Tangentially / A Machine Learning Blog

WEIGHTED RANDOM SAMPLING WITH REPLACEMENT WITH DYNAMIC WEIGHTS

FEBRUARY 14, 2016 | AARON DEFAZIO | LEAVE A COMMENT

Weighted random sampling from a set is a common problem in applications, and in general library support for it is good when you can fix the weights in advance. In applications it is more common to want to change the weight of each instance right after you sample it though. This seemingly simple operation doesn't seem to be supported in any of the random number libraries I've looked at.

If you try googling for a solution, you find lots of papers and stack overflow posts on reservoir sampling, but nothing useful for solving the problem. After digging through Knuth and reading old paywalled papers, I've managed to piece together an approach that is extremely fast and easy to implement. This post details that method and provides a simple Python implementation. I have made a fast Cython version availiable on github also.

First some notation. We want to sample an index 0 to N-1, according to an array of weights w[i]. These weights form the unnormalized probability distribution we want to sample from, i.e. each instance i should have probability w[i]/sum(w) of being chosen. Ideally we want an algorithm that gives constant time sampling and constant time weight mutation operations.

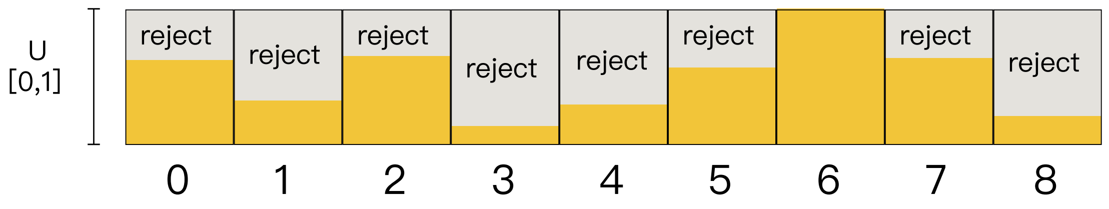
The simple slow approach: rejection sampling

Normally I avoid wasting time on approaches that don't work well in practice, however the simple rejection sampling approach to the problem turns out to be the vital building block of the algorithms that do work.
The rejection sampling approach is only a few lines of Python:

```
w = [1,4,2,5] # Some data
w_max = max(w)
n = len(w)

while True:
        idx = random.randrange(n)
        u = w_max*random.random()
        if u <= w[idx]:
                break
print idx
```

rejection.py hosted with ❤ by GitHub view raw

The idea is simple. We sample uniformly from the indices, then do a rejection sampling correction to account for the actual non-uniformity of the data. The best way to visualise what it's doing is to consider it as picking a point in 2 dimensions uniformly, then doing a reject or accept operation based on if the point is under the "graph" of the distribution or not. This is shown schematically as the yellow shaded regions below.



The general idea of rejection sampling and this graph interpretation is quite subtle, and I'm not going to attempt to explain it here. If you just stare at it for a while you should be able to convince your self that it works.

Unfortunately, rejection sampling like this is not practical when the weights are uneven. The re - jection probability depends on the magnitude of the largest weight compared to the average, and that can be very large. Imagine if the first eight boxes above where %1 full and the last 100% full. It's going to reject roughly 80% of the time. It can of course be much worse when n is larger.

Making it practical

Rejection sampling can be very fast when all the weights are similar. For instance, suppose all the weights are in the interval [1,2]. Then the acceptance probability w[i]/w_max is always more than half, and the expected number of loops until acceptance is at most 2. It's not only expected con - stant time, but it's fast in practice as well. More generally, this is true whenever the interval is of the form [2^i, 2^(i+1)].
This leads to the idea used by most of the practical methods: group the data into "levels" where each level is an interval of that form. We can then sample a level, followed by sampling within the level with rejection sampling. Matis et al. (2003) show that the level sampling can be done in O(log*(n)) time, where log* is the iterated logarithm, a slowly growing function which is always no more than 5. Effectively it is an expected constant time sampling scheme.

We don't recommend using the Matis scheme though, as its practical performance is hampered by its complexity. Instead, we suggest using a method that has a (weak) dependence on the size of the weights, which we detail below.

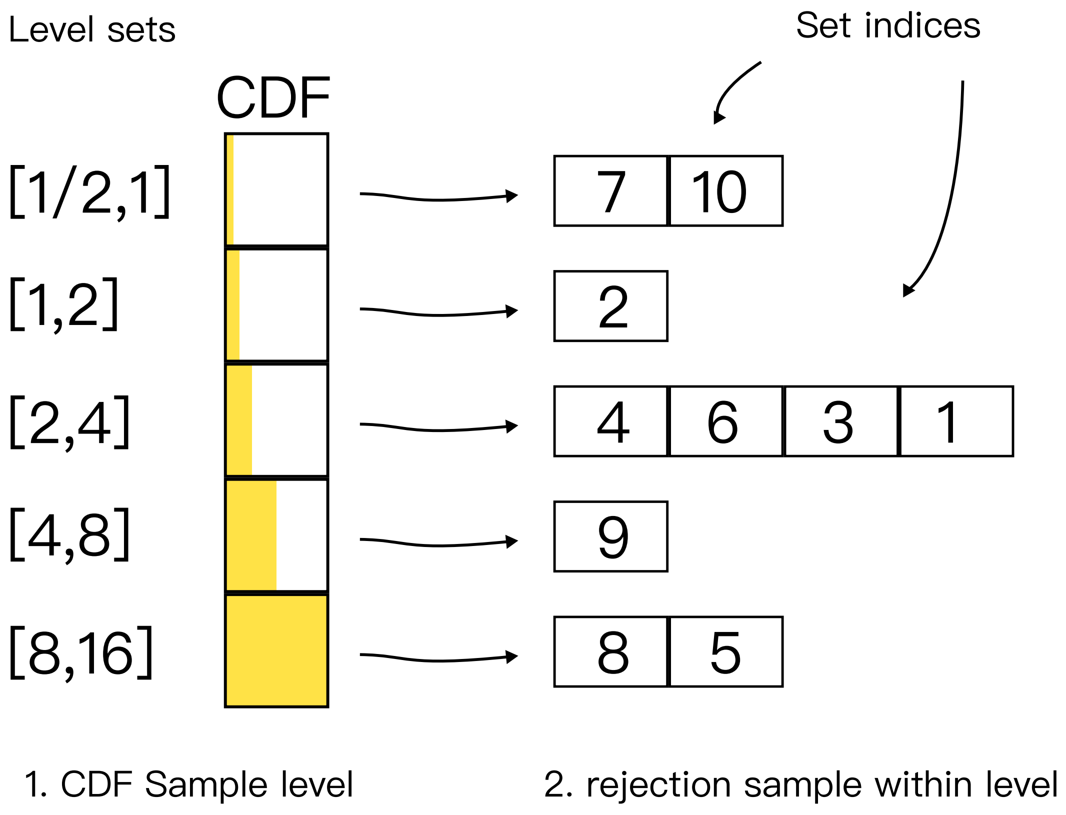
Consider the intervals [2^i, 2^(i+1)]. The number of unique intervals ("levels") we need to consider is just the log2 of the ratio of the largest and smallest weights, which on practical problems won't be more than 20, corresponding to a 1 to 1 million difference. The extra overhead of the Matis method is not worth it to reduce the constant from 20 to 5. In practice the Matis method requires building a tree structure and updating it whenever the weights change, which is way slower than traversing a 20 element sequential array.

Lets be a little more concrete. The algorithm will maintain a list of levels, in order of largest to smallest. Each level i consists of a list of the elements that fall within that levels range: [2^i, 2^(i+1)]. The list for each level need not be sorted or otherwise ordered. To sample an instance from the set, we sample a level, then we perform rejection sampling within that level. In Python, it looks like:

```
    def sample(self):

        u = random.uniform(high=self.total_weight)

        # Sample a level using the CDF method
        cumulative_weight = 0
        for i in range(self.nlevels):
            cumulative_weight += self.level_weights[i]
            level = i
            if u < cumulative_weight:
                break

        # Now sample within the level using rejection sampling
        level_size = len(self.level_buckets[level])
        level_max =  self.level_max[level]
        while True:
            idx_in_level = random.randint(0, level_size)
            idx = self.level_buckets[level][idx_in_level]
            idx_weight = self.weights[idx]
            u_lvl = random.uniform(high=level_max)
            if u_lvl <= idx_weight:
                break

        return idx
```

sampler_sample.py hosted with ❤ by GitHub   view raw

The full class including weight updating is on github. Notice that we use a linear-time algorithm for sampling the levels (A cumulative distribution table lookup). Alternative methods could be used, such as a balanced binary tree or Walker's algorithm (see below). The only difficulty is that we want to change the weight of an element potentially after every sample, so any pre-computa - tion needed for the level sampling needs to be fast. It doesn't seem worth it in practice to use something more complicated here since the number of levels is so small (as mentioned above usually less than 20).



Enhancements

A few small changes are possible to improve the usability and performance. The rejection sam - pling actually only needs a single random sample instead of 2. We can just take a U[0,1] sample, then multiply by level_size. The integer part is the idx_in_level and the remainder is the u_lvl part.
When updating weights, we need to delete elements potentially from the middle of a levels index list. For example, imagine we need to move element 6 from the [2,4] bucket to the [1,2] bucket in the above diagram. We need to store the indices in a contiguous array for fast O(1) lookup, so initially this would look like a problem. However, since the order within the lists doesn't matter, we can actually take the *last* element of the list, move it to the location we want to delete from, then delete from the end of the list.

In the sample code given, we keep a level_max array which just contains the upper bounds for each level. With a little extra code we could instead change this to be the largest element that has been in that bucket so far. This could lower the rejection rate a little at the cost of a few more operations maintaining the level_max array.

Other methods

Marsaglia et al. 2004 describe an early method that also treats the data in levels, but instead of rejection sampling it spreads each datapoint's probability mass across multiple levels. I believe upating the weights under their scheme would be quite complex, although in big O notation it should be roughly the same as the algorithm I described above.

Walker's alias method

This post is concerned with methods can be used when we wish to modify the weights dynamically as the algorithm runs. However, I would be remise to not mention the alias method, which given a fixed set of weights at initialization time constructs a table that allows extremely efficient constant time sampling without any of the complexity of the other approaches I mentioned.
The key idea is simple. Take a table like that used in the rejection sampling method. Instead of normalizing each bucket by w_max, normalize by w(sum)/n. When doing this, some buckets will overflow, as the probability within them could be larger than w(sum)/n. So we take the excess, and spread it amoung those buckets that are not full. It's not too difficult to do this in such a way that each bucket is exactly full, and contains only two indices within it. Now when we go to sample, we pick a bucket uniformly as before, but within the bucket there is no longer a rejection option, just two indices to choose from.

Unfortunately, there is no fast way to modify the table built by Walker's method when we change just a single weight. I've not been able to find any methods in the literature that manage to do so efficently. Given it's simplicity, there are a few implementations of Walker's method out there, for example a Python version here.