"Read Me" for `CSSR`

Kristina Lisa Klinkner and Cosma Rohilla Shalizi
kklinkner@gmail.com, cshalizi@stat.cmu.edu

Last revised 11 September 2005

Contents

1 General

`CSSR` tries to infer the minimal Markovian model capable of generating a time-series, or set of time-series from the same source. The program implements the algorithm proposed in the paper ""Blind Construction of Optimal Nonlinear Recursive Predictors for Discrete Sequences", hereafter BC.[1] We won't describe the algorithm in any detail here (see BC for that), but the next two paragraphs say a little about what it produces and how it does it.

The output of the algorithm is a set of states which form a Markov chain. Each state has a certain probability of emitting any of the symbols in the original time series. The current state and the symbol it emits fix the next state. (The states are "future-resolving", if you're from nonlinear dynamics, or "deterministic", if you're from automata theory.) Each state, moreover, corresponds to a distinct set of strings, in the following sense. If the state A contains a string w, and at time t the time-series ends with w, then at time t the Markov chain is in state A. The set of states, their transition probabilities and connections, is called the state machine.

The algorithm uses a recursive inference procedure to find the simplest set of states with the above properties that can reproduce the statistical properties of the data. If we could give the algorithm an infinitely long time series, and let it consider infinitely long sub-strings, it would produce the *causal states* of the process, which are its ideal predictors (see BC for a formal definition). Since we have only finite data, there is always some probability that the inferred or estimated states are not the true causal states. Nonetheless, for the rest of this file, when we say "causal states", we mean the estimated causal states.

[1] Cosma Rohilla Shalizi and Kristina Lisa Shalizi, pp. 504–511 of Max Chickering and Joseph Halpern (eds.), *Uncertainty in Artificial Intelligence: Proceedings of the Twentieth Conference*, available from `http://arxiv.org/abs/cs.LG/0406011`.

2 Obtaining and Installing the Program

The code for `CSSR` can be obtained from `http://bactra.org/CSSR/`. If you'd like to set up a new archive for it, we'd appreciate hearing about it.

From any site, download the CSSR-v0.1.tar.gz file for the code. When gunzipped and untarred, this will produce a directory called CSSR-v0.1, containing all the necessary header and source code files, a copy of this documentation, the release license, and a make file. Running `make` inside that directory will produce an executable, which should be moved to someplace in the command path. On most Unix systems, the following sequence of commands will create the executable and put it in the your `bin` directory, usually part of your command path.

```
gunzip CSSR-v0.1.tar.gz
tar xvf CSSR-v0.1.tar
cd CSSR-v0.1
make
cp CSSR ~/bin/
```

The code has been successfully compiled with gcc 3.1 on Macintosh OS X (10.2), with gcc 3.2 on Linux (Red Hat 9), with gcc 3.3 on Macintosh OS X (10.3) and with Microsoft Visual C++ on Windows 98. On some systems, compilation may produce warnings about escape sequences or the use of deprecated headers. These can be safely ignored.

3 Usage

`CSSR` is a command-line program.

`CSSR` *alphabetfile datafile maxlength* `[-m] [-s siglevel] [-ch]`

The first argument is the name of a file which contains all the symbols in the alphabet used by your data. The second argument is the name of the data file itself. Only one data file can be used, but it may contain multiple lines. The third argument is the maximum length of history to examine, which we will abbreviate by L here. The three trailing flags are optional arguments. Set the `-m` flag if the data-file contains multiple lines (see below). If the `-s` flag is set, it must be followed by a significance level; the default value is 0.001. (For more on setting parameters, see Section 4.) If the `-ch` flag is set, the program will use the χ^2 significance test, rather than the default Kolmogorov-Smirnov test.

In multiple-line mode (entered through the `-m` flag), each line of the data file is treated as a distinct time series from the same source. (Technically, as independent realizations of a single stochastic process.) The lines need not be of equal length.

The program will create the following files after running, where *datafile* is the name of the file the data is in:

1. *datafile*_results
2. *datafile*_info
3. *datafile*_state_series
4. *datafile*_inf.dot

(1) contains the information on the inferred states and their properties. For each state, it gives:

- the histories of length $L-1$ and L in the state
- the probability that the state will emit different symbols (e.g. $P(a) = x$) and the states transitioned to when those symbols are emitted (e.g. $T(a) = s$)
- the observed probability of the state in the data-stream

(2) is the file containing the metrics run on the causal state machine. These are:

- the number of states
- the statistical complexity (entropy of the states)
- the entropy rate
- three measures of the difference between the empirical distribution of symbol sequences, and that generated by the inferred causal state machine. These are: the divergence or relative entropy between the inferred and empirical distribution; the relative entropy rate, or increase per symbol in the divergence; the total variational distance ("variation") between the two distributions.

Note that the relative entropy and the relative entropy rate can be infinite; this indicates that the inferred model gives a probability of zero to a sequence in the data.

Note that sometimes, when the relative entropy should be very small (order of 10^{-6} bits or less), numerical rounding errors result in a negative number being calculated. In these cases, the program outputs zero. Similarly, the complexity of one-state machines is sometimes reported as -0.

(3) is the series of causal states in the data. That is, the program scans through the data, looks up which state the history-to-date is in, and writes the corresponding symbol to this file. What you see is then the trajectory through estimated causal state space of the data/process. Multiline data results in a multiline state-series file.

(4) represents the state machine as a labeled directed graph, where each state has its own node, and each transition between states its own edge. The file is for use with the program `dot`, available from `http://www.graphviz.org/`.

4 Some Suggestions About Parameters

It is always good to use as much data as you can. While it is generally good practice to hold back some data for testing or cross-validation, we recommend that this be minimized. High-entropy processes are especially data-hungry. (See BC.) For reference, let us call the number of data-points N.

The two key parameters of the program are the maximum history length, L, and the significance level used in the test, s. For any given process, there is a minimum history length Λ, such that the true states cannot be found if $L < \Lambda$. The number of states returned may be less than the correct number *or higher*. If $L \geq \Lambda$, and there is enough data, there will generally be a "plateau" of values of L where the correct number of states is returned. For fixed N, if we keep increasing L, then past a certain point there are not enough examples of each string in the data. This tends to erroneously create new states, which spawn others through determinization. Thus there is generally a "blow-up" when L is too large (relative to N and s). A rough guide-line is to limit L to no more than $\log_2 N / \log_2 k$, where k is the alphabet size (see BC for details).

In general, one should use as small an L as possible, since under-sampling, even before the blow-up, will reduce the accuracy of many probability estimates. Blow-up can be delayed by reducing s — that is, reducing the probability of mistakenly splitting a state — but this carries the risk of failing to create valid new states. We suggest exploring the data at low L and high s initially, and then increasing L and lowering s. If a stable architecture is found, it should be recorded at the lowest possible L.

5 Known Issues

There are a few known issues with `CSSR` 's behavior. These arise from certain unavoidable approximations in the determinization procedure. (For details, see below.)

After creating and determinizing the states, `CSSR` goes over the input data and attempts to produce the corresponding sequence of causal states, both as a filtering procedure, and as part of estimating the fit of the causal-state model (see Section 3, item (3)). Typically there will be some initial number of symbols which it must read before "synchronizing" to one of the causal states, after which it will follow the transitions deterministically, and never de-synchronize. Because of the approximations mentioned, it can happen that certain transitions are deleted from the causal state model which shouldn't be. This can lead to three kinds of problem: (1) `CSSR` can never synchronize to a state at all; (2) it has to re-synchronize at some point; (3) it encounters an apparently "impossible" sequence in the data.

In case (1), `CSSR` writes the message "Error: Could not synchronize, cannot use this data" to standard error and halts execution. In case (2), it produces a warning message on both standard error and the `_info` output file. In case (3), it produces a warning, and discounts that particular string from various

calculations.

The best approach to these problems is to use a longer history length, if possible, and to provide more data. In the case of a third error, it can sometimes arise if a particular string appears only once, at the very beginning of the data, and sometimes removing that string from the data-file fixes matters.

All three errors arise because we have only finite-length histories available to us, while what we want are really infinite ones. This forces us to make certain approximations in our implementation of the theory. Specifically, in the determinization procedure, we are forced to "guess" which state certain strings belong to, even though we have not directly examined these strings. The particular approximation scheme (or "closure") we have adopted may be investigated by consulting the code in AllStates.cpp. (Others are possible, but this one seemed to give the best over-all results.) Sometimes this closure will "guess wrong", and possible transitions will be labeled forbidden, etc. In these cases, extending the history length *should* solve the problem, if enough data is available for reliable inference. Similar approximations must be made in determining whether or not a given state is transient or recurrent on the basis of finite data. This occasionally leads to a recurrent state being labeled transient, which in turn is the most common cause of the code mistaking an actually-occurring string for an impossible one. Again, the best approach is to provide more data, and a longer history.

6 Bug Reports, Fixes, Modifications

We welcome bug reports or reports of strange behavior. These reports are welcomed with more enthusiasm when accompanied by successful modifications to the code! (See the accompanying file on the Gnu Public License for information about modifying the code.) Even if you can't fix it, however, please do tell us about it; at the least it will be documented for other users.

If you modify `CSSR` , and want to make the resulting program available, please let us know. We are happy to provide a link, and have a (limited) capability to host alternate versions and descendants. Also, if you use `CSSR` successfully in some application, we'd love to hear about it.

Please check `http://bactra.org/CSSR/` for up-to-date contact information.

7 Some Details on the Code

There are twelve classes in the program. They are each comprised of a .cpp file and a .h file, except for ArrayElem (contained in G_Array) and StringElem (contained in State), as well as a source file Main, and header files Common.h and Main.h.

AllStates	Contains and manipulates growable array of states
ArrayElem	Each element in the growable array
G_Array	A generic growable array
Hash	Hash table which points from histories to their parent states
Hash2	Hash table which points from indices to symbol/alpha values
Machine	Manipulates the determinized state machine and runs metrics
ParseTree	Reads in data file and stores all strings present in file up to length L the maximum length input at the command line
States	A state, contains all data for a single state
StringElem	the element containing the history for a single state
Test	Performs statistical significance tests
TransTable	Stores initially estimated transitions from all histories of length L in any given state. This class is used by AllStates to check for transient states before determinization.

For brief descriptions of the classes, see the top of their source files. Note that the terms "string" and "history" are used interchangeably in the program. The terms both correspond to the concept of a "history" (as described in BC), but the program implements these as strings of symbols.

Also, after initial state splitting, all strings/histories of less than maximum length minus one are deleted. This has no effect on the outcome of the algorithm and saves time and space.

Lastly, the removal of transient states prior to determinization implemented in the `AllStates::CheckConnComponents` procedure is not strictly necessary. With a different implementation, the deletion of these states could automatically occur during the determinization process itself (see the pseudocode in BC for details), and the outcome of the algorithm would be the same. As it is implemented here the code is slightly redundant.

7.1 Revision History

0.1	11 September 2005	Minor bug-fixes to multi-line mode
0.1	7 August 2005	Corrects minor numerical bugs. These only affected the calculations of relative entropy, relative entropy rate and total variation distance, and then only by about $O(k/N)$.
0.0	7 August 2003	First public release.

8 Credits and Acknowledgments

`CSSR` was written by KLK, with some help in debugging and testing from CRS; this documentation was jointly written. Parts of the `CSSR` code were written at the Santa Fe Institute and at the University of San Francisco. Support at SFI was provided by SFI's core grants from the National Science Foundation and the MacArthur Foundation, by the NSF's Research Experience for Undergraduates program, and by the Dynamics of Learning group, under DARPA contractual

agreement F30602-00-2-0583. Support at USF came from the Clare Boothe Luce Foundation.