Needles in Haystacks
Finding Unusual Data Values

James G. Wheeler
SmartArrays, Inc.

Much of data analysis is aimed at finding out typical values – averages, medians, and the like. But sometimes it's just as important to sift through large collections of data for values that are *atypical***.** Unusual values often mean something interesting. They may indicate an error in the way data was collected (i.e. a keyboarding mistake or a sensor glitch). In manufacturing, they may indicate a quality problem. In the realms of security and intelligence, an unusual data value may indicate something that deserves a closer look.

Let's imagine that you have been given a log of network traffic for computers accessing some secure resource like a database. The log includes the IP addresses of the requesters, and most of these are from a small group of computers that normally access the service and access it frequently. You have been asked to identify any oddball IP addresses, i.e. requests from outside the "normal" set.

Suppose the log file records 10 million requests. How can you quickly find the ones that don't belong? A frequency table – a list of unique values and the number of times each occurs -- would answer the question. So, what's the best to produce it? I suspect many people would take one of two approaches:

- Load the data into a database and then produce the counts of unique values using SELECT COUNT with GROUP BY.
- Write a program that sifts out the unique IP addresses in the file and then counts the occurrences of each.

The database option will certainly work if the data is already in a database. But the effort to load the data into the database may be considerable, and no database will import 10 million new rows very rapidly. Once it's there the query will run slowly unless you also invest the time to create an index on the column.

The write-it-yourself approach will also work, if you can afford the time to develop a program and make sure it works correctly. But this can take quite a bit of time if you start from scratch, especially if you want to avoid writing an N-squared algorithm that takes forever to execute.

Using SmartArrays can make the write-it-yourself approach much more attractive. You can code quickly, be confident that the answer will be correct, and your solution will be very fast to run. Here's how:

Let's assume that the file has already been read and that the IP addresses have been extracted from it. (If you want to see a SmartArrays technique for this, I'll show it at the end). Assume that set of ten million IP addresses is now stored as an SmartArrays vector of 10,000,000 strings:

```
        SmArray ipaddrs = read_IP_Addrs();
        ipaddrs.showDebug();
S[10000000] "201.123.43.123" "201.123.43.156" "134.32.36.43" "201.123.43.123"…
```

To get the frequencies, we first want to find the set of unique strings in the set:

```
        SmArray unique_addrs = ipaddrs.unique();
```

This gives us a short vector of the distinct strings that are found in **ipaddrs**. Now we will use **lookup()** to find map each item in **ipaddrs** to the set of unique strings:

```
        SmArray inds = unique_strings.lookup(strings);
        inds.showDebug();
I[10000000] 0 1 2 0 0 1 1 3 2 2 2 1 0 0 1 3 4 1 2 3 0 1 1 3 2 2 2 1 0 0 1 3 4 …
```

In other words, the first of the ten million matches the string at position 0 in the unique list. The second matches the string at position 1, and so on. To get the number of times each string occurs, we need to count the number of times 0 occurs, the number of times 1 occurs, etc. One very fast way to do this with SmartArrays is to use the aggregation operation **selectUpdateBySubscript()**. Here's the code:

```
        SmArray counts = unique_strings.shape().reshape(0);
        counts.selectUpdateBySubscript( Sm.plus, SmArray.scalar(1), inds );
```

The first line creates a vector with one zero for each unique string. Then we add 1 to each position for each time its subscript occurs in **inds**. **selectUpdateBySubscript()** takes three arguments:

- The identifier for the scalar method to apply, **Sm.plus** in this case to aggregate the values additively.
- The values to be aggregated. Normally there is one value for each subscript in the third argument, but SmartArrays allows "scalar extension" here – so the 1 is used as the value to add for every subscript.
- The positions in the target array (**counts**) where the values are to be aggregated. The result of **lookup()** gives us one position for each of the ten million IP addresses.

So, looking at where the values of **inds** were displayed above, we can see that **selectUpdateBySubscript()** will add 1 to **counts** at position 0, at position 1, at position 2, at position 0 again, and so forth. When it's done, we will have the count of occurrences of each unique string.

And you don't have to wait very long either. On my laptop computer, the whole process takes *less than 3 seconds to execute.*

Here's a standalone function for producing the set of unique values and the number of times they occur, starting with the most frequent. Note that the input data does not have to be strings. The same function works just as well with arrays containing integers or other kinds of values.

```
    public SmArray frequency( SmArray v )
    {
        SmArray unique_v = v.unique();
        SmArray counts = unique_v.shape().reshape(0);
        counts.selectUpdateBySubscript( Sm.plus, 1, unique_v.lookup(v) );
        return unique_v.colCat(counts).row( counts.gradeDown() );
    }
```

Give it any vector and it will return a 2-column matrix with the unique values and the number of times they occur in descending order of frequency. Let's run frequency() on our sample data set and see if there are any unusual IP addresses in the set:

```
    frequency( ipaddrs ).show();
134.32.36.53   2656878
134.32.36.45   2598302
201.123.43.156 1759606
134.32.36.43   1717301
201.123.43.123 1261538
102.100.99.99  2
```

Sure enough, there's one IP address that only occurs twice, which may be worth investigating further. This is a good example of how using SmartArrays can help you quickly and easily build applications that are fast at crunching through large data sets.

Appendix: Efficiently Reading and Parsing a Flat File

In this example, we assume that we receive the data in a text file such as might be produced by a program's logging function. Let's assume that the records are newline delimited and that the first field is the IP address, followed by a space, followed by other logging information that we don't care about, like these:

```
Line 1:  "201.123.32.146 port=20 time=2004.10.21.22.04.123 …"
Line 2:  "132.32.36.53 port=14807 time=2004.10.21.22.06.410 …"
Line 3:  "132.32.36.45 port=14807 time=2004.10.21.22.09.809 …"
```

Suppose also that there are 10 million of these records, so the text file will be large – on the order of half a Gigabyte or more.

Before we can apply SmartArrays techniques to the IP addresses, we must first extract them from the file and store them in an array. The SmartArrays SDK offers a number of ways to fill arrays with data, some of which are much more efficient than others.

An important consideration to fast processing with SmartArrays is to try to do a substantial chunk of work in each call to a SmartArrays method, since there is a measure of overhead associated with performing an array call. For example, the following technique can take a long time to build up an array of 10 million strings:

```
SmArray ipaddrs = SmArray.emptyString();
while ( more to process )
{
    string s = read and process next line;
    ipaddrs.append( SmArray.scalar(s) );

}
```

If there are 10 million strings, this will take 10 million calls to SmArray.scalar() to create a scalar array and another 10 million calls to **append()** to add them to the vector **ipaddrs**. The overhead of 20 million SmArray calls really adds up and a loop like this would take a long time to run.

Fortunately, there is another SmArray initializer that's more appropriate: SmArray.vector(string[]), which will build an SmArray vector from a native string array. This lets us use a buffering approach to building the array. Here's a function in C# that does that using a .NET StreamReader to read the flat file.

```
public SmArray readIPAddrs( string file, int bufsize )
{
    // start with an empty vector of strings
    SmArray ipaddrs = SmArray.emptyString();

    // Use a StreamReader to read the file one line at a time.
    System.IO.StreamReader r = new System.IO.StreamReader( file );

    // Create a buffer of C# strings to hold the strings.
    // We will read each line from file, strip out the IP
    // address from the rest of the line, and hold it in
    // the buffer.  When the buffer is full, we append the
    // strings to the SmArray ipaddrs.
    string s;
    int ibuf = 0;
    string[] sbuf = new string[bufsize];

    while ( (s=r.ReadLine()) != null )
    {
        if ( ibuf == bufsize )
        {   // buffer is full - make into an SmArray vector
            // and append it to the string list
            SmArray chunk = SmArray.vector( sbuf );

            // using append()is more efficient than catenate
            // because it will write into extra space at the end
            // of the array.  Passing the current size of ipaddrs
            // as the growth factor means that if the extra space
            // if used up, the array will be given enough extra
            // space to double in size before it must be expanded
            // again.
            ipaddrs.append( chunk, ipaddrs.getCount() );
            ibuf = 0;
        }

        // strip off first space and everything after that in line,
        // leaving just the IP address
        string ipaddr = s.Substring( 0, s.IndexOf( ' ' ) );
        sbuf[ibuf++] = ipaddr;
    }

    // add remaining strings in buffer
    if ( ibuf > 0 )
    {
        string[] temp = new string[ibuf];
        for ( int i=0; i<ibuf; i++ )
            temp[i] = sbuf[i];
        SmArray last_chunk = SmArray.vector( temp );
        ipaddrs.append( last_chunk );
    }
    return ipaddrs;
}
```

Then the file may be read by specifying an appropriate buffer size for the string buffer (100,000 works well in this case).

```
SmArray ipaddrs = readIPAddrs( filename, 100000 );
```

Once you have the strings in an SmArray, the fastest way to save them to file is as a serialized array using the **toFile** and **fromFile** array methods, which store an array to file in a form that can be turned back into an array. However, these are binary files whose format is understood only by SmartArrays, so they are useful for storing arrays for later use by SmartArrays.

Notes:

In the examples below, the output from the SmArray functions show() and showDebug() is shown below the statement in blue. This is the output that would normally appear in the debug window of your IDE or on the output stream of a console application.