2 Heart
28    3 Heart
```

```
29    4 Heart
30    5 Heart
31    6 Heart
35   10 Heart
36    J Heart
37    Q Heart
38    K Heart
39    A Heart
```

```
> setdiff(B, A)
```

```
   rank    suit
6     7    Club
7     8    Club
8     9    Club
19    7 Diamond
20    8 Diamond
21    9 Diamond
45    7   Spade
46    8   Spade
47    9   Spade
```

Notice that `setdiff()` is not symmetric. Further, note that we can calculate the *complement* of a set A, denoted A^c and defined to be the elements of S that are not in A simply with `setdiff(S,A)`.

While our example did not have a `probs` column, the set algebra functions operate the same way even if one had been there (although, in that case one would need to be careful that the events were each subsets of the *same probability space*, or some strange answers may result).

There have been methods written for `intersect()`, `setdiff()`, `subset()`, and `union()` in the case that the input objects are of class `ps`. See Section 9.1.

Note:

When you loaded the `prob` package, you most certainly noticed the message: "`The following object(s) are masked from package:base: intersect setdiff, subset, union`". The reason is that there already exist methods for the functions `union()`, `intersect()`, and `setdiff()` in the `base` package which ships with `R`. However, these methods were designed for when the arguments are vectors of the same mode. Since we are manipulating probability spaces consisting of data frames and lists, it was necessary to write methods to handle those cases as well. When the `prob` package is loaded, `R` recognizes that there are multiple versions of the same function in the search path and acts to shield the new definitions from the existing ones. But there is no cause for alarm, thankfully, because the `prob` functions have been carefully defined to be the usual `base` package definition in the case that the arguments are vectors.

6 Calculating Probabilities

Now that we have ways to find subsets, we can at last move to calculating the probabilities associated with them. This is accomplished with the `prob()` function.

Consider the experiment of drawing a card from a standard deck of playing cards. Let's denote the probability space associated with the experiment as S, and let the subsets A and B be defined by the following:

```
> S = cards(makespace = TRUE)
> A = subset(S, suit == "Heart")
> B = subset(S, rank %in% 7:9)
```

Now it is easy to calculate

```
> prob(A)

[1] 0.25
```

Note that we can get the same answer with

```
> prob(S, suit == "Heart")

[1] 0.25
```

We also find `prob(B) = 0.23` (listed here approximately, but 12/52 actually) and `prob(S)=1`. In essence, the `prob()` function operates by summing the `probs` column of its argument. It will find subsets "on-the-fly" if desired.

We have as yet glossed over the details. More specifically, `prob()` has three arguments: `x` which is a probability space (or a subset of one), `event` which is a logical expression used to define a subset, and `given` which is described in the next subsection.

WARNING. The `event` argument is used to define a subset of `x`, that is, the only outcomes used in the probability calculation will be those that are elements of `x` and satisfy `event` simultaneously. In other words, `prob(x,event)` calculates `prob(intersect(x, subset(x, event)))`. Consequently, `x` should be the entire probability space in the case that `event` is non-null.

6.1 Conditional Probability

The conditional probability of the subset A given the event B is defined to be

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}, \quad \text{if } \mathbb{P}(B) > 0.$$

We already have all of the machinery needed to compute conditional probability. All that is necessary is to use the `given` argument to the `prob()` function, which will accept input in the form of a data frame or a logical expression. Using the above example with S={draw a card}, A ={`suit = "Heart"`}, and B={`rank` is `7, 8, or 9`}.

```
> prob(A, given = B)

[1] 0.25

> prob(S, suit == "Heart", given = rank %in% 7:9)

[1] 0.25

> prob(B, given = A)

[1] 0.2307692
```

Of course, we know that given the event `B` has occurred (a 7, 8, 9 has been drawn), the probability that a Heart has been drawn is 1/4. Similarly, given that the `suit` is "Heart", there are only three out of the 13 Hearts that are a 7, 8, or 9, so $\mathbb{P}(B|A) = 3/13$. We can compute it by going back to the definition of conditional probability:

```
> prob(intersect(A, B))/prob(B)

[1] 0.25
```

We have seen that there is some flexibility in the `given` argument in that it can be either a data frame or it can be a logical expression that defines the subset. HOWEVER, that flexibility is limited. In particular, if `given` is a logical expression, then `event` must also be specified (also a logical expression). And in this case, the argument `x` should be the entire sample space, not a subset thereof, for the reason described in the last section.

Pedagogical Notes

We can now begin to reap the benefits of this framework. Suppose we would like to see an example of the General Addition Rule:

```
> prob(union(A, B))

[1] 0.4230769

> prob(A) + prob(B) - prob(intersect(A, B))

[1] 0.4230769
```

Or perhaps the Multiplication Rule:

```
> prob(intersect(A, B))

[1] 0.05769231

> prob(A) * prob(B, given = A)

[1] 0.05769231
```

We could give evidence that consecutive trials of flipping a coin are independent:

```
> S = tosscoin(2, makespace = TRUE)

> prob(S, toss2 == "H")

[1] 0.5

> prob(S, toss2 == "H", given = toss1 == "H")

[1] 0.5
```

There are many topics available for investigation. Keep in mind, however, that the point is not that `R` (or any other software package, for that matter) will ever be an effective surrogate for critical thinking; rather, the point is that statistical tools like `R` serve to change the classroom landscape, hopefully for the better. Plus, it's free.

7 Simulating Experiments

Given a probability space, how do we actually perform the experiment in `R`? The `prob` package handles this with the `sim()` function. The only arguments are `x` for the probability space and `ntrials` for the desired number of repetitions.

We already have all of the outcomes contained in the probability space, and we know each outcome's probability from the `probs` column. Thus, the only task is to sample the rows of `x` according to their respective probabilities, which is deftly perfomed with the existing `sample()` function. Essentially, the `sim()` function is a wrapper for `sample()`, except that it strips the `probs` column from the result and renames the rownames of the data frame consecutively from `1:n`.

Suppose, for example, that the experiment consisted of picking a single card at random from a standard deck of playing cards. The code would look something like the following.

```
> S = cards(makespace = TRUE)
> sim(S, ntrials = 5)
```

```
  rank    suit
1    8   Spade
2    K   Heart
3    K   Heart
4    J   Heart
5    6 Diamond
```

We only simulated the experiment 5 times in the interest of space; a person could simulate as many trials as desired (within reason). Note that when the sample space has more than 250 reasonably likely rows then a special procedure (Walker's alias method) is used to perform the sampling. See `?sample` and the reference to Ripley (1987) listed there.

7.1 Comparing Theory with Practice

Some may be skeptical of the results from `sim()`. After all, how does one know that the sampling procedure is going according to plan? The answer is: we don't, at least not for certain. But we can get a pretty good idea using the `empirical()` function.

The function works by adding a `probs` column to the simulated data frame (called `sims`, say) with equally likely entries of $1/n$, where n is the number of rows. Then it aggregates the duplicated rows of `sims`, all the while accumulating the probabilities associated with each. The final result will be a data frame containing the unique rows of `sims`, together with a column `probs` at the end containing the respective observed relative frequencies of the rows.

Let's look at an example. For simplicity suppose the experiment is tossing a coin twice. We will construct the probability space, simulate 50,000 double coin flips, and `empirical()` the result.

```
> S = tosscoin(2, makespace = TRUE)
> sims = sim(S, ntrials = 50000)
> empirical(sims)

  toss1 toss2   probs
1     H     H 0.24946
2     T     H 0.25174
3     H     T 0.24870
4     T     T 0.25010
```

We see that each outcome has an observed relative frequency fairly close to 0.25, as we would expect. Of course, this doesn't prove that the sampling is perfect, but it is good news nonetheless.

8 Adding Random Variables

We have seen how to construct sample spaces, find subsets, define probability spaces, and simulate experiments. But, sometimes we are not interested so much in the experiment itself, rather, we care more about a certain numerical quantity associated with the experiment, also known as a *random variable*. This section describes how to define random variables based on an experiment and how to examine their behavior once defined.

The primary vessel we use for this task is the `addrv()` function. There are two ways to use it, and will describe both.

8.1 Supplying a Defining Formula

The first method is based on the existing `transform()` function written by Peter Dalgaard. See `?transform`. The idea is to write a formula defining the random variable inside the function, and it will be added as a column to the data frame. As an example, let us roll a 4-sided die three times, and let's define the random variable $U = X1 - X2 + X3$.

```
> S = rolldie(3, nsides = 4, makespace = TRUE)
> S = addrv(S, U = X1 - X2 + X3)
```

Now let's take a look at the values of U. In the interest of space, we will only reproduce the first few rows of S (there are $4^3 = 64$ rows in total).

```
> S[1:9, ]
```

```
  X1 X2 X3  U    probs
1  1  1  1  1 0.015625
2  2  1  1  2 0.015625
3  3  1  1  3 0.015625
4  4  1  1  4 0.015625
5  1  2  1  0 0.015625
6  2  2  1  1 0.015625
7  3  2  1  2 0.015625
8  4  2  1  3 0.015625
9  1  3  1 -1 0.015625
```

We see by looking at the U column it is operating just like it should. We can now answer questions like

```
> prob(S, U > 6)
```

```
[1] 0.015625
```

8.2 Supplying a Function

Sometimes we have a function laying around that we would like to apply to some of the outcome variables, but it is unfortunately tedious to write out the formula defining what the new variable would be. The `addrv()` function has an argument `FUN` specifically for this case. Its value should be a legitimate function from `R`, such as `sum`, `mean`, `median`, *etc.* Or, you can define your own function. Continuing the previous example, let's define $V = \max(X1, X2, X3)$ and $W = X1 + X2 + X3$.

```
> S = addrv(S, FUN = max, invars = c("X1", "X2", "X3"), name = "V")
> S = addrv(S, FUN = sum, invars = c("X1", "X2", "X3"), name = "W")
> S[1:9, ]
```

```
  X1 X2 X3  U V W    probs
1  1  1  1  1 1 3 0.015625
2  2  1  1  2 2 4 0.015625
3  3  1  1  3 3 5 0.015625
4  4  1  1  4 4 6 0.015625
5  1  2  1  0 2 4 0.015625
6  2  2  1  1 2 5 0.015625
7  3  2  1  2 3 6 0.015625
8  4  2  1  3 4 7 0.015625
9  1  3  1 -1 3 5 0.015625
```

Notice that `addrv()` has an `invars` argument to specify exactly to which columns one would like to apply the function `FUN`. If no input variables are specified, then `addrv()` will apply `FUN` to all non-`probs` columns. Further, `addrv()` has an optional argument `name` to give the new variable; this can be useful when adding several random variables to a probability space (as above). If not specified, the default name is "X".

8.3 Marginal Distributions

As we can see above, often after adding a random variable V to a probability space one will find that V has values that are repeated, so that it becomes difficult to understand what the ultimate behavior of V actually is. We can use the `marginal()` function to aggregate the rows of the sample space by values of V, all the while accumulating the probability associated with V's distinct values. Continuing our example from above, suppose we would like to focus entirely on the values and probabilities of $V = \max(X1, X2, X3)$.

```
> marginal(S, vars = "V")

  V    probs
1 1 0.015625
2 2 0.109375
3 3 0.296875
4 4 0.578125
```

We could save the probability space of V in a data frame and study it further, if we wish. As a final remark, we can calculate the marginal distributions of multiple variables desired using the `vars` argument. For example, suppose we would like to examine the joint distribution of V and W.

```
> marginal(S, vars = c("V", "W"))

   V  W    probs
1  1  3 0.015625
2  2  4 0.046875
3  2  5 0.046875
4  3  5 0.046875
5  2  6 0.015625
6  3  6 0.093750
7  4  6 0.046875
8  3  7 0.093750
9  4  7 0.093750
10 3  8 0.046875
11 4  8 0.140625
12 3  9 0.015625
13 4  9 0.140625
14 4 10 0.093750
15 4 11 0.046875
16 4 12 0.015625
```

Note that the default value of `vars` is the names of all columns except `probs`. This can be useful if there are duplicated rows in the probability space; see the next section.

9 Additional Topics

9.1 More Complicated Sample Spaces

A Poker Case Study

This subsection is for those who are interested in calculating probability on more complicated spaces than those represented by a simple data frame. We will walk through the example of a game of poker because it highlights some of the issues encountered when investigating more complicated sample spaces.

We have discussed the case of drawing a single card from a standard deck. In that case, each card has two properties: a `rank` and `suit`. We can ask questions about either or both to define subsets and calculate probabilities.

But now let us imagine a standard poker hand of five cards (Five Card Stud, for example). How may we represent a typical outcome? One way is to make 10 columns in a data frame. The first two could be the `suit` and `rank` of the first card, the next two could be the `suit` and `rank` of the second card, and so forth.

This representation is undesirable for two reasons. The first is that the order of draws in a poker hand is irrelevant for the Five Card Stud game. Thus, we would need to disregard the order of the entries with respect to columns, while keeping the columns paired with respect to the draws, which is not tractable to perform in `R`. The second reason is that many of the important properties of a poker hand (are all of the `suits` the same? do the `ranks` form an ascending sequence?, *etc.*) are not easy to examine when the data are split between columns. For instance, to see if we had a flush, we would need to check the value of the `suit` in columns 2, 4, 6, 8, and 10. To combine this question with something about `rank` makes it even more complicated.

It would be nice if all five cards in the hand could be represented by a data frame, with the two columns `rank` and `suit`, and five rows, one for each card in the hand. It would be easier to ask questions of a particular outcome (are all rows of `suit` the same? *etc.*), and we already have been considering the rows of a data frame sample space to be unordered. Of course, the sample space of all possible five card hands would now be the set of all possible data frames, dimension 5x2, consisting of respective rows of the `cards()` sample space.

But this last task could be easily achieved by doing `urnsamples(replace = FALSE , ordered = FALSE)` on the rows of `cards()`. Samples would be of size 5. And for each possible sample, we construct a 5x2 data frame of the respective rows and store the results. Where to store them? In a *list*, called `outcomes`. Each entry of the list will be a data frame, conceptually, a five card hand.

Once we have the `outcomes`, to each entry we should associate a probability, which we could store in a vector called `probs`. To construct a probability space, we combine the list `outcomes` and the vector `probs` into a master list, which we call `space`.

Details

The preceding discussion has motivated the definition of what in the `prob` package is the most general probability space: a list containing the two components `outcomes` and `probs`. The component `outcomes` is a list containing elements of the most arbitrary sort; they can be data frames, vectors, more lists, whatever. The `probs` component is a vector (of the same length as `outcomes`), which associates to each element of `outcomes` a nonnegative number. The only additional requirement is that the `sum` of `probs` be one.

Of course, in most cases of interest the elements of `outcomes` will have a decidedly regular structure, to facilitate asking useful questions. To handle experiments similar in format to the poker example, the `urnsamples()` function has been made generic and a data frame method has been added. In this way, one can `urnsample` from a data frame `x`, and a list of `outcomes` is the returned value. Each entry of `outcomes` is a data frame with the same columns as `x`, simply with the number of rows equaling `size`.

In addition, the `probspace()` function is also generic, and has a list method to construct probability spaces from an input list `x` and a vector of probabilities `probs`. If no `probs` are specified, then `probspace()` assumes that the equally likely model was intended and automatically generates a vector of the correct length and with the correct entries.

The output object has class `ps`, to represent a probability space. It also has a secondary class of `list`, so that the `list` methods for other useful functions (such as `print()`) will work in the case that no `ps` method is found.

Example

We will walk through an example to get you started making more general sample spaces, and we will continue with the poker example above with one modification. For our simple poker game, we will only draw two cards instead of five. (Even two cards takes a few seconds, and five cards is incredibly large, over 2.5 million outcomes. See Section 4.3.)

```
> L = cards()
> M = urnsamples(L, 2)
> N = probspace(M)
```

The first command initializes the set of 52 cards into the data frame `L`. The second command generates the set of all possible 2 card hands from the deck and stores them in `M`; note that `M` is a list of length 1326. The third command takes `M` and combines it with a vector `probs`, which is by default assumed to represent the equally likely model. The result is the probability space `N`.

We then define some subsets so that we can do probability. The next four commands set up the four events `A`, `B`, `C`, and `D`.

```
> A = subset(N, all(suit == "Heart"))
> B = subset(N, any(suit == "Heart"))
> C = subset(N, any(suit == "Heart") & all(rank == "A"))
> D = subset(N, any(suit == "Spade") & any(rank == "A"))
```

Now let's do some probability:

```
> prob(A)

[1] 0.05882353

> prob(B)

[1] 0.4411765

> prob(C, given = D)

[1] 0.01149425

> prob(D, given = C)

[1] 0.3333333
```

We can use `all(suit=="Heart")` to check for a flush, for instance. Note that general sample spaces constructed in the above fashion tend to get very large, very quickly. This has consequences when it comes time to do set algebra and compute conditional probability. Be warned that these can use a large amount of computing resources. Since the theme of the `prob` package leans toward simplicity over efficiency, users interested in doing authentically complicated probability problems would be advised to streamline the `prob` package functions for their own uses.

9.2 Discrete Multivariate Distributions

It is plain to see that while we were doing all of this talking about "outcomes", "sample spaces", *etc.*, what we were really talking about were discrete multivariate distributions with finite support (strictly speaking, only when all of the outcomes are numeric). The multinomial distribution is one example, and there are infinitely many others.

It is customary to represent the support of an n dimensional discrete random variable as a suitable subset of an n dimensional lattice or Euclidean space, and it seems like some of the existing `R` packages for discrete multivariate distributions have made progress in this area. We hope that the `prob` package helps for further development of this important topic.

For a simple example, let's set up a standard multinomial probability space. Our experiment will be taking a sample, with replacement, of size 2 from an urn with three balls inside, labeled 1, 2, and 3, respectively. The random variables `X1`, `X2`, and `X3` will be the respective counts of the balls labelled 1, 2, and 3 in our sample (do you see how to do this the long way?).

```
> library(combinat)
> out = data.frame(t(xsimplex(3, 2)))
> p = apply(out, MARGIN = 1, FUN = dmultinom, prob = c(1, 1, 1))

> S = probspace(out, p)

  X1 X2 X3     probs
1  2  0  0 0.1111111
2  1  1  0 0.2222222
3  1  0  1 0.2222222
4  0  2  0 0.1111111
5  0  1  1 0.2222222
6  0  0  2 0.1111111
```

The first command sets up the sample space. The `xsimplex()` function (in the `combinat` package) constructs a 3x6 matrix of all points on a {3,2} simplex lattice. The transpose function `t()` transposes the matrix to be 6x3, so that the outcomes will be rows instead of columns. The `data.frame()` function coerces the matrix into a data frame, in the format of a `prob` sample space.

The next line defines the `probs` vector associated with the outcomes, and it is done with the `apply()` function. The idea is to apply the `dmultinom()` function to each row (`MARGIN=1`) of the data frame `out`. In general, given any multivariate probability mass function, we can use the `FUN` argument of `apply()` to define the `probs` vector.

Once we set up the probability space `S`, we are free to study it in detail. We can simulate, calculate marginal distributions, measure correlations, *etc.*

9.3 (In)distinguishable Elements

We hinted at the end of Section 2 that some interesting things happen when sampling from an urn with repeated elements. As we remarked there, as far as `urnsamples()` is concerned, **all** elements of the urn are distinguishable, even if they may not appear to be to our naked eye. (The reason is that `urnsamples()` operates on the basis of the *indices* of the elements of the urn `x`, and not actually on the elements themselves. This can be very useful for keeping track of the elements, but gives some surprises if the user is not careful.)

Let's look at a concrete example. We will sample from an urn that has 4 red balls and 10 blue balls inside. We will sample without replacement, and in an unordered fashion (the default behavior of `urnsamples`).

```
> x = rep(c("Red", "Blue"), times = c(4, 10))
> P = urnsamples(x, size = 3, replace = FALSE, ordered = FALSE)
> Q = probspace(P)
```

So far, so good. If I were to ask the reader about the experiment just described, (s)he would likely say, "Of course! This corresponds to the hypergeometric model, where `Red` constitutes a success, and in the case that we are interested in the number of `Red`s in our sample". However, if we take a look at the probability space `Q`, then we are in for quite a surprise! It has 364 elements, most of which being all `Red` or all `Blue`. So what happened? The answer is that while many outcomes `Blue Blue Blue` may *look* the same listed in the data frame `Q`, these were derived from actually many, many `Blue`s in separate positions in `x`. Imagine that we took out a large magnifying glass and took a closer look: many of the `Blue`s that look the same from a distance reveal themselves

to be quite different upon closer examination. Again, the technical reason for the repeats is that `urnsamples()` operates on the $1, 2, \ldots, 14$ indices of `x` by drawing unordered samples, without replacement, of `1:14`. It then takes those samples and substitutes back to generate the entries of the data frame. In the case where `x` has repeated values, the result is many repeated rows in the sample space `P`.

So what do we do? What if we really want to see the hypergeometric model, and we really only care about the number of `Red`s in our sample? Well, the good news is that while there are many repeated rows, it turns out that there are *exactly* the right number of rows repeated *exactly* the right number of times. We simply need to accumulate them to get the desired answer.

The `noorder()` function was designed for this case. The way it works is to 1) sort the outcomes of each row, 2) aggregate these new rows to eliminate duplications, and 3) accumulate the probabilities of each of the duplicated rows. Its only input is a probability space, and its return value is also a probability space. Let's `noorder()` the probability space `Q` and see what happens:

```
> noorder(Q)
```

```
    X1   X2   X3      probs
1 Blue Blue Blue 0.32967033
2 Blue Blue  Red 0.49450549
3 Blue  Red  Red 0.16483516
4  Red  Red  Red 0.01098901
```

Now, this is more like it. There are four outcomes, each with a differing number of `Red`s in the sample, namely, either 0, 1, 2, or 3 `Red`s. The probabilities associated with each outcome perhaps look familiar, and we can confirm the connection between these and the hypergeometric model with the `dhyper()` function:

```
> dhyper(0:3, 4, 10, 3)
```

```
[1] 0.32967033 0.49450549 0.16483516 0.01098901
```

Indeed, the matches are no coincidence. This example drives home the point that in every case, one must think carefully about the experiment, look carefully at the results during every step of the process, and apply the tools at hand to ultimately solve the problem.

Keep in mind that we could have arrived at the same answer by simply defining the random variable $X =$ number of Reds and finding the `marginal()` distribution of X.

9.4 Some Things That This Package Does Not Do

Some users will notice that some topics were omitted from the `prob` package. The following is a partial list of some of the more important missing topics.

- **Expectation:** the concept of expectation is nowhere to be found in the `prob` package. There are two reasons for this. First, once we have the probability space it is simple to use column operations to find expected values and it is instructive for students to perform those tasks themselves. Second, there already exists a perfectly fine expectation operator `E()` in the `distr` family of packages; there is no need to duplicate that functionality here.

- **Infinite Sample Spaces:** the package as a whole assumes that the sample space is finite. This rules out common sample spaces discussed in many probability courses, for instance, the sample space associated with tossing a coin until Heads occurs. Unfortunately, it is impossible to store an infinitely long vector in the finite memory of a computer.

  There are of course tricks around difficulties like these, such as chopping off the sample space once the probabilities drop below a certain threshold. But the geometric case also poses the problem that the outcomes are of differing lengths, which does not line up nicely in data frame format. It was decided to forego addressing these issues at the present time.

- **Bayes' Rule:** there are no specific functions devoted to Bayes' Rule, and this is simply because they are not needed; we have the entire sample space at our disposal and can discuss partitions of it naturally without any extra help. The `prob` package was written to be largely neutral with respect to both the frequentist and subjective interpretations of probability, and it can be used under either paradigm. In fact, the only function even remotely frequentist is `empirical()`. The fact that `probs` is arbitrary leaves it to the user to determine an appropriate model.

If you know of topics that would be of general interest and could be incorporated in the `prob` package framework, I would be happy to hear about them. Comments and suggestions are always welcomed.