Subject: Re: majority voting
Posted by JD Smith on Thu, 12 Feb 2009 18:49:02 GMT
View Forum Message <> Reply to Message

On Feb 12, 10:45 am, Mort Canty <m.ca...@fz-juelich.de> wrote:
> Allan Whiteford schrieb:
>
>
>
>> mort canty wrote:
>>> Hi all,
>
>>> Given a 2-D array such as
>
>>> 0 1 1 2 1
>>> 0 2 1 1 1
>>> 1 0 2 2 1
>
>>> where the entries are labels, the columns represent items and the rows
>>> are voters, I want a IDL function that returns the majority vote
>>> labels. So here I should get
>
>>> 0 ? 1 2 1
>
>>> as output, where ? = "don't care". There must _not_ be a loop over
>>> columns. I've got a clumsy solution, but I'm sure there's an elegant
>>> one somewhere?
>
>>> Cheers,
>
>>> Mort
>
>> Hi Mort,
>
>> It might be less efficient than JD's histogram solution (I didn't check)
>> but the following also fits the problem specification:
>
>> x=[[0,1,1,2,1],$
>> [0,2,1,1,1],$
>> [1,0,2,2,1]]
>
>> voters=(size(x,/dim))[1]
>> items=(size(x,/dim))[0]
>> max_label=max(x)+1
>
>> f=intarr(max_label,items)
>> ++f[max_label*(indgen(voters*items) / voters)+ $
>> reform(transpose(x),voters*items)]

>> junk=max(f,idx,dim=1)
>> print,idx - max_label*findgen(items)
>
>> Note that the above solution will also blow up when you end up with
>> sparse arrays (e.g. if you have someone voting for label 1000000 then f
>> will end up being an items x 1000000 array even if nobody votes for any
>> labels between 3 and 1000000).
>
>> I think all the discussions on finding the mode (either in 1D or nD)
>> probably pre-dated the ++ operator. It could be that using the
>> vectorised ++ operator is a better way to do it - I doubt it though,
>> normally if histogram can do something then histogram will be the best
>> way! You'd also need to introduce a clumsy offset to deal with negative
>> selections (Not an issue for you here but would be if finding the mode
>> in a more general way).
>
>> It would make David's 1D example from his webpage into something like this:
>
>> array = [1, 1, 2 , 4, 1, 3, 3, 2, 4, 5, 3, 2, 2, 1, 2, 6,-3]
>> f=intarr(max(array)-min(array)+1)
>> f[array-min(array)]++
>> junk=max(f,idx)
>> mode=idx + min(array)
>> print,mode
>
>> again, with no idea on what would be more efficient. If you're doing
>> analysis on measurements (typically non-integers) then you'd need to
>> invoke histogram anyway to bin them before trying to find the mode.
>
>> Thanks,
>
>> Allan
>
> Hi Allan,
>
> Thanks! Not only have I learned that something called vectorized ++
> exists, but that it bumps the indexed value multiple times if that index
> is repeated. Live and learn! But where the hell is all that on the IDL Help?
>
> Anyway, what I had in mind was trying to program an ensemble image
> classifier, so that the items are rows of pixels (lots and lots), the
> labels are land cover classes (contiguous small integers) and the voters
> are classifiers (e.g. neural networks, also not too many). Hence the
> wish to avoid the loop over items. I certainly got my money's worth :-)

This was big news to me as well, since normally this does *not* work
like this. Example:

```
IDL> x=[0,0]
IDL> x[[1,1]]+=1
IDL> print,x
       0       1
```

This limitation is even called out in HISTOGRAM's help. Now let's try the new operators:

```
IDL> x=[0,0]
IDL> x[[1,1]]++
IDL> print,x
       0       2
```

I'm not sure if this is intentionally or accidentally inconsistent, but it is indeed very useful. Allan's solution using this is actually quite powerful. It easily bested the speed of my HIST_ND version by a factor of ~5. Not surprising, given that HIST_ND is optimized for floating point data, and spends lots of unnecessary (for this problem) time scaling the (already integer) item number.

I was interested to see how a bare HISTOGRAM would perform on straight integer data compared to the "sensibly" vectorized '++' operator. I sped up Allan's solution somewhat by avoiding the TRANSPOSE:

```
  s=size(x,/DIMENSIONS)
  f=lonarr(issues,items)
  ++f[x+issues*rebin(lindgen(s[0]),s,/SAMPLE)]
  junk=max(f,vote,dim=1)
  vote mod= issues
```

where 'issues' is the range of the possible "vote" (3 in the original example). I simplified my example to use only HISTOGRAM (for now as in Allan's case forgetting about "no preference" or tie votes):

```
  s=size(x,/DIMENSIONS)
  h=reform(histogram(x+issues*rebin(lindgen(s[0]),s,/SAMPLE), $
                MIN=0,MAX=issues*s[0]-1L),issues,items,/
OVERWRITE)
  m=max(h,DIMENSION=1,mode)
  mode mod= issues
```

The results were almost identical in all cases of input array size and data range.

This result is not at all specific to this problem. Here's an even rawer example, doing a very simple integer histogram of a vector:

```
 n=100000L
```

```
cnt=3
x=long(randomu(sd,n)*cnt)
t=systime(1)
h1=histogram(x,MIN=0,MAX=cnt-1L)
print,'H1:', systime(1)-t

t=systime(1)
h2=lonarr(cnt)
++h2[x]
print,'H2:',systime(1)-t
```

```
H1:   0.00035190582
H2:   0.00035905838
```

Almost identical! Changing the number and max cnt puts one over the other by a few % at most, consistent with no difference whatsoever. In some ways, it's not entirely surprising, given that both operations are doing the exact same thing.

So the bottom line is that '++f[inds] ' and 'f=histogram(inds)' for integer 'inds' are essentially identical in results and performance. For floating point data, you can simply pre-scale to integer (which is what HIST_ND actually does). Obviously, you don't get reverse indices, but otherwise you could quite literally rewrite HIST_ND without once using HISTOGRAM! Even if it's not any faster, the ++ syntax may be clearer for certain problems.

Good find.

JD
