Checking validity of input data

When analysing large amounts of data we are constantly confronted with a question: are input and intermediate data in a right form, and do they conform to processing programs' input requirements? Deviations from the correct input can be present already in the input data, or can be caused by mistakes in the processing pipeline. For sure such deviations can distort computation results and lead to incorrect conclusions.

Here are three simple methods how you can check, and with one method even validate, the inputs and outputs of your programs. The first three examples will be applicable to data in tabular form, the one that is most naturally obtained from GNU/Linux and Unix tools, and the one which we used in our experiments. The example of such file is given below:

```
saulius@varanas 3rd-assignment/ $ head -n 4 frequency.lst
    860 10.1038/ncomms15123
    364 10.1016/j.str.2016.06.010
    152 10.1016/j.tube.2014.12.003
     86 10.1021/jm200642w
```

Method 1. Count the columns

Count number of columns in each of your data files. The number of columns must be the same in all lines, except maybe the file header or comments (which you can easily filter out with 'grep'). The column number must also be suitable for the next processing program. Number of columns can be easily obtained by printing out the 'awk' NF ("Number of Fields") variable:

```
saulius@varanas 3rd-assignment/ $ awk '{print NF}' frequency.lst | uniq -c | head
  27618 2
 137709 4
  30364 2
```

As you see, in this case some lines (in fact, a lot – 137709 of them!) have four columns, which is already suspicious. You can print out such columns with 'awk':

```
saulius@varanas 3rd-assignment/ $ awk '{if( NF > 2 ) print NR, $0}' frequency.lst | head -2
27619       1 ==> ./outputs/downloads/pdb/9x/9xim.biblst <==
27620       1 ==> ./outputs/downloads/pdb/9x/9xia.biblst <==
```

The NR variable holds the current line number, so you also obtain the position of the "strange" lines in the file.

As a historical digression we can note that the method reminds the one used by Masoretes to count letters in each line of hand-written manuscript copies (https://en.wikipedia.org/wiki/Masoretic_Text#Numerical_Masorah), so that various copying mistakes can be detected.

I use this method every time I get data tables, can can highly recommend it to you.

Method 2. Check the random sub-sample of your data

Inspecting head and tail of your (large) data tables is a good habit, but what of faulty lines are in the middle? In this case, inspecting a random sample of your data lines may help, especially of the faulty lines are not to rare. The 'shuf' GNU tool will give you such sample:

```
saulius@varanas 3rd-assignment/ $ shuf frequency.lst | head -n 4
      1 ==> ./outputs/downloads/pdb/4h/4hr4.biblst <==
      1 10.1016/J.STR.2005.07.025
      1 ==> ./outputs/downloads/pdb/2f/2fsg.biblst <==
      2 10.1110/PS.03518104
```

As you see, the problematic lines show up immediately. Of course you will not find them that easily of there is just one or two such lines among hundreds of thousands. To increase the probability of detecting faulty lines, you can run the 'shuf …' pipeline several times.

Method 3. Check lines using regular expressions.

Since your data must follow certain syntax, it is nearly always possible to write a regular expression that matches *correct* lines. Then, you can invert you selection to get incorrectly formatted lines, and you can select the correct lines for further processing using 'grep'. Perl Compatible Regular Expressions (PCRE) are worth considering because of their power and ease of use. They are supported by 'perl' and GNU 'grep -P' commands:

```
saulius@varanas 3-homework-assignment/ $ head -2 frequency.lst
    860 10.1038/ncomms15123
    364 10.1016/j.str.2016.06.010
saulius@varanas 3-homework-assignment/ $ grep -P '^\s*[0-9]+\s+10\.[0-9]+/' frequency.lst |
    860 10.1038/ncomms15123
    364 10.1016/j.str.2016.06.010
saulius@varanas 3-homework-assignment/ $ grep -v -P '^\s*[0-9]+\s+10\.[0-9]+/' frequency.ls
      1 ==> ./outputs/downloads/pdb/9x/9xim.biblst <==
      1 ==> ./outputs/downloads/pdb/9x/9xia.biblst <==
saulius@varanas 3-homework-assignment/ $ grep -v -P '^\s*[0-9]+\s+10\.[0-9]+/' frequency.ls
      1 ==> ./outputs/downloads/pdb/10/100d.biblst <==
      1 10.1126
saulius@varanas 3-homework-assignment/ $ grep -v -P '^\s*[0-9]+\s+10\.[0-9]+/' frequency.ls
```

```
      1 10.1126
```

With this check, we see that there is a line with the wrongly formatted DOI, present just once on all text. Thus, regular expressions, although they take some time and ingenuity to compose, allow you to make a very thorough filtering of your data.

We can use 'find' to figure out where does the misformatted DOI line come from:

```
saulius@varanas 01-darbas/ $ find ~/GNU-type-OS/data/rsync-demo/saulius-grazulis.lt/outputs/
/home/saulius/GNU-type-OS/data/rsync-demo/saulius-grazulis.lt/outputs/downloads/pdb/1u/1u04.
```

Fetching data from the PDB shows that the misformatted data item come from the PDB:

```
saulius@varanas 01-darbas/ $ curl -sSL https://www.pdb.org/pdb/files/1u04.cif | grep _DOI
_citation.pdbx_database_id_DOI      10.1126
```

Method 3 expanded: check data against schema

Regular expressions are just the simplest for of grammars that allow you to check whether your data conform to some specific syntax. They work for any tabular form data; for example, FASTA or PDB files can also be validated using regexps. For more structures formats, like XML, CIF or JSON, more elaborate checks exists:

a) for XML files, you can use XML schema to check data:

   saulius@varanas 01-darbas/ $ curl -sSL https://www.pdb.org/pdb/files/1u04.xml > 1u04.xml

   saulius@varanas 01-darbas/ $ grep schemaLocation 1u04.xml xsi:schemaLocation="http://pdbml.pdb.org
   v50.xsd pdbx-v50.xsd">

   saulius@varanas 01-darbas/ $ xmllint –schema http://pdbml.pdb.org/schema/pdbx-v50.xsd –noout 1u04.xml 1u04.xml validates

Note that although XML schema allow to specify regexps to check structure of values (such as DOI) in .xml files, the current XML schema fails to discover the faulty DOI.

b) for CIF files (crystallographic interchange files), validation against CIF Dictionaries has the same function as 'xmllint' validation against XML schema; see our on-going work in cod-tools (https://github.com/cod-developers/cod-tools)

c) for JSON files, a similar schema are being developed (http://json-schema.org/); validating can be performed using Perl JSON::Validator module. On Ubuntu and LinuxMint, install libjson-validator-perl package ('apt install libjson-validator-perl'), and then use a Perl wrapper (our one can be fetched here: svn://saulius-grazulis.lt/scripts/json-validator, fetch with, e.g., ‘svn co svn://saulius-grazulis.lt/scripts’ or ‘svn cat svn://saulius-grazulis.lt/scripts/json-validator’).