barf: a drop-in bioinformatics file format validator

Camille Scott

March 18, 2016

Abstract

The growth of high-throughput DNA sequencing has necessitated that biologists embrace computing as an integral part of their work, and resulted in software being an import tool for basic biology. Studies often rely on complex pipelines to transform and filter these data, which is frequently passed through numerous bioinformatics applications. With so much software involved, the odds of faulty data being introduced are high; worse, that software is often written and maintained by academic labs with limited resources for software engineering or verification and validation. While biological meaning is difficult to verify and highly application-specific, verifying data integrity is tractable and understood. The resources to formally verify so much scientific software do not exist; however, if simple tools existed for checking integrity, it is likely researchers would use them. To that end, we introduce *barf*, a prototype for a drop-in bioinformatics validator which operates via UNIX pipes: upon encountering erroneous data, it "barfs", stopping the pipeline and alerting the user.

1 Introduction

To motivate this approach, let us begin with a couple real-life scenarios the author has encountered. We start with "L", a new graduate student with a background in bench biology who has been diving deeper into bioinformatics as a part of her PhD research. "L" is assembling a genome, and her analysis pipeline includes the widely-used program Trimmomatic [1] to remove low-quality sequences. Some days later, when the pipeline has completed, she starts to look more closely at her results, and realizes that one of the sequence files output by Trimmomatic is truncated: the FASTQ formatted file ends part-way through a DNA sequence, and includes no quality score. This does not trigger a failure until a few steps down the pipeline, when another program mysteriously crashes. As it turns out, Trimmomatic occasionally fails due to some unpredictable error which cannot be reproduced, and instead of returning an error code, returns 0 and truncates its output. Had the program behaved more appropriately, "L" would have identified the problem early-on and saved significant time.

Next up, we consider a hypothetical researcher using the software khmer [2]. khmer provides a utility for filtering sequences based on their redundancy in a de Bruijn graph, and this utility is now commonly used as a preprocessing step prior to genome assembly. Unfortunately, like most scientific software, it sometimes misbehaves: for example,

when it encounters an N in a sequence (the accepted way of representing an uncertain nucleotide), it simply replaces it with an A to avoid the performance hit associated with increasing the alphabet of its hash function from four ($\{A, C, G, T\}$) to five ($\{A, C, G, T, N\}$). While this is documented, it is not immediately clear to the user, and some downstream analyses, such as measurement of GC-content, could be affected.

Both of these cases might seem mundane, or even avoidable; "L" could report the bug to the developers, or even submit a patch, and the khmer team could make its sequence mutations more obvious by providing a warning message. The problem, of course, is that such solutions don't scale: soon enough, "L" is acting as a QA tester for two dozen different bioinformatics labs, and her research suffers. Further, there are countless bugs which undoubtedly go unnoticed, or are only intermittent, the so-called Heisenbugs [3]. Another solution is to simply expect that labs spend the time to formally verify their software; this would be ideal, but the land of academic software is rarely ideal, and prescriptive solutions require that the patient take their medicine.

Instead, we offer a more tractable solution: provide researchers with simple utilities for data verification, and make it easy to include them in their pipelines. A key part of this approach is to rely on the KISS principle: tools with limited scope are more likely to actually work. *barf* results from these goals, a serves as an example of what a more complete utility might look like. The current implementation operates on the FASTA format, a simple but ill-defined format which provides an appropriate test-bed. Originally developed for early aligner of the same name [4], it is now ubiquitous in the field. FASTA is a plain-text format which can be described by the following Backus-Naur grammar, provided by the bioboxes project:

```
<file>     ::= <token> | <token> <file>
<token>    ::= <ignore> | <seq>
<ignore>   ::= <whitespace> | <comment> <newline>
<seq>      ::= <header> <molecule> <newline>
<header>   ::= ">" <arbitrary text> <newline>
<molecule> ::= <mol-line> | <mol-line> <molecule>
<mol-line> ::= <nucl-line> | <prot-line>
<nucl-line>::= "^[ACGTURYKMSWBDHVNX-]+$"
<prot-line>::= "^[ABCDEFGHIKLMNOPQRSTUVWYZX*-]+$"
```

Plainly, a FASTA record starts a description line, which starts with a $>$ character and contains free text, followed by a line break, followed by a sequence line of either nucleotide or protein codes as specified by IUPAC. Many programs will hard-wrap the sequence line, which the above BNF grammar captures, but such behavior was not originally defined in the format. An entire ecosystem of alternative formulations exists, many specifying a format for the header line, but the standards are ad-hoc, leading to a multiple of practical issues with the format.

2 Approach

The main function of this tool is quite simple: intercept input streaming to a program via standard input; validate it and send it to the program; allow the program to operate

as normal; intercept the data again on standard out; and verify it before sending it to the disk (or the next stage in the pipeline). Correctly verifying the data is then key.

2.0.1 Limited Data Models

The approach for the FASTA format is to define a simple data model for a record. While we considered using a formal grammar early on, such as BNF or extended BNF, these methods turned out to be more verbose and unwieldy than was necessary: they are often aimed at parsing, rather than validating. Our utility should give some reason for why a parse fails, and in that manner, approaches the class of assertion languages. Instead, our approach is to assign data types to fields in the record of interest, and further describe properties of collections of those records.

The FASTA header is a string type, with the alphabet of all printable ASCII characters. The sequence is a string type with an alphabet limited to the IUPAC codes. These are well-defined via regex; in this view, the FASTA record is simple a tuple of two strings, which themselves are constrained by some parameters. We don't worry about parsing the format, per se, instead focusing on verifying that the data itself is valid. We rely on third-party parsers, which work well, but often do not constrain their data fields.

2.0.2 Scalable Collections

The above naive method would catch simple errors like proteins being mixed with DNA sequences: the user would simple specify that we validate the sequence field with the DNA grammar, and we would fail when encountering a protein. However, it does not catch the khmer error, and might not catch the Trimmomatic error in some cases. We need a concept of a collection for that – many of these errors, including both our examples, can be caught by simply checking whether the output is a subset of the input. An obvious way is to just track hashes, but this does not scale.

Instead, we turn to one of the bread-and-butter methods for dealing with sets in streaming applications, the Bloom Filter. Bloom filters are probabilistic data structures which track set membership in fixed-memory with a tunable false-positive rate; they have found great utility in genomics applications for a variety of operations on sequences [5][6]. We use a bloom filter (BF) to track what data has entered a program, and then assert that the data leaving the program is a subset of that which entered by querying against the BF. By attaching our data models to the bloom filter, we essentially type it, and both validate individual records for correctness and the data set as a whole.

3 Implementation

The *barf* prototype is implemented in Python, and uses the screed package for basic FASTA parsing. screed is maintained by my lab, and is used in a lot of our software and that of several other groups. The data models are implemented with simple native Python libraries: each data model is a regex from the `re` module, wrapped in a Python object with a `chechValid` method; when the check fails, we raise an `AssertionError` with the offending data. The pybloom package provides the bloom filter; it is wrapped on a class in the same manner as the data models, with a check function raising an `AssertionError`

on failure. Finally, there is a model for a `FastaRecord`, which is a `namedtuple`; the user selects a data model for the sequence on invocation, and a factory function produces a `FastaRecord` model as appropriate.

A typical invocation would take the form:

```
cat in.fasta | ./barf --sequence-model IUPAC analyze-prog > out.fasta.
```

This is modeled after the GNU `time` command: it takes the target program as input and forks to run it. In this implementation though, the process management is slightly more complex, because we need to intercept `STDIN` and `STDOUT`. It first creates a temporary `FIFO` pipe, and then spawns two processes. The first reads `STDIN` and validates the data, then pushes it to the `FIFO` pipe. The second uses `subprocess` to run the command, which is given the `FIFO` as its `STDIN`; the host process monitors the command's `STDOUT`, validates the data, and finally writes the results to *barf*'s `STDOUT`.

Several test cases are provided with this document. In one, the input contains an invalided DNA sequence. The program will crash with:

```
AssertionError:  SequenceModel<ACGT> failed to match
                "FASTALINEWITHBADTHINGS"
```

4 Related Work

5 Conclusions

References

[1] A. M. Bolger, M. Lohse, and B. Usadel, "Trimmomatic: a flexible trimmer for Illumina sequence data," *Bioinformatics*, p. btu170, 2014.

[2] M. R. Crusoe, G. Edvenson, J. Fish, A. Howe, E. McDonald, J. Nahum, K. Nanlohy, H. Ortiz-Zuazaga, J. Pell, J. Simpson, and others, "The khmer software package: enabling efficient sequence analysis," *URL http://dx. doi. org/10.6084/m9. figshare*, vol. 979190, 2014.

[3] J. Gray, "Why do computers stop and what can be done about it?," in *Symposium on reliability in distributed software and database systems*, pp. 3–12, Los Angeles, CA, USA, 1986.

[4] D. Lipman and W. Pearson, "Rapid and sensitive protein similarity searches," *Science*, vol. 227, pp. 1435–1441, Mar. 1985.

[5] R. Chikhi and G. Rizk, "Space-efficient and exact de Bruijn graph representation based on a Bloom filter," *Algorithms for Molecular Biology*, vol. 8, no. 1, p. 1, 2013.

[6] J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J. M. Tiedje, and C. T. Brown, "Scaling metagenome sequence assembly with probabilistic de Bruijn graphs," *Proceedings of the National Academy of Sciences*, vol. 109, no. 33, pp. 13272–13277, 2012.