

Choosing random samples

How to get a representative sample from a large data set



In this issue...

▶ WHAT'S COVERED

Within many different fields of study there is often a need to select a sample set of records from a much larger set. Statistical methods are then used to analyse the sample in order to discover information about the larger set. It's a common problem that seems easy enough to implement, but in reality there are some traps for the unwary programmer.

Statistics is a method of gaining pertinent information from a study of a sample of the data rather than trying to analyse the whole set. From the sample, we hope to gain information and intelligence about the larger set as a whole without having to go through the pain and problems of a large-scale analysis.

The problem is that we need to get a representative sample. In other words, we need to select a set of items at random from the entire set.

Let's make the argument easier to understand by supposing that we have a total population of 10 records and we need to create a set of three records, chosen at random. How do we obtain this set without bias?

The first algorithm is a simple one. Since we need three items out of the 10, let us look at each item and select it with a probability of $3/10$, or 0.3. In other words, we generate a random number between 0 and 1 for each item, and if the number is 0.3 or less, we select the item to be in our sample. Overall, so the argument goes, we'll choose three items to

be in our sample. Unfortunately, it doesn't work. Although each item has a probability of 0.3 of being chosen, there's no guarantee that if we did this for 10 items we'd end up with three selections. On average, over many, many trials, we'd end up with three items. However, for a single experiment, there's no guarantee.

This is exactly the same issue when tossing coins. Sure, the probability of getting heads on a single toss is 0.5, but over 10 tosses you may get four, five or six heads quite easily. Sometimes, you might get counts that are even further out, like two or nine. However, we want an algorithm that will give a guaranteed three items in our sample, no matter what. Not two or four, but three.

Obviously, the only way we can use the simple algorithm is to try and try again until we get a three-item sample. According to probability, we'll get there in the end, but it may take many trials before we do.

So let's abandon that version and do something else, something that will only take one pass through the data set and guarantee that we only select

Reservoir sampling

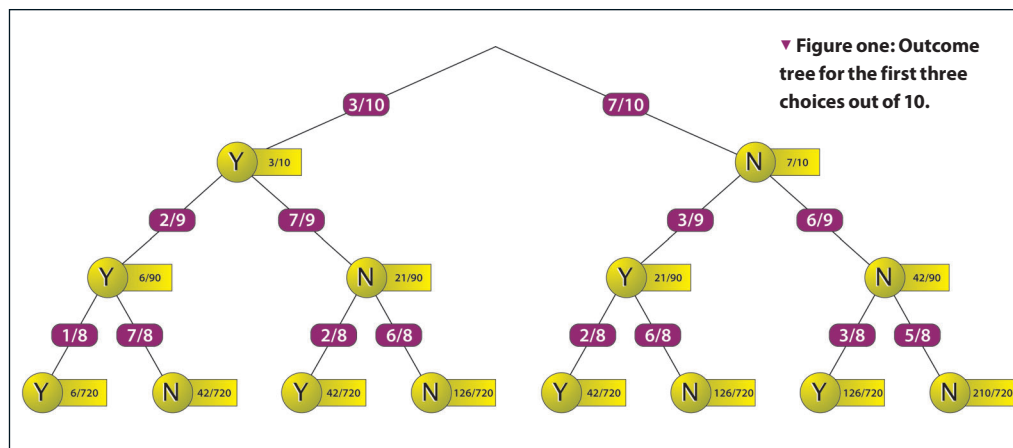
Suppose you can split up the population of items among several machines. Maybe each PC already has a part, such as web servers and web logs. How do you create a sample in this case? Part of the solution is to get each machine to create its own reservoir and then merge the reservoirs into a single sample. The issue is that each reservoir is 'weighted' by the number of items in its original population and so the random selection process is a little complicated – and patented. ■

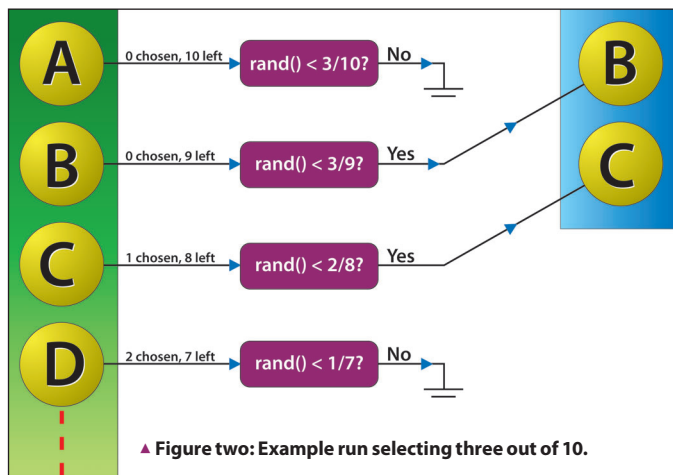
three items out of the 10, and that each item is selected with the same probability.

A better selection

Let's imagine we have two buckets in which to put the items. One bucket is for the selections for our sample and the other is for the rejects. We start off with 10 items and we need three items to go into our sample bucket. Take the first item. Since we need three items and there are 10 to choose from, generate a random number. If it's less than 0.3, put the item into our sample bucket. If not, toss it in with the rejects.

For the second item, the comparison changes a little bit. For a start, we now have one less item in our population. Secondly, we may already have selected an item (in which case we need two more items for our sample), or we may have rejected one (in which case we still need three items). For the first case, which occurs 0.3 of the time, we generate a random number between 0 and 1, and if it's less than $2/9$ we select it (the two because we need two more





items, the nine for the number of items in the current population). For the other case, which occurs 0.7 of the time, we use 3/9 as the comparison number.

Notice that, by using Bayesian probabilities, we can calculate the overall chance of selecting the second item in the population as $(3/10) \times (2/9) + (7/10) \times (3/9)$, which is 0.3, the probability we want. (The first term is the probability of choosing the second item given that we've selected the first, and the second term the probability of choosing the second item given that we rejected the first.)

The tree in Figure one shows all possible outcomes (and their probabilities) for the first three items. Each link to the left involves selecting an item (Y), and each link to the right, rejecting it (N). Using this you can calculate the total probability of selecting the third item (all the left links on the third level): $6/720 + 42/720 + 42/720 + 126/720$, which is 0.3.

In general, to decide whether to select the nth item, we calculate a random number. If it's less than the number of items still to get for our sample divided by the number of items remaining in the

population, we select it. The fun part of this technique is that once our sample bucket is full the numerator of the division is zero and we'll never select another item. If our sample is still empty once we get towards the end of the population, the number of remaining items in the population will be exactly the sample size and we'll select every remaining item. Either way, we can easily cut short the random number generation process.

Doing it in one pass

Figure two shows the start of an example run to select a sample of three out of a population of 10. The nice thing about this algorithm (devised in 1962 by Fan, Muller and Rezucha) is that we only make one pass through the population of items and select our sample as we go along. The not-so-nice thing about this algorithm is that we need to know the number of items in the sample beforehand in order to calculate the probabilities. Sometimes the only way of determining this value is by counting the items right there and then, which counts as another pass through the

population. So is there a way of modifying this algorithm so that we don't need to know the initial number of items while still only needing one pass?

As it happens, there is: reservoir sampling. According to Don Knuth, this algorithm was first devised by Alan C Waterman, although he doesn't give a citation or date.

Reservoir sampling

The algorithm makes use of a bucket that can hold the sample. This bucket is known as the 'reservoir'. Let's again assume that we need a sample of three items. We fill the reservoir with the first three items from our population. Obviously, if the population has already run out of items, the sample is complete and we're done. Generally though, there are lots more items in the population. We now go through the remaining set. For each item, we'll apply a random selection technique and if the item is selected, we'll just select one of the items at random from our reservoir, throw it out and replace it with this new item. Two questions remain: how do we decide whether to select a new item for our reservoir and how do we know that each item in the reservoir has been selected with equal probability?

Suppose we've just started this process, so we're at the fourth item in the set. We want to select this item in such a way that at the end of perusing the fourth item, the three items in the reservoir have an equal probability of being there. In this case, after we decide whether to choose the fourth item or not, the probability of each item in the reservoir being there should be 3/4.

For the fourth item, this is easy: we generate a random

Weighted sampling

Another variant of reservoir sampling is weighted sampling. Here, each item is assumed to have a weight attached to it that alters the probability of that item being selected into the sample; a higher weight implies a greater probability of selection. The looping part of the reservoir algorithm remains the same, but the probability part must include the weights. In essence, we need to maintain the total weight of the reservoir and the total weight of the items seen so far, and factor these into the probability. ■

number between 0 and 1, and if it's less than 3/4 we select it and replace one of the other items in the reservoir. Hence it has a probability of 3/4 of being in the sample. However, what about the items in the reservoir? Take item one as an example. The chance of it being rejected is $3/4 \times 1/3$, or 1/4 (that is, a probability of 3/4 that the 4th item will be selected times the probability of 1/3 that item one will be selected for removal). The same argument applies to the other items as well. At the end the decision of whether to choose the fourth item or not, the items in the reservoir each have an equal probability of being there.

To generalise, suppose we are at the nth item in the population. We generate a random number between 0 and 1, and if it's less than $3/n$ we select it. If so, we randomly select an item in the sample to discard and replace it with this new item. We can use exactly the same argument as above for this nth step to show that the items in the reservoir have an equal probability of being there, whether we choose the nth item or not.

Once we've processed all of the items in the population, we know that the items in the reservoir were selected randomly and that each has exactly the same probability of being present. ■

Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express. feedback@pcplus.co.uk

Helpdesk

Our experts help you with your PC problems and suggest some useful tips, **page 88**

Spotlight on... Simple random sampling

Another simple sampling algorithm for the case where we can hold the population in memory is to generate and store a random number between 0 and 1 for each of the items in our set. We then sort the items in random number order and take the top (or bottom) n items from the sorted

list (where n is the sample size). For instance, if your items were all rows in a spreadsheet, all you'd need to do is to add another column to the data, populate it with random numbers using the 'rand()' function and then sort the whole data set by that column. The problem with this method is

that you'll be making two passes through the data: one pass will allocate a random number to each item and the second will sort the data in random number order. There are faster algorithms that will select the top n of the items without sorting, but they still require that extra pass. ■