Neurobayes-Iterators

Bruno Daniel

April 18, 2011

Contents

1 Introduction and principles **2**

2 Iterator objects **4**

3 Data sources **5**

4 Data sinks (consumers) **8**
4.1 Park iterators 10

5 Data flow control **10**
5.1 Database functionality analogous to select-statements in SQL 10
5.2 Iterators seeing more than one data row 13
5.3 Row-type converters 15
5.4 Abstract Machine iterators 16
5.5 Text utilities 16
5.6 Other data flow iterators 18

6 Branches in iterator chains **18**

7 Using iterator chains as functions on data rows or data matrices **19**

8 Column-wise and chunkwise access **20**

9 Mechanisms for algorithmic iterators **20**
9.1 Dynamic typing (generic values) 21
9.1.1 Serialization and Deserialization 22
9.1.2 Functions and macros for generic values, garrs and gmatrices 23
9.2 Data structures and formats for expertise and monitoring data 23
9.3 Basic iterators for learning algorithms 25

10 Neurobayes **26**

11 Other machine learning algorithms **27**
11.1 Special Neural Networks . . . 27
11.2 Clustering algorithms . . . 29
11.3 Others . . . 29
11.4 Text mining algorithms . . . 29

12 Components for monitoring, logging and debugging **29**
12.1 Measuring iterators . . . 30
12.2 Evaluating iterators . . . 30
12.3 Displaying iterators . . . 30
12.4 Debugging components . . . 31

13 Live utilities **32**
13.1 Threads . . . 32
13.2 Parallelization . . . 34
13.3 Iterator buffers for inter-thread communication . . . 34
13.4 Snapshots . . . 36
13.5 Socket pairs . . . 36
13.6 Web-GUI utilities and XMLHttpRequests . . . 37
13.7 Other components for live applications . . . 38

14 Performance optimizations **38**
14.1 Sorting tricks . . . 38

15 Links to further documentation and examples **39**
15.1 Unit tests as examples . . . 40

16 Installation **41**

17 Frequently asked questions **42**

A Appendix: Software architecture of the Neurobayes-tools (Bottom-up design) **42**

1 Introduction and principles

Some software packages are based on a single coherent concept with everything revolving around it and that is easy to understand. They are usually quite successful with it: Excel is based on *spreadsheet cells* linked together by formulas, Matlab is based on (mostly numeric) *matrices* and on calculating new matrices from old ones, Lisp is based on *s-expressions*, the Unix shell is based on *rows of ASCII text* passed between processes in process pipes, SQL is based on *database tables* combined in queries, Postscript, Forth, the Java Virtual Machine and some firmware assembly languages are based on a *stack* for data and/or commands, HTML and XML are based on *tags* in ASCII texts, some of them forming links to other documents.

What could be the best data structure for predictive modeling? Most machine learning algorithms use training data in the form of a series of training events, each of which being a list of numerical measurements characterizing the event. The whole data may be too large to fit into primary storage – while RAM prices are rapidly falling according to Moore's law, the amount of data to be processed is also rapidly increasing. Thus a matrix-based approach would be too limited and inefficient; Neurobayes is based on the *iterator* concept instead [1]. An iterator is a data structure for traversing through a series of numerical or textual rows of data. Only one row or a small period of rows are actually kept in memory instead of millions or billions of rows. Piecing together iterator chains feels very much like plumbing of data streams.

In a chain of iterators one iterator receives the current row, makes some changes and passes it to the next without having to accumulate the resulting rows in a temporary data structure first, as would be the case in a matrix-based approach requiring many temporary matrices or convoluted optimizations avoiding them. By saving RAM, the processor cache works more efficiently, so time is saved as well.

Iterators are analogous to processes in Unix pipes. They may be chained together as in

```
cat <Filename> | uniq | grep ... | more .
```

The ASCII lines are not accumulated in intermediate steps, and even the last process (`more`) only accumulates a period of rows potentially much smaller than the entire data. It is this last component that pulls the rows through the chain. As long as the user doesn't scroll down, no further data is requested (i.e. pulled by `more`) and all the processes stand still and wait. Components that pull like this are named **consumers** in Neurobayes and are marked by the suffix "`_of_it`". All other components are called **iterators** and carry names like `make_..._it`.

By introducing iterator chains, programs may be modularized without significant performance penalty. Recurrent structures for manipulating data flows in programs may be factored out into iterators representing processing steps (see below for examples). Without iterators, the processing steps could be separated, too, but because this would involved temporary data accumulations, it would result in an intolerable slowdown of the programs.

Due to the modularity of iterator chains, the processing steps may be written, understood and tested separately, the functionalities don't pollute each other and the components are simpler. Automatic **unit testing** is facilitated and it's easier for the processing steps to be written by different people. It is also quite easy to break up a program into two or more processes (see the remarks on `make_csv_it` and `csv_of_it` in sections 3 and 4). Iterators may be combined in iterator chains in any way except for restrictions like one iterator needing results of a previous one.

Every iterator produces rows of a length fixed at run time. The names of the rows' entries (columns) constitute the *headline* of the iterator. Iterators normally find the columns of previous iterators of an iterator chain in their headline, so in general neither the number of columns nor their order needs to be known at compile time.

The data rows may be either

- completely **numeric**, called `darr` (double-float array),
- completely **textual**, called `sarr` (string array),
- or **generic**, called `garr` (dynamically typed array for mixtures of different data types which include nested types like arrays, matrices or trees, see the tagged-union data structure described in section 9.1).

There are iterators for converting the rows between the three worlds, see section 5.3.

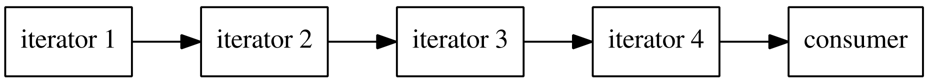
Many applications will begin with strings, transform them to numbers in several steps by extracting parts of columns, by lookup tables and other methods (see section 5.5), and then switch to numbers altogether. More complicated applications that have to keep the strings around or are in need of nested data structures, for instance in natural-language processing, are better advised to use generics. Generics will also be the best choice if the number of entries per row is not constant from row to row, because `garr`s support nested arrays.

Missing numerical values in a row entry are characterized by a special value given by the user, e.g. -999. Missing string values are given as NULL pointers, missing generic values are denoted by the special tag `m`, see section 9.1.

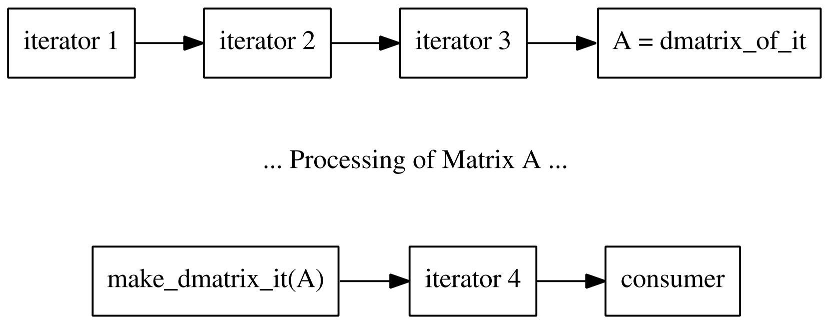
Iterators traverse the data typically in the forward direction, but most iterators may be reset (by calling `reset_it`) to the beginning, offering the same data over again. This is useful for algorithms (e.g. neural nets) that need multiple runs through the data in learning mode.

Random access across a group of rows (*period*) is possible with iterators like `make_map_period_it` which buffers row groups in a matrix, presents the matrix to the user who can change it, and passes its rows on to the next iterator in the chain.

If completely random access across all rows is needed at some point, an iterator chain may be broken up in two parts and the data may be put into a temporary matrix where random access is possible, the result then being presented as a matrix-iterator, e.g. (written schematically)



may be rewritten as:



Data may also be accumulated (in a file or a matrix) on the fly, e.g. for debugging purposes or in order to avoid reexecuting costly processing steps after a reset (see the park-iterators in section 4.1).

For *live* data sources like socket interfaces to a server giving live data, there is no alternative to iterators: Each row must be processed immediately. It is not possible for the individual processing steps to collect the results before passing them on, since this would involve waiting for all the rows from the socket first. Section 13 introduces some special components for live applications.

The smaller and simpler the components, the easier it is to reuse them. This is a general principle: If there are two similar software projects with subtle differences and they are written in a monolithic design, the software of the first may not be reused for the second, except for copy-and-paste-programming with its disastrous consequences. If, on the other hand, they are written in a modular design without performance penalty with, say, 20 processing steps, 18 of these may be reused and only 2 components need to be rewritten, and those 18 are already well-tested.

Neurobayes' iterators and consumers are written in C. Since almost every programming language has a decent C interface, often including callbacks and raw access to C data structures without the need to copy them, it is easy to wrap the software for other languages efficiently (like Python, Java, C#, Perl, SAS, SPSS, R, Matlab, Mathematica) and offer the functionality for them as well. This has been done for Python. Since the internal calculations (e.g. in `make_statistics_it`) are written in C, the performance penalty for using a scripting language like Python is reduced – if the language is only used to piece the iterator chain together it is even negligible. Graphical programming, i.e. connecting boxes representing iterators and consumers in a GUI, is conceivable. (It will still involve mathematical expressions written by the user, as in data filters.)

It is possible to use if-statements or even loops in iterator chains. This way, the user may switch on or off certain iterators according to context or repeat subchains of iterators.

Iterator chains may also be branched, e.g. in order to separate rows belonging to different contexts – each branch then only sees rows of a single context which makes them simpler – or for parallelization (see section 6 for details).

Recurrent subchains of iterators may be factored out into a new iterator summarizing them (see the iterator `make_wrap_it` in section 5.6). This way, even the code for specifying iterator chains may be reused.

2 Iterator objects

Iterator objects implement the following methods:

- `get_headline_ar_it`: returns the headline, i.e. a list of column names, of the iterator.
- `get_n_rows_it`: returns the number of rows of the iterator. Many iterators don't know their number of rows in advance and return -1, e.g. the `socket_it` that returns live data. This value is overwritten by a valid value as soon as the end of the data is reached.
- `get_n_columns_it`: returns the number of columns of the iterator.
- `get_rowno_it`: returns the number of the current row.
- `get_drow_it`: returns the current row in numeric double-float format. This method is not always implemented for iterators operating on strings. Requesting a numeric row results in a wave of `get_drow_it` calls through the iterator chain. (This wave may be diverted to another type, see section 5.3.)
- `get_srow_it`: returns the current row in string format. This method is not implemented for purely numeric iterators. Requesting a row of strings results in a wave of `get_srow_it` calls through the iterator chain.
- `get_grow_it`: returns the current row in generic format. This method is not implemented for all iterators. Requesting a row of strings results in a wave of `get_grow_it` calls through the iterator chain.
- `close_it`: closes the iterator, frees its memory, closes files, connections and other resources. Normally, an iterator will close itself automatically when reaching the end of the data, so if the user needs a reset, this must be announced by passing the argument `keep_open = true` to the constructor and the iterator must be closed explicitly by calling `close_it` in this case. `close_it` may also be used to abort right in the middle of an iteration.
- `close_rec_it`: convenience method; calls `close_it` on all members of an iterator chain.
- `reset_it`: resets the iterator to the beginning of the data. This method is not always implemented, e.g. for `socket_it`, or sometimes it's not efficient to repeat the same calculations of the attached iterators after a reset. Such cases can easily be circumvented by inserting a park-iterator (see section 4.1) into the iterator chain.
- `promote_it`: tells the iterator about mode updates of following iterators and returns the current mode of the iterator. Normally, this method involves a recursive call of the predecessors in the iterator chain.
- `get_headline_hash_it`: returns the headline as a hashtable indexed by the column names and carrying the column indices as values.
- `find_column_index_it`, `find_column_index_error_it`: find the position of a column name in the headline.

Iterators usually carry a reference to their predecessor in the iterator chain.
Many iterators, for instance `map_it` or `filter_it`, receive user functions as arguments. These user functions can always be supplemented by a struct to operate on (a.k.a. "thunk") which is called `enclosed`, an optional destructor for this struct called `delete_enclosed` and an optional resetting function called `reset_enclosed` which will be called by `reset_it` internally.

The handling of column indices is facilitated by the `cols` argument in connection with Neurobayes' `codegen2` mechanism. In an inner loop, column indices are much more efficient than looking up the column every time by its name (which is a string). Note: If the same column indices object is initialized twice in the same iterator chain and columns have been dropped in between, inconsistencies will result.

3 Data sources

These are the iterators that may be used to start an iterator chain. They read from some data source located in RAM, in a file or on the net:

- `make_dmatrix_it`: reads from a numeric (double-float) matrix in RAM. A headline may be added.
  row types: darr.
- `make_smatrix_it`: reads from a textual (string) matrix in RAM. A headline may be added.
  row types: sarr.
- `make_gmatrix_it`: reads from a generic (dynamically typed) matrix in RAM. A headline may be added.
  row types: garr.
- `make_darr_row_it`: reads from a horizontal numeric array in RAM, i.e. this data source has only one row.
  row types: darr.
- `make_sarr_row_it`: reads from a horizontal string array in RAM, i.e. this data source has only one row.
  row types: sarr.
- `make_garr_row_it`: reads from a horizontal generic array in RAM, i.e. this data source has only one row.
  row types: garr.
- `make_fct_it`: generates rows by calling a function on the current column and row index. This may be used to simulate data, for example. A headline may be added.
  row types: darr, sarr, garr.

- `make_csv_it`: reads from a CSV file; allows a user function to translate the strings into double-float values for later use. CSV is an ASCII data format understood by Excel and many other software packages. If the filename is not given, it reads from standard input.
  row types: darr, sarr, garr.

- `make_ascii_lines_it`: reads ASCII lines from a file or stdin as strings. The resulting iterator has only one column. If `is_process = true`, the filename is interpreted as a shell command for a process to read from.
  row types: sarr, garr.

- `make_h5_it`: reads data from an HDF5 file. HDF5 is a highly efficient self-compressing data format [2].
  row types: darr (so far).

- `make_root_it`: reads data from a ROOT file. ROOT is a data-analysis library from CERN [3].
  row types: darr.

- `make_re_it`: reads data as hits of a regular expression in a string.
  row types: sarr, darr.

- `make_mysql_it`: reads data from a query in a MySQL database; allows a user function to translate the strings into double-float values.
  row types: sarr, darr.

- `make_oracle_it` (planned) : reads data from a query in an Oracle database; allows to translate the strings into double-float values.
  row types: sarr, darr.

- `make_sqlite_it`: reads data from a query in an Sqlite database; allows to translate the strings into double-float values.
  row types: sarr, darr, garr.

- `make_excel_sheet_it`: reads data from an Excel table on a chosen sheet of an Excel file with optional row and column offsets to start from.
  row types: sarr, garr.

- `make_socket_it`: reads data from a socket connection on the intra- or internet, see section 13.5.
  row types: darr, sarr, garr.

- `make_darr_it`: reads from a vertical numeric array in RAM, i.e. this data source has only one column.
  row types: darr.

- `make_sarr_it`: reads from a vertical string array in RAM, i.e. this data source has only one column.
  row types: sarr.

- make_buffer_it, make_buffers_it: pull rows from one or a list of iterator buffers, respectively (see section 13).
  row types: darr, sarr, garr.

- make_webpage_it (planned) : opens a webpage as a series of ASCII lines in a single column as in make_ascii_lines_it. Besides the URL, GET or POST parameters may be added for Web forms. Supported protocols: HTTP, HTTPS. This may be used in combination with make_re_columns_it and other text utilities (see section 5.5) for writing Web spiders. The C-library W3C-libwww is used internally.
  row types: sarr.

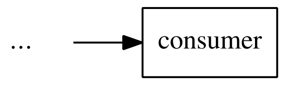
- make_directory_it: returns the filenames of a directory in the column filename.
  If properties_w = true, the following columns will be added (This is only supported in garr-mode):

  - mode (including the file type and access rights)
  - user_id
  - group_id
  - size (in bytes)
  - atime
  - mtime
  - ctime

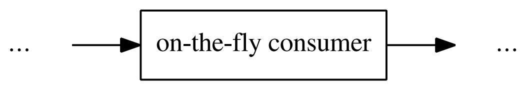
  Except for the mode which has tag f (flag), all these columns have tag l (long integer). The times are given as datetime schemes, i.e. in the format YYYYMMDDHHMMSS as a long integer.
  row types: sarr, garr.

- make_directory_tree_it: returns the filenames of a directory tree. The directory tree will be traversed in depth-first order. The output format is the same as in make_directory_it except for an additional sarr column pathar (path array), if requested.
  If max_depth > 0, it limits the depth of the tree.
  It's easy to implement the Unix command find as a combination of make_directory_tree_it, make_re_filter_it and make_filter_it.
  row types: sarr, garr.

4 Data sinks (consumers)



These are consumers that may be used to end an iterator chain. They put the received data into some data structure in RAM, in a file or on the Net. Alternatively, the user may just write a loop requesting the rows directly by calling get_drow_it for numeric rows or get_srow_it for rows of strings or get_grow_it for generic rows on the last iterator in the chain.



Most of the consumers also have an on-the-fly version that passes the rows to a following iterator as well, e.g. `make_csv_of_it` is the on-the-fly version of `csv_of_it`.

- `dmatrix_of_it`, `make_dmatrix_of_it`: the opposite of `make_dmatrix_it`; saves data in a double-float matrix.
  row types: darr.
- `smatrix_of_it`, `make_smatrix_of_it`: the opposite of `make_smatrix_it`; saves data in a textual matrix.
  row types: sarr.
- `gmatrix_of_it`, `make_gmatrix_of_it`: the opposite of `make_gmatrix_it`; saves data in a generic matrix.
  row types: garr.
- `darr_of_it`: the opposite of `make_darr_it`; saves data in a vertical array.
  row types: darr.
- `csv_of_it`, `make_csv_of_it`: the opposite of `make_csv_it`; saves data in a CSV (comma-separated values) file; allows a user function to translate double-float values back into strings. If the filename is not given, it writes to standard output. This may be used in combination with `make_csv_it` to piece together separate processes written as iterator chains on the same machine just by using the Unix pipe mechanism. Appending rows to old files is also supported.
  row types: darr, sarr, garr.
- `ascii_lines_of_it`, `make_ascii_lines_of_it`: prints ASCII lines of a chosen string column into a file or to stdout. If `is_process = true`, the filename is interpreted as a shell command for a process the output is sent to.
  row types: sarr, garr.
- `h5_of_it`, `make_h5_of_it`: the opposite of `make_h5_it`; saves data in an HDF5 file. Appending rows to old files is also supported.
  row types: darr (so far).
- `split_h5_of_it`: splits the data by user function or by random choice into two HDF5 files, e.g. for separating training data from test data.
  row types: darr.
- `root_of_it`: the opposite of `make_root_it`; saves data in a ROOT file.
  row types: darr.
- `mysql_of_it`, `make_mysql_of_it`: the opposite of `make_mysql_it`; saves data in a MySQL table; allows a user function to translate double-float values back into strings. The table may optionally be recreated in the database. Non-double-float types for columns and primary keys may be specified.

Appending rows to old MySQL tables is also supported.
row types: darr, sarr.

- `oracle_of_it`, `make_oracle_of_it` (planned) : the same for Oracle databases.
  row types: darr, sarr.

- `sqlite_of_it`, `make_sqlite_of_it`: the same for Sqlite databases.
  row types: darr, sarr, garr.

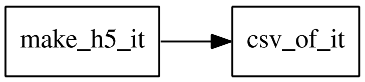
- `socket_of_it`: the opposite of `make_socket_it`; (see section 13).
  row types: darr, sarr, garr.

- `excel_of_it`, `make_excel_of_it` (planned) : writes data into an Excel table on a chosen sheet of an Excel file with optional row and column offsets to start from. Optionally, a range of cells around this Excel table may be cleared if there is any old data.
  row types: sarr, darr, garr.

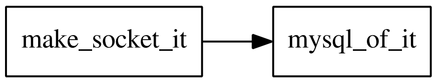
- `buffer_of_it`, `make_buffer_of_it`, `make_buffers_of_it`, `buffers_of_it`: push rows into one or a series of iterator buffers (see section 13).
  row types: darr, sarr, garr.

- `nothing_of_it`: drops the data without using it; corresponds to `> /dev/null` in shell programming.
  row types: darr, sarr, garr.

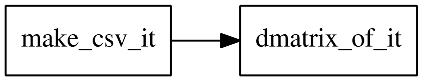
Format converters are easy to write by just coupling a consumer to an iterator, e.g.



will convert an HDF5 file to CSV,



will put data from a socket connection into a database,



will read a CSV file into a matrix and so on.

4.1 Park iterators

These are iterators that park the data in some data structure or file along the way and offer it to the next iterators in the iterator chain as well. After a reset, the data is no longer retrieved from the predecessors in the iterator chain but from the data structure or file directly.

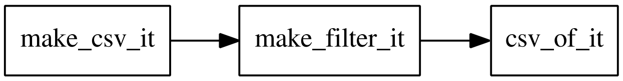
- `make_park_dmatrix_it`: parks the data in a double-float matrix.
  row types: darr.
- `make_park_smatrix_it`: parks the data in a textual matrix.
  row types: sarr.
- `make_park_gmatrix_it`: parks the data in a generic matrix.
  row types: garr.
- `make_park_h5_it`: parks the data in an HDF5 file.
  row types: darr (so far).
- `make_park_csv_it`: parks the data in a CSV file.
  row types: darr (so far).

5 Data flow control

5.1 Database functionality analogous to select-statements in SQL

These iterators mimic the functionality of queries in relational databases while other sections of this document offer functionality needed in predictive modeling but missing in SQL.

- `make_filter_it`: drops rows that are no longer needed; corresponds to `where` in SQL and `grep` in shell programming. Data filters are easy to write by just coupling a consumer to an iterator and putting a filter in between:

  will filter CSV files.
  row types: darr, sarr, garr.
- `make_map_it`: transforms a data row into a new data row with more or fewer entries by user function; corresponds to certain aspects of the part between `select` and `from` in SQL statements.
  row types: darr, sarr, garr.
- `make_new_columns_it`: adds new columns that are calculated from the old ones. This is an optimized special case of `make_map_it` that avoids copying the old columns.
  row types: darr, sarr, garr.
- `make_constant_column_it`: adds a constant number in a new column.
  row types: darr, sarr (so far).

- `make_map_in_situ_it`: changes entries in rows by user function without copying and without changes to the number of columns. This is also an optimized special case of `make_map_it`.
  row types: darr, sarr, garr.

- `make_drop_columns_it`: drops columns that are no longer needed.
  In the numeric case, it's often more efficient to use an HDF5 file with column-wise access, see section 8.
  row types: darr, sarr (so far).

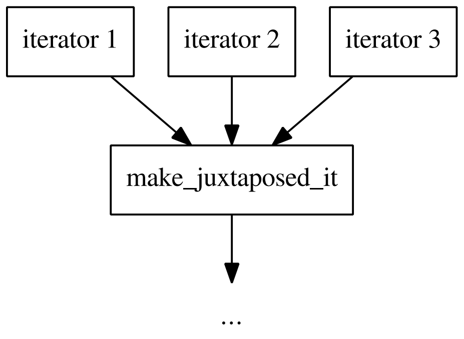
- `make_en_passant_it`: an alias for `make_map_in_situ_it` used for just looking en passant at the rows flowing through the iterator chain. The row contents shouldn't be changed in the user function. Many on-the-fly-consumers are based an `make_en_passant_it`.
  row types: darr, sarr, garr.

- `make_column_sel_it`: chooses columns of the previous iterator in an arbitrary order and may even repeat columns.
  In the numeric case, it's often more efficient to use an HDF5 file with column-wise access, see section 8.
  row types: darr, sarr (so far).

- `make_rename_columns_it`: renames columns; corresponds to `as` after `select` in SQL statements.
  row types: darr, sarr, garr.

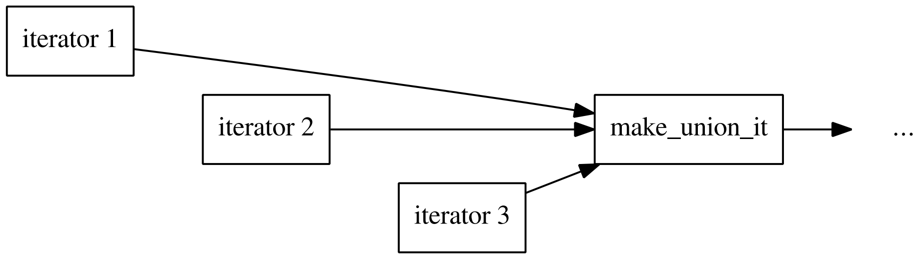
- `make_random_choice_it`: random choice of rows.
  row types: darr, sarr, garr.

- `make_juxtaposed_it`: juxtaposes the corresponding rows of several iterators to larger rows.

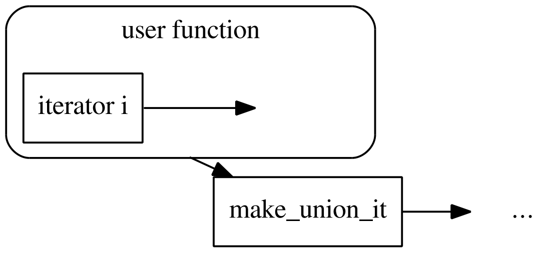


  row types: darr (so far).

- `make_union_it`: offers the data of several iterators one after the other. The input iterators may all be opened in advance:



or by a user function called in a loop n times:



row types: darr (so far).

- `make_group_by_foldarr_it`: groups rows by equal values in one or more columns and calculates some overall result for each row group; corresponds to `group by` in SQL statements but is more general, because arbitrary aggregate functions (not just summing, counting, averaging but arbitrary user functions) are possible.
  Special case: `make_foldarr_it` (all the data comprise a single row group).
  Special cases as consumers: `group_by_count` (counts rows in row groups), `sum_of_it_column` (sums one column over all data); `sumall_of_it` (sums all columns); `avgall_of_it` (averages all columns); `fold_of_it` (calculates a single value from all data); `foldarr_of_it` (calculates an array of values from all data).
  In most cases it is much easier to use the cl-iterators below with the option `only_last_in_group`, see section 5.2.
  row types: darr (so far).

- `make_join_ordered_it`: joins two iterators according to one or more compared columns; corresponds to `join...using` in SQL statements. The join either makes use of a consistent ordering of the two data sources or of a lookup table to an accumulation of the second iterator or to an optimized mixture of both if several columns are compared.
  row types: darr (so far).

- `make_resort_via_h5_it`: resorts rows of the previous iterator; corresponds to `order by` in SQL statements and `sort` in shell programming. The data needn't fit into primary storage,

because they are buffered internally in an HDF5 file. A stable sorting algorithm (Merge sort, not Quicksort) is used.
Internal optimizations of `make_resort_it`:
If there are many passive columns in the HDF5 file besides the key columns to be sorted, an optimization is used in order to reduce the number of file accesses: First only the key columns and a column for the permutation vector are sorted. Then the permutation vector is applied to the complete rows. Currently, this optimization takes place only inside HDF5 memory buffers, but an improved implementation that takes advantage of column-wise access to HDF5 files will be able to permute rows of complete HDF5 files efficiently (function `hdf5_permute_rows`).
Section 14.1 explains how to use sorting algorithms efficiently.
row types: darr (so far).

- `make_resort_groups_it`: resorts rows within row groups that fit into primary storage. The order of the groups needn't be changed, because they are already supposed to be sorted. This is a recurrent special case of `resort_it` that can be optimized much better than the general case. A stable sorting algorithm is used that takes place in primary storage. Only a single row group is stored in RAM at a time. Please don't use this iterator for extremely large row groups that don't fit into RAM.
  Section 14.1 explains how to use sorting algorithms efficiently.
  row types: darr (so far).

- `make_limit_offset_it`: an iterator offering only a limited number of rows of the previous iterators, optionally after an offset of n rows. It is analogous to `limit ... offset ...` in MySQL or `head -n ...` in the Unix shell. Optionally, an additional user function may be given for the start condition that is applied after the offset and before counting the rows for the limit.
  row types: darr, sarr, garr.

- `make_start_stop_it`: similar to `make_limit_offset_it` but controlling the first and last row taken from the previous iterator by user functions.
  row types: darr, sarr, garr.

5.2 Iterators seeing more than one data row

It is sometimes necessary to compare successive rows, add missing rows or form timeseries out of row sequences:

- `make_row_sequence_it`: juxtaposes successive rows to a larger row.
  row types: darr (so far).

- `make_unsparse_it`: transforms data in a sparse format into complete rows. Gaps are either filled by a user-specified `null_value` or by a previous value in the row group.
  row types: darr.

- `make_resparse_it`: the opposite of `make_unsparse_it`; transforms complete data rows back into a sparse format. Occurences of `null_value` are left out.
  row types: darr.

- `make_interpolation_it`: adds missing rows between the incoming rows according to a user function.
  row types: darr.

- `make_interpolate_missing_it`: special case of `make_interpolation_it` for the recurrent case of adding rows for missing values in a single column; values in other columns are either set to `null_value` or set equal to the previous existing value.
  row types: darr.

- `make_fill_null_value_it`: replaces `null_value` by previous value in a row group.
  row types: darr (so far).

- `make_timeseries_it`: adds new columns carrying values of selected columns in previous (or later) rows inside the same period of rows. This way, a timeseries of selected quantities may be added to the output row. Using array slices it is easy to reference timeseries like these as separate arrays without copying.
  row types: darr.

- `make_period_in_situ_it`: presents a period of rows as a matrix to a user function and thus gives the user a broader view over the period. The user may change the values or add or remove rows in the matrix (the number of columns may not be changed). The iterator then presents the rows of the changed matrix to the next iterator in the iterator chain. By `period = 0` the user may choose to handle row groups of variable size.
  row types: darr (so far).

- `make_map_period_it`: more general case than `make_period_in_situ_it`. The user function is confronted with the matrix of rows of a period and fills a new separate matrix that may have a different number of columns and rows and a different headline.
  row types: darr, sarr, garr.

- `make_neighbors_it`: adds all values of selected columns within the same period of rows as new columns. The rows of a period are here considered as neighbored events and the iterator allows the user to compare certain values of the current row to the values of its neighbors.
  row types: darr (so far).

- `make_merge2_it`: merges the series of rows of two iterators together to form a new sorted series of rows. The individual series must already be sorted.
  row types: darr (so far).

- `make_count_if_val_changed_it`: adds `new_column` which is a counter that is increased if one column of `columns` changes.
  row types: darr

- `make_cumulated_columns_it`: cumulates values of some chosen columns (within row groups defined by `checked_equal_columns` or `end_of_group_column`) by a user function and puts the cumulations into new columns.
  Special cases:

  – `make_cl_sum_it`: cumulative sum.

  - `make_cl_diff_it`: first difference. With boundary mode = `bm_zero`, this is the exact right and left inverse of `make_cl_sum_it`.
  - `make_cl_avg_it`: cumulative average (or moving average).
  - `make_cl_weighted_sum_it`: cumulative weighted sum.
  - `make_cl_max_it`: cumulative maximum.
  - `make_cl_min_it`: cumulative minimum.
  - `make_cl_count_it`: cumulative count.
  - `make_cl_previous_value_it`: value of the previous row in the same row group.
  - `make_cl_emov_it`: exponential moving average (based on a decreasing factor).
  - `make_cl_count_distinct_it`: cumulative count of distinct values in a chosen column.
  - `make_cl_infinite_impulse_response_it`: a general discrete-time-signal filter based on the infinite impulse response (IIR) in digital signal processing. With this component, many linear, time-translation invariant, causal filters (low-pass, high-pass, band-pass filters, PID controllers etc.) may be programmed just by specifying two coefficient arrays. The IIR's spectrum may be calculated by the function `infinite_impulse_response_spectrum`.
  - `make_cl_argmax_it`: position (as a value of a special column, e.g. time) of the cumulative maximum.
  - `make_cl_argmin_it`: position of the cumulative minimum.
  - `make_cl_arg_last_change_it`: position of the last change.
  - `make_cl_only_last_in_group_it`: dummy iterator that switches on the option `only_last_in_group`, see below.

  The last row in each row group is the "group-by" aggregation as in SQL. If the option `only_last_in_group` is set in a cl-iterator, only this row will be passed to the next iterator. In an iterator chain of cumulating iterators where only the last cl-iterator carries this option, several aggregations may be performed in a single run through the data without `reset_it` (see figure 1).
  row types: darr.

5.3 Row-type converters

These are iterators that receive rows of one type and return rows of another. This way, the wave of `get_drow_it` or `get_srow_it` or `get_grow_it` calls through the iterator chain may be diverted to a wave of another kind forth and back.

- `make_numeric_it`: receives textual or generic input rows and produces numeric rows by calling a user function or using a default conversion.

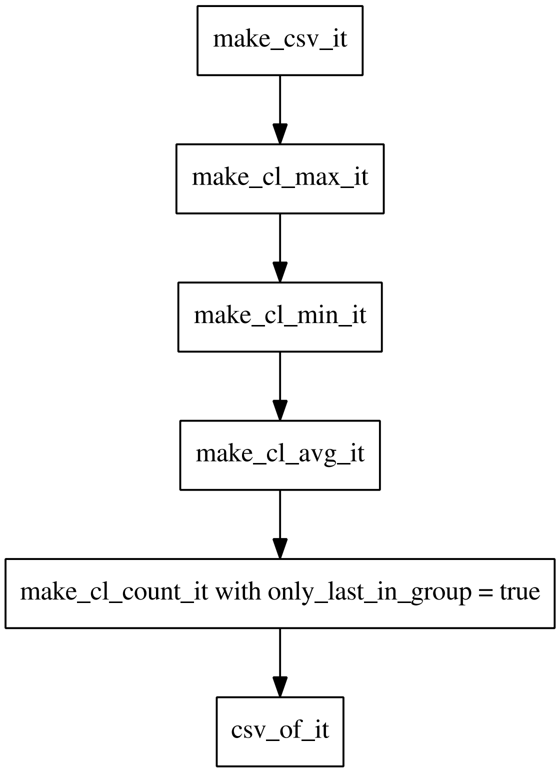
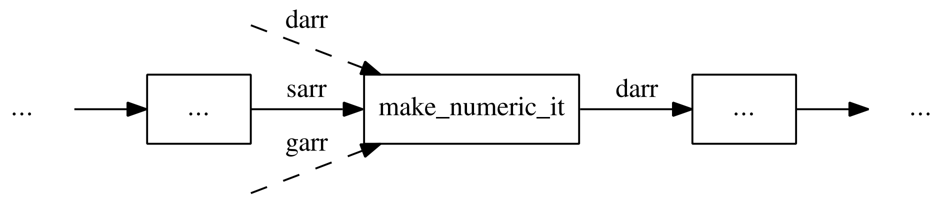


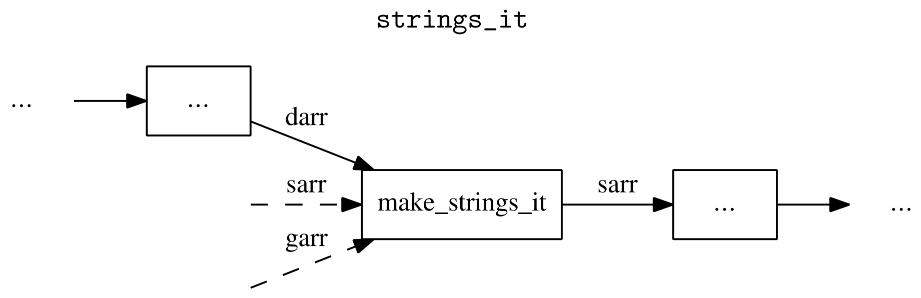
Figure 1: Here is an example for the calculation of the maxima, minima, averages and counts in the row groups of an ordered CSV file. If the option "only_last_in_group = true" is omitted the cumulated maxima, minima etc. will be calculated for each row inside the row groups.

`numeric_it`



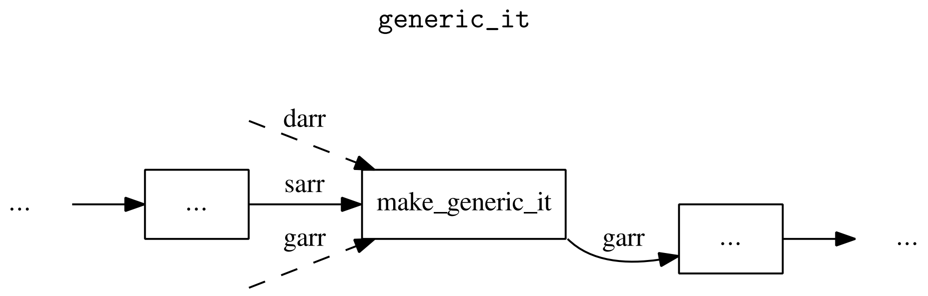
row type (in output): darr.

- `make_strings_it`: receives numeric or generic input rows and produces textual rows by calling a user function or using a default conversion.



row type (in output): sarr.

- `make_generic_it`: receives numeric or textual input rows and produces generic rows.



row type (in output): garr.

5.4 Abstract Machine iterators

- `make_finite_state_machine_it` (planned) : a Mealy machine producing output rows depending on the input rows and the current state. A user function controls these transitions. It is possible to produce no output row for an input in certain states, e.g. in an accumulation. It is also possible to uphold a fixed input row in certain states, e.g. in an interpolation. The user may also pass a thunk `enclosed` and the usual auxiliaries `reset_enclosed` and `delete_enclosed` (see above).
  For further modularization, parallel finite state machines can be realized as successive `finite_state_machine_its` in an iterator chain. In these cases, aggregations and interpolations should be avoided in order to avoid conflicts between the parallel automata. A to-do-marker column for a later aggregating or interpolating iterator in the chain could be used as an alternative.
  Finite subautomata (transitions between substates) are also possible as special cases of parallel finite state machines that act only if the superstate (the state of the parent automaton) has a certain value. For this, the options `state_column` (adding a column for the current automaton state) and `superstate_column`, `superstate_value` (for reading and comparing

the superstate) are provided.
row types: darr, sarr, garr.

- make_pushdown_automaton_it (planned) : similar to make_finite_state_machine_it but more general because it also provides a stack (type: garr) for user variables. One of its applications is the parsing of grammars of abstract languages.
row types: darr, sarr, garr.

5.5 Text utilities

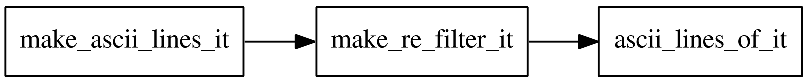
- make_tokenizer_it (planned) : reads tokens according to a series of token descriptions (Perl 5 regular expressions and token IDs) from a string column of the rows of the previous iterator, e.g. an ascii_lines_it, i.e the stream of lines is converted into a token stream. Each token is given as an iterator row with the token ID in the first column and the token parameters as a garr in the second column. This iterator is similar to the Unix tools lex and flex. It is the first step in parsing a format or an abstract (programming) language.
A pushdown_automaton_it (see section 5.4) may be used to parse the tokens according to the grammar of an abstract language which corresponds to the functionality of the Unix tools yacc and bison.
row types: garr.

- make_strings_lookup_it: iterator that translates strings in selected columns to numbers (still saved in an sarr) starting from 1 by forming a lookup-table of the encountered strings and saving it for later use. The lookup-table may optionally be alphabetically sorted and optionally all except the most frequent n values are mapped to 0. This iterator may also be set to automatically find columns that contain only numerical values in their strings. The lookup-table is saved in learning mode and reread in application mode from a dynvars datastructure, see section 9.2. If a lookup-table for a column is already found in this data structure, it is updated.
row types: sarr.

- make_rev_strings_lookup_it: the opposite of make_strings_lookup_it; translates the numbers back into the values found in the lookup table. 0 is mapped back to some default string chosen by the user. The lookup-table is read from a dynvars datastructure, created by make_strings_lookup_it;.
row types: sarr.

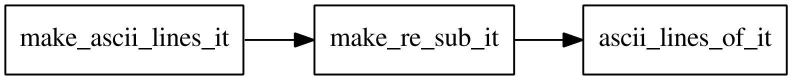
- make_re_columns_it: creates new columns out of matches of a regular expression pattern in a chosen column. If no match is found, the results are set to a default value specified by the user.
row types: sarr.

- make_re_filter_it: filters rows by a regular expression acting on a chosen string column. The Unix tool **egrep** is just this:



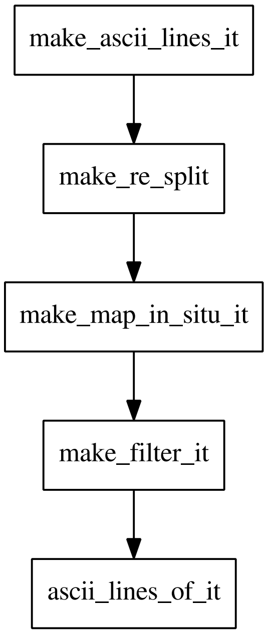
If an `egrep` over all files and subdirectories is needed (”recursive grep”), it suffices to call this in a `make_directory_it` loop.
row types: sarr, garr.

- `make_re_sub_it`: performs a regular expression replacement on a chosen string column. The Unix tool **sed** is just this:



  row types: sarr, garr.

- `make_re_split_it`: splits rows by a regular expression for the separator acting on a chosen string column. This column is replaced by a nested sarr. The separator pattern defaults to white space.
  Unix **awk** scripts may be translated to iterators this way (not showing preliminary and final calculations):



  (The component `map_in_situ_it` is redundant, because `filter_it` also allows in-situ changes to the rows.)
  row types: garr.

- `make_date_and_time_scheme_it`: translates date and/or time strings to date and time schemes which are human-readable numeric representations of dates and times.
  row types: sarr.

- `make_replace_values_it`: uses a user-specified lookup-table for string replacements in the chosen columns.
  row types: sarr.

- `make_reduce_to_numerical_columns_it`: uses `make_date_and_time_scheme_it`, `make_replace_values_it`, `make_strings_lookup_it`, `make_drop_columns_it`, and `make_numeric_it` to reduce the incoming sarr-rows to numeric rows and tries to keep as much information as possible.
  row types: sarr (input), darr (output).

- `make_ascii_assoc_it`: reads ASCII lines from a previous iterator in a kind of sparse CSV format that names explicitly the columns to be set in each row. Missing values are set to a special value given by the user. The results may also be converted to numbers.
  row types for input: sarr.
  row types for output: sarr, darr.

5.6 Other data flow iterators

- `make_wrap_it`: auxiliary iterator that wraps an iterator chain and user data structures. The result will behave the same way as a single iterator. A user-specified destructor function may also be given to avoid memory leaks.
  row types: darr, sarr, garr.

- `assert_of_it`, `make_assert_of_it` (on-the-fly version): used in tests to check whether the rows and the headline produced by the previous iterators are as expected. The consumer `assert_of_it` calls `close_rec_it` in the end if necessary.
  row types: darr, sarr, garr.

- `make_repeat_all_it`: uses `reset_it` internally to repeat the row sequence of the previous iterators n times.
  row types: darr, sarr, garr.

- `make_join_to_prev_it_after_reset_it`: allows to join aggregated values of row groups to each original row (i.e. to each row before the aggregation).
  row types: darr, sarr, garr.

- `diagplot_of_it`, `make_diagplot_of_it`: creates a plot comparing truth values with (mean) predictions.
  row types: darr.

- `recording_of_it`: (Python) puts an iterator's data into a Python recording object that may easily be plotted using Python's Matplotlib library.
  row types: darr

- `make_compared_pairs_it` (planned) : creates pairs of training events for trainings that are supposed to predict rank instead of the absolute value of a quantity.
  row types: darr

6 Branches in iterator chains

An iterator chain may branch at some point into a series of separate branches where the data flows independently. Several applications of this mechanism are conceivable:

1. The data from the incoming iterator may be a heterogeneous mixture in arbitrary order. It is much easier to construct timeseries etc. from more homogeneous series of rows. By branching the iterator chain into a set of branches comprised of the same kinds of iterators and distributing the rows among the branches accordingly, the iterators in the branches are confronted with a more homogeneous context.

2. Many machine learning algorithms give bad results when confronted with a heterogenous mixture of data where the interdependence between the measurements and the value to be predicted is different. They usually give much better results if multiple instances of them are trained, each with a more homogeneous selection of data.

3. Most machine learning algorithms don't give significantly better results when confronted with more training data if they already have copious training cases, so normally a random choice is indicated to save time; but in several parallel trainings on different clusters of data as in item 2 more training cases may be actually used for the benefit of the prediction quality.

4. A cross-validation test using multiple instances of a machine learning algorithm may be done using branches.

5. The branches may be executed in separate threads of execution, i.e. in parallel on different CPUs in order to save time. This only makes sense if the execution of the iterator subchains in the branches is much costlier than the unbranched parts of the iterator chain.

The following iterators handle the branching mechanism:

- `make_branch_it`: branches an iterator chain. Several steps later, all the branches *must* be rejoined in a `make_unbranch_it` iterator.
  row types: darr (so far).

- `make_unbranch_it`: rejoins branches created by `make_branch_it`; pulls the incoming rows of the latter according to its user function through the branches and passes the resulting rows on to the next iterators (see figure 2). The user may pass a function that controls the distribution of the incoming rows among the branches. The same row may be distributed to several branches. row types: darr (so far).

- Instead of consumers, please use the corresponding on-thy-fly iterators, like `make_csv_of_it` instead of `csv_of_it`,
  see section 4.

Not all iterators may be used in branches, yet, see the header-file documentation for details.

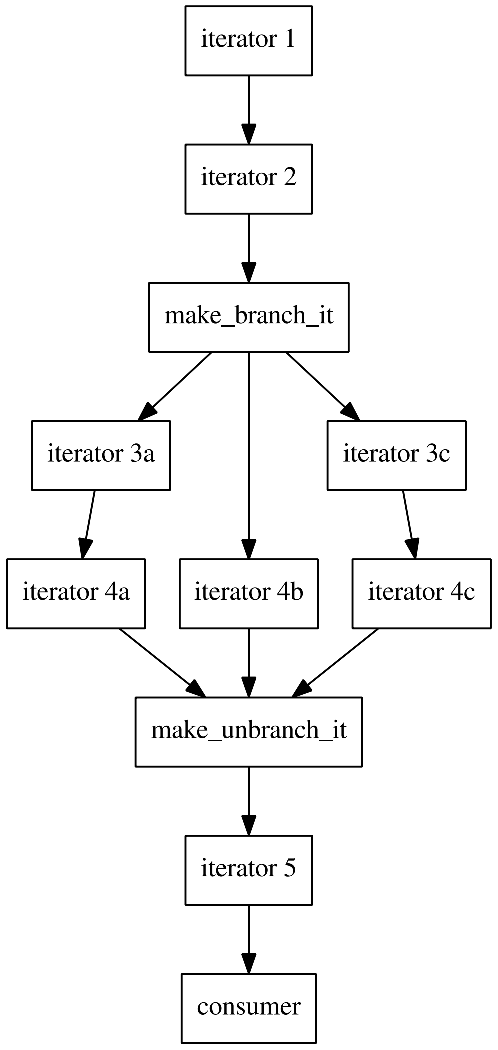


Figure 2: Example of a branched iterator chain.

7 Using iterator chains as functions on data rows or data matrices

It is also possible to use iterator chains as functions acting on passed rows or row groups of values. This way, iterator chains and machine learning code may be used in arbitrary places of user programs.

- `push_darr_row_it`: pushes a new row into a `darr_row_it`.
- `push_dmatrix_it`: pushes a new matrix into a `dmatrix_it`.
- `push_and_pull_it`: allows using an iterator chain (specified by its begin- and end-iterator) as a function on individual rows. The iterator chain must produce exactly one output row for each input row.

- `push_and_pull_it2`: allows using an iterator chain (specified by its begin- and end-iterator) as a function from individual rows to matrices. The iterator chain must produce one or more output row for each input row.
- `push_and_pull_it3`: allows using an iterator chain (specified by its begin- and end-iterator) as a function on matrices. The iterator chain must produce a series of at least one output row for each input matrix.

8 Column-wise and chunkwise access

Some data sources like HDF5 allow column-wise or chunkwise access to the data, i.e. if only a small subset of the data columns is needed – say, only two columns in a comparison – the iteration is proportionally faster. This is not the case for data sources with row-wise data layout like MySQL tables or CSV files, because each row must first be read completely before restricting it to the needed columns.

There is a drawback in pure column-wise access: If all columns are needed, then each of them is read completely into RAM before assembling the rows.

Chunkwise data layout is the best compromise between column-wise and row-wise data layout, because it has the speed advantage of column-wise access without consuming too much memory: Only a single chunk of each column is kept in RAM at any time.

Iterators and consumers that support column-wise access:

- `make_h5_it`: for column-wise read access. The optional argument `input_columns` restricts access to a subset of columns of an HDF5 file.
- `h5_of_it`: for column-wise write access. By the option `fill_existing` only those columns of an existing HDF5 file are overwritten that are found in the previous iterator.
- `make_root_it`: for column-wise read access (planned) . Unfortunately, it is not possible to change ROOT files – only new files may be created – so column-wise write access is impossible.

9 Mechanisms for algorithmic iterators

Algorithmic iterators are processing steps of machine learning algorithms wrapped as iterators. They usually have a *learning mode* (also called training or teacher mode) where connections between the events' measurements and the values to be predicted are learned and an *application mode* (also called expert mode) where the learned connections are applied to new events in order to execute a processing step of a prediction.

The learned connections are saved in a special data structure called `dynvars_t`, see below.

All the algorithmic iterators may be arbitrarily combined with each other and with the data flow iterators above and most of them don't require the data to fit into RAM.

9.1 Dynamic typing (generic values)

Dynamic typing in C is realized by a tagged-union datastructure, i.e. a combination of a tag (of enumeration type in C) and a C-union. This datatype is named `gen_t` (generic). The tag determines which type slot of the union is currently in use, the tag for slot `<x>` being `gt_<x>`.

Arrays of element-type `gen_t` are called `garr`. Matrices of this element-type are called `gmatrix`. Floating-point values are serialized exactly, using a hex-float format if necessary.

The following tags and types are currently supported in generic values:

- **m** for missing values. (The contained number may distinguish different kinds or different reasons for the lack of value.)
- **b** for booleans
- **i** for integers
- **l** for 64-bit integers
- **e** for control codes (unsigned 64-bit integers)
- **f** for flags (unsigned 64-bit integers). They are serialized in hexadecimal format.
- **d** for double-floats
- **v** for void pointers
- **s** for strings, immutable by convention
- **ia** for integer-arrays (iarr)
- **la** for long-arrays (larr)
- **da** for double-float-arrays (darr)
- **pa** for void-pointer-arrays (parr)
- **sa** for arrays of strings (sarr), immutable by convention
- **ca** for mutable strings / string buffers (charr, i.e. bounds-checked character arrays), byte arrays, binary data, images, sounds and other signals. Serialization will be in hexadecimal format, so control and NUL characters are allowed in the middle of the buffer.
- **ga** for generic arrays (garr). This tag enables arbitrary nesting of garrs and building dynamically-typed trees.
- **dm** for double-float-matrices (dmatrix)
- **dda** for arrays of darrs (ddarr)
- **clo** for function closures, i.e. a function pointer, an optional thunk `enclosed` and an optional destructor `delete_enclosed`.

Strings and other nonscalar datastructures are read-write and owned by the generic variable, so e.g. literal strings must be duplicated before putting them into a generic variable or garr or gmatrix.

There is a well-defined **sorting order** of different generic values (see the function `compare_gen` below): First the tags are compared (according to the ordering in the above list) and *only if the tags* *are equal*, the values are compared, for instance missing values are always smaller than integers, integers are always smaller than long integers, longs are always smaller than flags, flags are always smaller than double-floats, double-floats are always smaller than strings and strings are always smaller than integer arrays and so on. Such a definite sorting order is internally necessary for many algorithms, e.g. binary search.

9.1.1 Serialization and Deserialization

Generic values may be translated to and from strings or string-buffers (a.k.a. serialization and deserialization) in the following format:

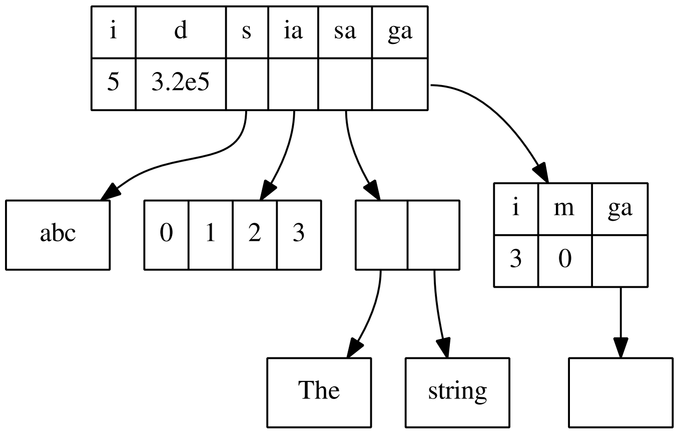
```
<tag>,<string representation of the value>
```

the string representation for arrays being

```
(<value0>,<value1>,...)
```

So at this level, the entries are separated by commas. Strings are always wrapped in double-quotes. (Nested double quotes are escaped by a backslash.) Here are some examples:

- The integer 5 is represented as `i,5`.
- The double-float `3.2e5` is represented as `d,3.2e5`.
- The string ”abc” is represented as `s,"abc"`.
- The integer array from 0 to 3 is represented as `ia,(0,1,2,3)`.
- The string array of the first two words of this sentence is represented as `sa,("The","string")`.
- A generic array with the integer 3, a missing value of kind 0 and an empty generic array as `ga,(i,3,m,0,ga,())`.
- The generic array comprised of all the examples above is represented as `ga,(i,5,d,3.2e5,s,"abc",ia,(0,1,2,3),sa,("The","string"),ga,(i,3,m,0,ga,()))`.



9.1.2 Functions and macros for generic values, garrs and gmatrices

- `G<tag>`: creates a generic value (on the stack) with the given tag and value.
- `G<tag>_`: (only for array tags), creates a generic value (on the stack) with the given tag and elements.
- `G<x>dup`: creates a generic value, duplicating the contained strings first.
- `new_gen`: creates a generic value on the heap.
- `gen_into_buffer`: adds a string representation of a generic value to a buffer.
- `gen_of_string`: creates a generic value (on the stack) out of a string representation.
- `fprint_gen`, `print_gen`: prints a generic value.
- `free_gen_contents`: frees the contents of a generic value if necessary.
- `deep_copy_gen`: deep copy of a generic value.
- `copy_garr`: deep copy of a garr.
- `compare_gen`: compares two generic values. If tags are different, the tags are compared; uses recursion for nested garrs.
- `gens_equal`: Are two generic values equal?
- `I<tag>`: bounds-checked indexing macro for garrs, extracts the value of `<tag>` and checks whether the tag is as expected.
- `II<tag>`: the same for gmatrices.
- `free_garr`: frees a generic array.
- `free_garr_contents`: frees only the contents of a generic array using `free_gen_contents`.
- `free_gmatrix`: frees a generic matrix.

9.2 Data structures and formats for expertise and monitoring data

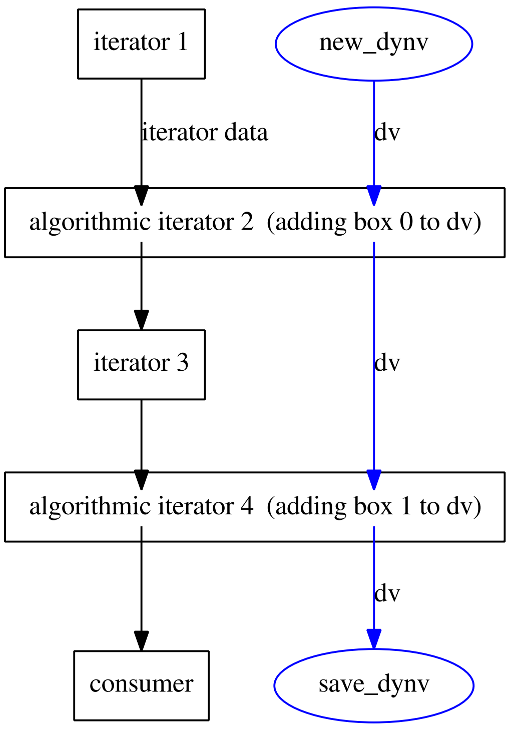
Dynamic variables are saved in a data structure `dynvars_t` comprised of a chain of "boxes", each box corresponding to an algorithmic iterator in the iterator chain. (The other iterators don't need boxes, see figure 3.) The user may add intermediate boxes if desired. A box is a hash table indexed by variable names and carrying generic values. For each box, its name (derived from the corresponding algorithmic iterator or topic) is saved in this hashtable as well.

Some iterators can add boxes to more than one dynvars data structure (see figure 4).

Functions for dynamic variables:

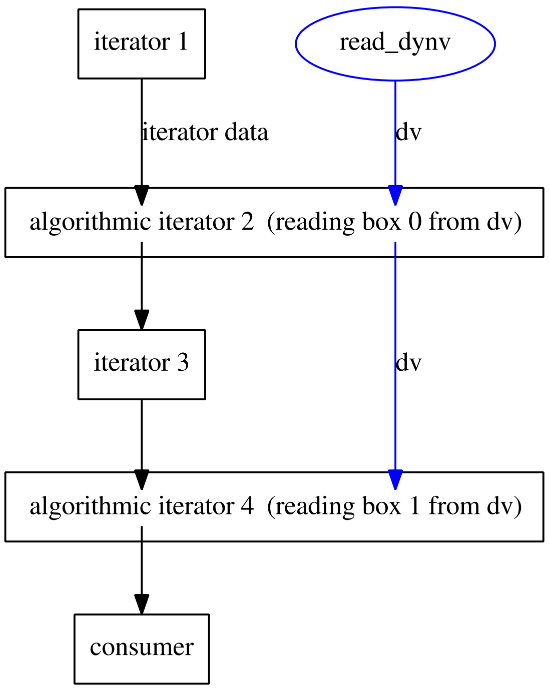
- `new_dynv`: creates a new empty dynvars object.

Learning mode:



iterator 1
new_dynv
iterator data
dv
algorithmic iterator 2 (adding box 0 to dv)
iterator 3
dv
algorithmic iterator 4 (adding box 1 to dv)
dv
consumer
save_dynv

Application mode:



iterator 1
read_dynv
iterator data
dv
algorithmic iterator 2 (reading box 0 from dv)
iterator 3
dv
algorithmic iterator 4 (reading box 1 from dv)
consumer

Figure 3: Example for using dynvars (blue) in an iterator chain with several algorithmic iterators.

- `free_dynv`: frees a dynvars object and the contained data structures.
- `forward_dynv`: moves to the next box in the dynvars object and check its name, creates a new box with the given name if necessary.
- `set_dynv`: sets a dynamic variable for the given key in the current box of a dynvars-object. Prefixes may be added to the key.
- `get_dynv`: gets a dynamic variable for the given key in the current box or previous boxes of a dynvars-object. For nonscalar variables, this is a reference to the value. Prefixes may be added.
- `get_<tag>_dynv`: special case of `get_dynv`, checks whether the tag is as expected.
- `set_flags_dynv`: special case of `set_dynv`, checks whether the tag is `f` and allows to switch individual bits (flags) in the value on or off.
- `set_varflags_dynv`, `get_varflags_dynv`: special case of `set_flags_dynv` and `get_f_dynv` for the key "varflags" which is used to manage variable properties like continuity, existence of missing values etc.
- `set_all_varflags_dynv`: switches on or off given varflags in all columns of an iterator or headline-array matching certain restrictions like "startswith", "endswith" etc.
- `set_re_varflags_dynv`: switches on or off given varflags in all columns of an iterator or headline-array matching a given regular expression.
- `save_csv_dynv`: saves dynvars in a CSV file.
- `load_csv_dynv`: loads dynvars from a CSV file.
- `save_pmml_dynv` (planned) : saves dynvars in an XML file in PMML format.
- `load_pmml_dynv` (planned) : loads dynvars from an XML file in PMML format.
- `save_mysql_dynv` (planned) : saves dynvars in a MySQL table.
- `load_mysql_dynv` (planned) : loads dynvars from a MySQL table.
- `fprint_dynv`, `print_dynv`: prints a dynvars object.
- `new_ref_dynv`: creates a new reference to an existing dynvars object (and to a chosen box in it).

The CSV format of dynamic variables is comprised of lines in the following format, one line for each dynvars entry:

```
<box number>;<entry name>;<string representation of a generic value>
```

(see section 9.1 for the string representation of generic values). At this level, the entries are separated by semicolons.

An entry name is a single identifier (comprised of letters, digits and underscores) or a sequence of identifiers (prefixes, context etc.) separated by commas:

`<name>` or
`<prefix1>,<prefix2>,<name>` etc.

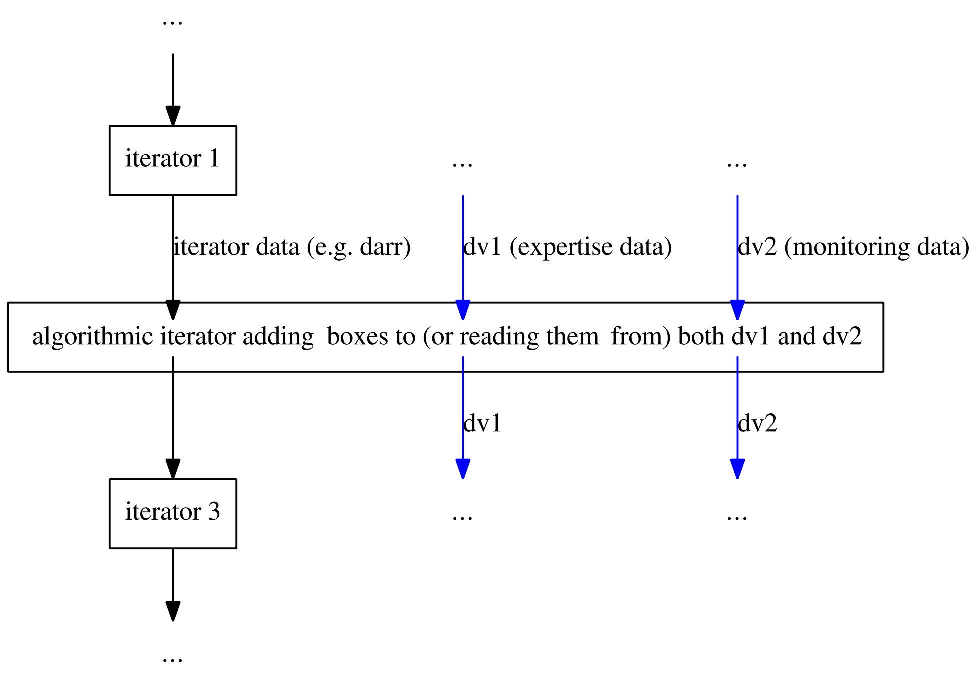

...
iterator 1
...
...
iterator data (e.g. darr)
dv1 (expertise data)
dv2 (monitoring data)
algorithmic iterator adding boxes to (or reading them from) both dv1 and dv2
dv1
dv2
iterator 3
...
...
...

Figure 4: Using more than one dynvars object in an algorithmic iterator, one for expertise data and the other for monitoring data.

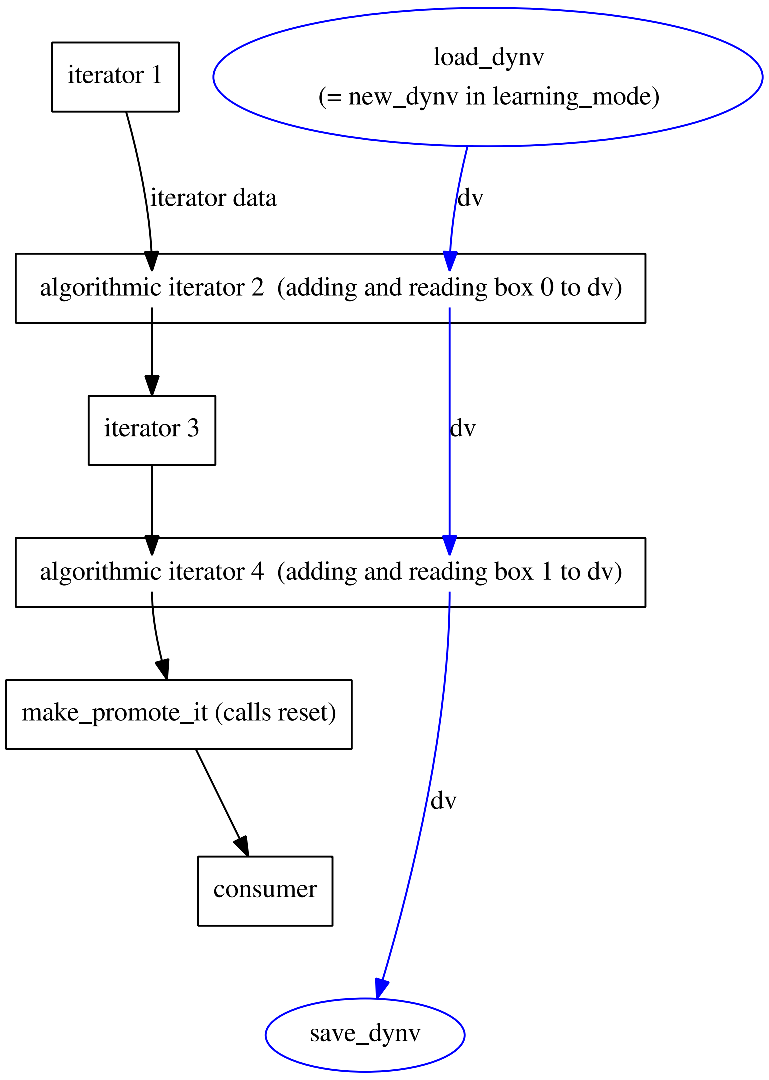
9.3 Basic iterators for learning algorithms

- `make_promote_it`: If the previous iterator is in learning mode, the rows are replaced by dummy rows until the end is reached where the previous iterators have the chance to switch to application mode. (This is internally handled by the method `promote_it`.) Then `reset_it` is called on the previous iterator.

  If the previous iterator is in application mode, the rows are just passed unchanged.

  With `make_promote_it`, the code in figure 3 may be summarized this way:

  Both learning mode and application mode:

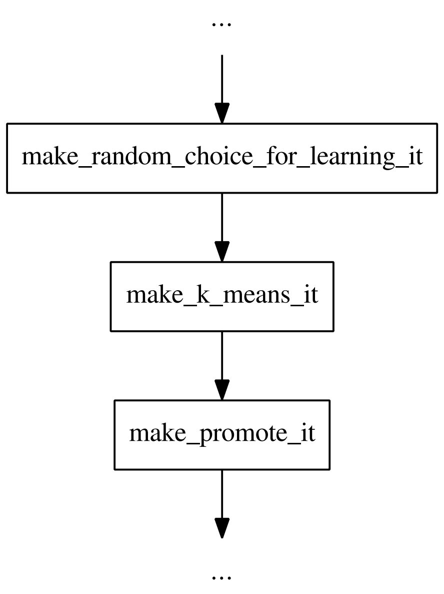


iterator 1
load_dynv
(= new_dynv in learning_mode)
iterator data
dv
algorithmic iterator 2 (adding and reading box 0 to dv)
iterator 3
dv
algorithmic iterator 4 (adding and reading box 1 to dv)
make_promote_it (calls reset)
dv
consumer
save_dynv

- `make_algit_in_situ_it`: general iterator that may be used as a base for writing algorithmic iterators that operate on existing columns.
  row types: darr, sarr (so far).

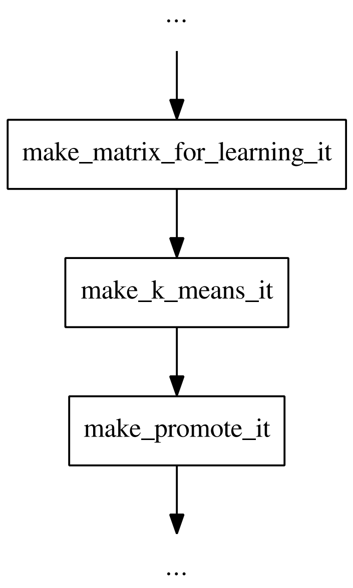
- `make_algit_new_columns_it`: general iterator that may be used as a base for writing algorithmic iterators that create new columns.
  row types: darr, sarr (so far).

- `make_random_choice_for_learning_it`: In learning-mode, a random choice of the incoming rows is presented. In application mode, all rows are presented.
  This iterator is useful for iterators that need random access across all rows only in learning mode. In order to avoid consuming too much memory, only a random choice of the rows is used.
  Put `make_random_choice_for_learning_it` before a group of iterators of this kind and a `make_promote_it` For instance:



  row types: darr, sarr, garr.

- `make_matrix_for_learning_it` (planned) : Only in learning-mode, selected columns of the incoming rows are parked in an internal matrix in RAM.
  This iterator is an optimization for iterators that need many runs over a subset of the columns of the data. The other columns will not be initialized in learning mode.
  Put `make_matrix_for_learning_it` before a group of iterators of this kind and a `make_promote_it`. For instance:



This iterator may be combined with a preceding `make_random_choice_for_learning_it`.
row types: darr, sarr, garr.

- `make_h5_for_learning_it` (planned) : the same as `make_matrix_for_learning_it` except the data is parked in an HDF5 file instead of a matrix in RAM.
  row types: darr.

- `make_shuffled_matrix_for_learning_it` (planned) : similar to `make_matrix_for_learning_it` but the rows are retrieved in a random order (each row begin chosen exactly once per run).
  row types: darr, sarr, garr.

- `make_matrices_learning_appl_it`: reads from two different matrices in learning mode and application mode. The headline must be the same for both matrices.
  row types: darr (so far).

- `make_h5_learning_appl_it` (planned) : read from two different HDF5 files in learning mode and application mode. The headline must be the same in both files.
  row types: darr (so far).

- `make_random_choice_learning_appl_it` (planned) : special case of `make_split_learning_appl_it`. A random choice of the incoming rows with the given probability is passed in learning mode and the rest in application mode.
  row types: darr (so far).

- `make_flatten_it`: In learning mode, this iterator determines bins of equal statistics of chosen columns. In application mode, the column values are optionally transformed into a scale that

makes their histogram flat without changing the order of the values.
row types: darr (so far).

- `make_spread_unordered_classes_it` (planned) : decomposes columns that have an unordered domain of values.
- `binary_matrix_of_it`: transforms the data into a binary matrix that may be used in machine learning algorithms operating on binary data.
  row types: darr (so far).
- `ATA_of_it`: optimized function for the calculation of the matrix product $A^T A$ from the rows of the iterator (A being the data matrix which is not loaded into RAM). If the column averages are all zero, this is the covariance matrix of the columns (except for a factor $1/n$).

10 Neurobayes

One of the goals of the future Neurobayes [4] development is to factor out all processing steps of the Neurobayes code into separate iterators that may be combined with other algorithms or user code. This makes it easier to extend Neurobayes by new ideas and to save execution time by factoring out common calculations of multiple Neurobayes trainings or predictions.

In some cases, mathematically equivalent algorithms have been found.

- `make_neurobayes_it`: in learning mode: starts a Neurobayes training on the data of the input iterator. (This iterator currently loads all data into RAM.) In application mode: fills new columns by Neurobayes predictions requested by a user function.
  row types: darr (so far).
- `make_multi_neurobayes_it`: automatically splits the data according to a user function and presents them to separate Neurobayes trainings. A fuzzy split is also possible.
  If there is some systematic difference between the training and the application data then such a split could aggravate the situation considerably, because the discrepancy could be much larger in the individual trainings.
  row types: darr (so far).
- `make_flatten_it`: see section 9.3.
- `make_purity_fit_it` (planned) : builds histograms of the connection between input variables and the prediction target and smoothes them by spline fits.
- `make_diagfit_it` (planned) : performs Neurobayes' diagonal fit.
- `make_neurobayes_neural_net_it` (planned) : trains and uses Neurobayes' neural net.
- `make_orth_poly_interpolation_it` (planned) : interpolates column values by orthogonal polynomials.
- `make_orth_poly_interpolation_pairs_it` (planned) : interpolates values of selected pairs of columns by orthogonal polynomials.
- `make_threshold_predictions_consistent_it` (planned) : makes the predictions for the probability of surpassing different thresholds of the same continuous or discrete quantity consistent with each other.

11 Other machine learning algorithms

Other algorithms may help in analyzing and clustering the data and in finding good input variables for Neurobayes.

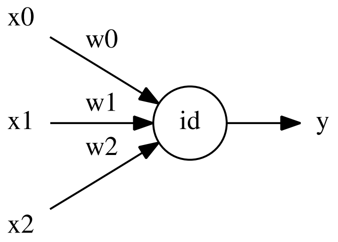
11.1 Special Neural Networks

- `make_oja_rule_it`: implements Oja's weight-update rule of normalized Hebbian learning [5, 6] (unsupervised learning).

  $$\Delta \mathbf{w} = \eta\, y\, (\mathbf{x} - \mathbf{w}\, y)$$

  ($\mathbf{w}$ is the weight vector for calculating the output y from the input vector $\mathbf{x}$. η is the learning rate.) In Hebbian learning neurons that "fire together wire together". The output y is calculated as (id = identity function instead of a sigmoid)

  $$y = \mathbf{x} \cdot \mathbf{w}$$

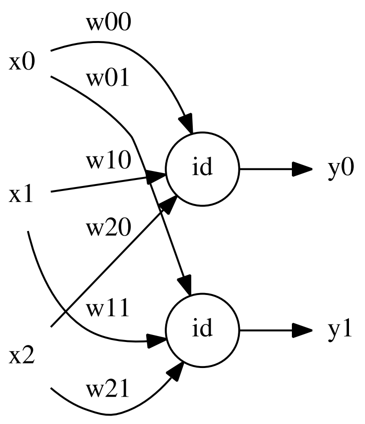
  If the learning rate is positive and the target column was always 1 in the training, the output of a trained network will be high if the input pattern is similar to the patterns already seen in the training.
  row types: darr (so far).

- `make_sanger_rule_it`: implements Sanger's weight-update rule of generalized Hebbian learning [7, 6] (unsupervised learning).

  $$\Delta w_{ij} = \eta\, y_j \left(x_i - \sum_{k=0}^{j} w_{ik} y_k \right)$$

  (w_{ij} is the weight between the ith input and the jth output. x_i is the ith input, y_j is the jth output, η is the learning rate.) The jth output y_j is calculated as (id = identity function instead of a sigmoid)

  $$y_j = \sum_{i=0}^{n-1} x_i w_{ij}$$



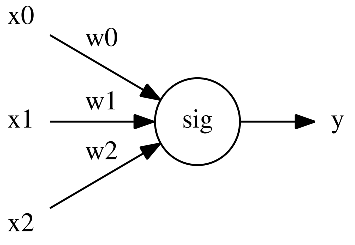
It is similar to Oja's rule but supports multiple targets and adds a Gram-Schmidt orthogonalization step.
This iterator may be used for Principal Component Analysis.
row types: darr (so far).

- `make_perceptron_it`: implements the update rule of a perceptron [8, 6] (supervised learning)

$$\Delta \mathbf{w} = \eta \, (Y - y) \, \mathbf{x}$$

(**w** is the weight vector for predicting the target Y from the input vector **x**. η is the learning rate.) The output y (prediction for Y) is calculated as (sig = sigmoid function, e.g. arctan):

$$\mathbf{y} = \mathrm{sig}(\mathbf{x} \cdot \mathbf{w})$$



row types: darr (so far).

- `make_neural_net_it` (planned) : trains and uses a more general Neural Network.

11.2 Clustering algorithms

- `make_k_means_it` (planned) : k-means clustering algorithm [9]. A chosen subset of the incoming columns (for instance as a result of a dimensional reduction, see `make_sanger_rule_it` and `make_svd_it`) is considered as the coordinate vector in an abstract n-dimensional space.

k center points in this space are chosen in the beginning or passed by the user. In each run, each center is then updated as the centroid of the points nearest to it.
In application mode, a new column is filled with the center number of the center nearest to the current point.
row types: darr (so far).

11.3 Others

- `make_svd_it` (planned) : calculates the singular value decomposition of selected columns and the components of the first few left singular vectors. These are the events' first few eigenfeature coefficients.
- `make_naive_bayes_it` (planned) : Naive-Bayes algorithm.
- `make_support_vector_machine_it` (planned) : support-vector machine algorithm.
- `make_self_organizing_map_it` (planned) : self-organizing map algorithm.
- `make_em_it` (planned) : EM algorithm.
- `make_adaboost_it` (planned) : AdaBoost algorithm.
- `make_k_nearest_neighbors_it` (planned) : k-nearest-neighbors algorithm.

11.4 Text mining algorithms

Some text mining algorithms transform the text into a series of numbers and then use machine learning algorithms like `make_svd_it` to analyze them and make predictions ("latent semantic indexing"). Others operate directly on the texts. Several of these will be added to Neurobayes in the future.

...

12 Components for monitoring, logging and debugging

Monitoring and logging tasks are subdivided into measuring, evaluating and displaying components. By this separation the reusability of the components is improved: Sometimes the user only needs some measurements without any evalution, because the measurements are needed for some algorithm, or he wants to customize the evalution, but not the measurements. In particular the display is usually project-specific and it's nice to be able to change it without having to copy the code for measurement and evalution.

These components consider each row and keep updating data structures in a dynvars object (for dynvars see section 9.2) or react to such changes. The dynvars mechanism is the same as for the expertise data of learning algorithms, but it's usually preferable to use a separate dynvars data structure in order to separate expertises from monitoring data (see figure 4).

Monitoring-dynvars for training data may be saved in a file and later be compared to the same measurements on live data.

12.1 Measuring iterators

These components perform live-updates of measurements for each incoming row.

- `make_histograms_it`: For each incoming row, histograms of the chosen columns are updated. An internal filter may be used in this and the following components. For the bins, the results of the component `make_flatten_it` are expected in the dynvars object.
- `make_histogram_graphs_it` (planned) : For each incoming row, the mean and standard deviation of variable y in the corresponding bin of x is updated for chosen variable pairs (x, y).
- `make_linear_regressions_it` (planned) : updates the linear regression between x and y for chosen variable pairs (x, y).
- `make_deviations_it` (planned) : updates deviations between chosen pairs of columns, e.g. truth and median-prediction, calculated by a user-defined deviation function, for instance the mean absolute deviation (MAD) or the root mean square of the deviation (rms).
- `make_statistics_it` (planned) : updates chosen statistical quantities of chosen columns.

12.2 Evaluating iterators

The evaluating components not necessarily react to each incoming row, because they usually are of a statistical nature. They may rather perform some updates every 5 minutes or every 10000 rows.

- `make_compare_histograms_it` (planned) : compares periodically the histogram measured live by `make_histograms_it` with another histogram measured earlier in the training (and read from the monitoring data file). Statistical deviations up to $r \cdot \sigma$ are tolerated.
- `make_compare_histogram_graphs_it` (planned) : the same for the means and standard deviations determined by a `make_histogram_graphs_it` component.
- `make_compare_linear_regressions_it` (planned) : the same for linear regressions determined by a `make_linear_regressions_it`.
- `make_check_diagonality_it`: Checks that the slope of a linear regression (the diagonality) is 1 within the error margins.
- further components for analyzing data and for checking the **data quality**.

12.3 Displaying iterators

These components display measurements or evaluations. Just as the evaluations, they can be configured to be react only periodically instead of to each incoming row. They report results in logfiles using the function `logprintf` (see below) or create or update plots or HTML pages.

- `make_graph_dv_it` (planned) : updates a plot of a chosen dynamic variable, which may be a measurement or an evalution, on the fly. Plotting and drawing commands are generally specified as universal garrs to be interpreted in a chosen engine for 2D or 3D drawings. Several drawing engines will be supported, including

  - GLE (planned) ,
  - Postscript (planned) ,
  - Qt (planned) ,
  - BMP (planned) ,
  - ROOT (planned) ,
  - Python's matplotlib (planned) ,
  - Javascript and Ajax (planned) ,
  - Flash (planned) ,
  - delegating engines (planned) that send and receive the drawing commands over a socket pair to an engine in another process or on another machine.

- `make_curses_dv_it` (planned) : updates chosen dynvars-values on an ASCII screen using the `curses` library.
- `make_html_dv_it` (planned) : updates chosen dynvars-values on an HTML page.
- `logprintf`: print-function for logfiles for the above components. The filename may be configured by

  1. the global C variable `logfile_destination` set by the user,
  2. otherwise the environment variable `$LOGFILE_DESTINATION`,
  3. otherwise the empty filename "" is taken which means that logging output will be written to `stdout`.

  The output is flushed after each `logprintf`. `lock_file` is used internally; so concurrent processes may print to the same logfile without interference.

12.4 Debugging components

When looking for the source of a problem or a discrepancy the following components may be used in a kind of binary search: Add a check to the middle of the program and find out whether the problem occurs in the first or the second half of the program. Next add a check to the middle of that half and so on until the crucial line of code is found. This process will only take a logarithmic number of steps. In many cases it's of course better to use the GNU debugger `gdb` with breakpoints and watchpoints directly.

- `make_debug_it`: allows a user function to show debugging output about the incoming rows. It has the same interface as `make_map_in_situ_it`.
  row types: darr, sarr, garr.
- `make_dprint_it`: shows a progress indicator while iterating through the rows. A C file pointer for the destination may be given.
  row types: darr, sarr, garr.
- `make_show_groups_it`: prints debug output reporting the beginning of new row groups defined by `checked_equal_columns` or `end_of_group_column`. A C file pointer for the destination and a prefix for the output may be given.
  row types: darr (so far).

- `make_compare_columns_it`: compares the values of each column pair in a list given by the user and prints error messages whenever discrepancies are found. There are several ways to limit the number of error messages printed.
  row types: darr (so far).

- `pos_n_times` ("positive n times"): For the first n times this macro is called in a program, a positive repetition number is returned, otherwise 0. This and the following macros are threadsafe, i.e. each thread uses its own internal count.

- `limprintf`: executed at most n times in order to avoid flooding, uses `pos_n_times` internally. The line number of the call is also printed.

- `pos_n_times_offs`: For the first `n` after `offset` times this macro is called in a program, a positive repetition number is returned, otherwise 0.

- `pos_each_n`: Each nth time this macro is called in a program, a positive repetition number is returned, otherwise 0.

- `pos_at_pow2`: Whenever the logarithm of the internal count surpasses an integer value, the count is returned, otherwise 0 is returned.

- `pow2_printf`: The same as `limprintf` except using `pos_at_pow2` internally. Flooding is avoided by printing whenever the current repetition number is a power of 2.

- `visitcount`: counts and returns the number of visits of a position in the code.

- `statistical_profiler` (planned) : executable program linked to the Unix library `libgdb` that profiles a running program by checking the function on top of its stack at regular time intervals. In the end, a profile ranking of the functions is printed.

13 Live utilities

These are reusable components for live applications, i.e. cases where individual prediction requests have to be handled immediately instead of just iterating over a file or a database query. Due to the nature of live applications, parking the data doesn't make sense in live iterator chains.

13.1 Threads

This library uses POSIX-threads, so genuine multi-core-programming is possible. Mutexes are used in order to protect the data integrity on the lowest level. On a higher level, it is easier and preferable to use the iterator-buffer components presented below.

New threads use their own random-number generators for thread-safety and better reproducibility. These random-number generators are initialized by a new random seed passed to the thread-spawning functions or otherwise to the random state of the parent thread.

- `start_new_joinable_thread`: spawns a new thread executing a user function. The parent thread may later wait for the end of that thread's execution by `join_thread`.

- `join_thread`: waits for the end of a thread started with `start_new_joinable_thread`.

- `start_new_thread`: spawns a detached thread executing a user function.
- `new_mutex`, `free_mutex`: allocates and frees a mutex (thread lock). These functions delegate to the corresponding POSIX-thread functions and send exceptions in the error case.
- `mutex_lock`, `mutex_unlock`: locks and unlocks a mutex (acquires and releases a thread lock). `mutex_unlock` must be put in a try-finally-block in order to make sure that the mutex is unlocked in exceptions as well. These functions delegate to the corresponding POSIX-thread functions and send exceptions in the error case.
- `sleep_microseconds`: microsecond-precise sleep.
- `time_plus_num_seed_fct`: a typical seed function that depends on the present time (including microseconds) and a user-specified number.
- `make_mutex_lock_it`: locks a mutex (acquires a thread lock) before passing each row.
  This may be used in combination with `make_mutex_unlock_it` to wrap a part of an iterator chain that is not threadsafe, e.g. (currently) Neurobayes calls or accesses to the same file in multiple threads (if `make_csv_of_it` is used with the same `FILE*` argument in several threads in order to record live predictions).
  row types: darr, sarr, garr.
- `make_mutex_unlock_it`: unlocks a mutex (releases a thread lock) before passing each row.
  row types: darr, sarr, garr.
- `new_thcond`, `free_thcond`: allocates and frees a thread condition (see the explanation for `thcond_wait` below). These functions delegate to the corresponding POSIX-thread functions and send exceptions in the error case.
- `thcond_wait`, `thcond_signal`, `thcond_broadcast`: If you find yourself writing sleep loops checking over and over again a condition that is under the control of another thread, it's cleaner and much more efficient to use *thread conditions*.
  In `thcond_wait` a thread and the passed mutex are suspended until another thread signals the validity of the condition by `thcond_signal` (sent to one thread waiting for this condition) or `thcond_broadcast` (sent to all threads waiting for this condition). These functions delegate to the corresponding POSIX-thread functions and send exceptions in the error case.
- `thcond_locked_wait`: convenience function that locks the mutex, waits for the condition and unlocks the mutex.

Since the column-indices mechanism (see section 2) internally uses thread-local variables in order to support different headline orders in multiple threads accessing the same column index object, the initialization and the use of the column index object must be put into the same thread function. Thus it's generally not a good idea to build the iterator chain in the main thread and then to call only the consumer of this chain in a secondary thread.

13.2 Parallelization

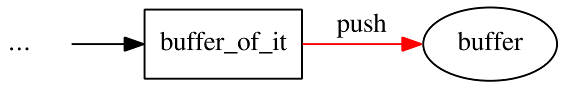
- `start_new_joinable_threads`: starts and returns an array of threads (each executing a user function with an index argument) that may be joined later by `join_thread`, `join_all_threads` or `join_fastest_thread`.

- `join_all_threads`: joins all threads found in an array of threads.
- `join_fastest_thread`: joins the first thread in an array of threads that is finished.
- `parallelize_by_fork`: Fork the current process and create a number of child processes (executing a user function with an index argument) of which at most `n` run simultaneously.
- `start_new_forks`: analogous to `start_new_joinable_threads` but using fork to create new child processes. The list of process ids is returned. The child processes may be joined later by `join_all_forks`.
- `join_all_forks`: analogous to `join_all_threads`; joins all child processes found in an array of process ids.

13.3 Iterator buffers for inter-thread communication

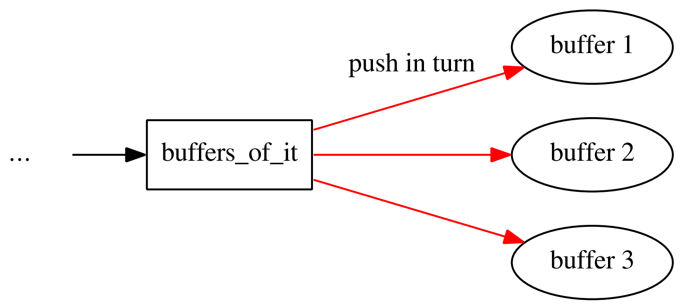
An iterator buffer temporarily saves iterator rows. Numeric, textual or generic rows may be pushed into it in one thread and pulled from it (possibly) in another thread. Iterator buffers also contain a copy of the headline of the pushing iterator. With iterator buffers, it is easy to isolate and protect the coupling (i.e. hazardous write access to the same data structures) between several iterator chains running in separate threads. Each iterator buffers has a capacity (`n_max`, passed in the constructor) that limits the number of rows stored at a given time. If a buffer is full then the pushing thread must wait until another thread retrieves some rows. This is internally handled by thread conditions.

- `buffer_of_it`, `make_buffer_of_it` (on-the-fly version): pushes each iterator row synchronously, i.e. protected by a mutex, into an iterator buffer. The iterator buffer is created if necessary. It includes the headline of the previous iterator.

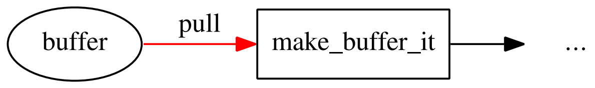
  row types: darr, sarr, garr.

- `make_buffers_of_it`: pushes each iterator row in turn into one of buffers in a series of iterator buffers. The iterator buffers are created if necessary. This may be used to distribute rows to several threads in an ordered manner. The iterator chain will thus be fanned out, e.g. for a series of threads.



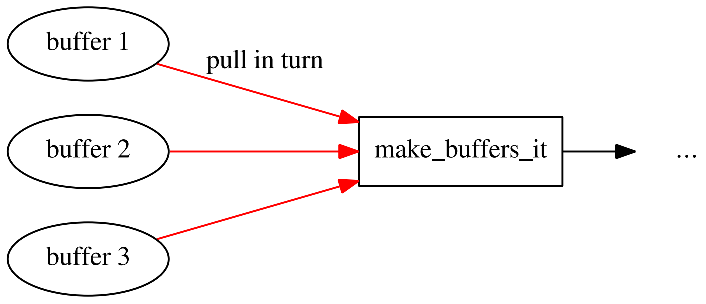
row types: darr, sarr, garr.

- `make_buffer_it`: data source; reads rows synchronously from an iterator buffer with the buffer's headline as its iterator headline.



row types: darr, sarr, garr.

- `make_buffers_it`: the opposite of `make_buffers_of_it`; retrieves the rows in turn from a series of iterator buffers. This may be used to recollect the rows fanned out by `make_buffers_of_it` in an ordered manner.

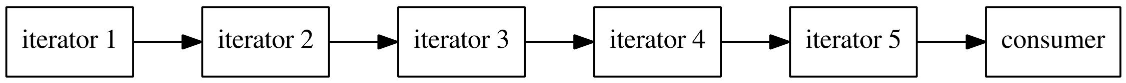


row types: darr, sarr, garr.

- `push_sarr_into_buffer`, `push_string_into_buffer`, `push_darr_into_buffer`, `push_garr_into_buffer`: functions for pushing individual rows directly into a buffer without an iterator. This may also be used to couple iterator chains to libraries that prefer to push data somewhere instead of being pulled, e.g. in software that follows the observer design pattern. Alternatively, the mechanism of section 7 may be used in this case.
row types: darr, sarr, garr.

- `make_buffer_request_it`: A gap in an iterator chain is filled by a call of another thread via a pair of iterator buffers. The latter thread will fill the output buffer with an answer for each row in the input buffers. It must produce exactly one answer for each request (see figure 5). row types: darr, sarr, garr (need not be the same for input and output).

before:



after:

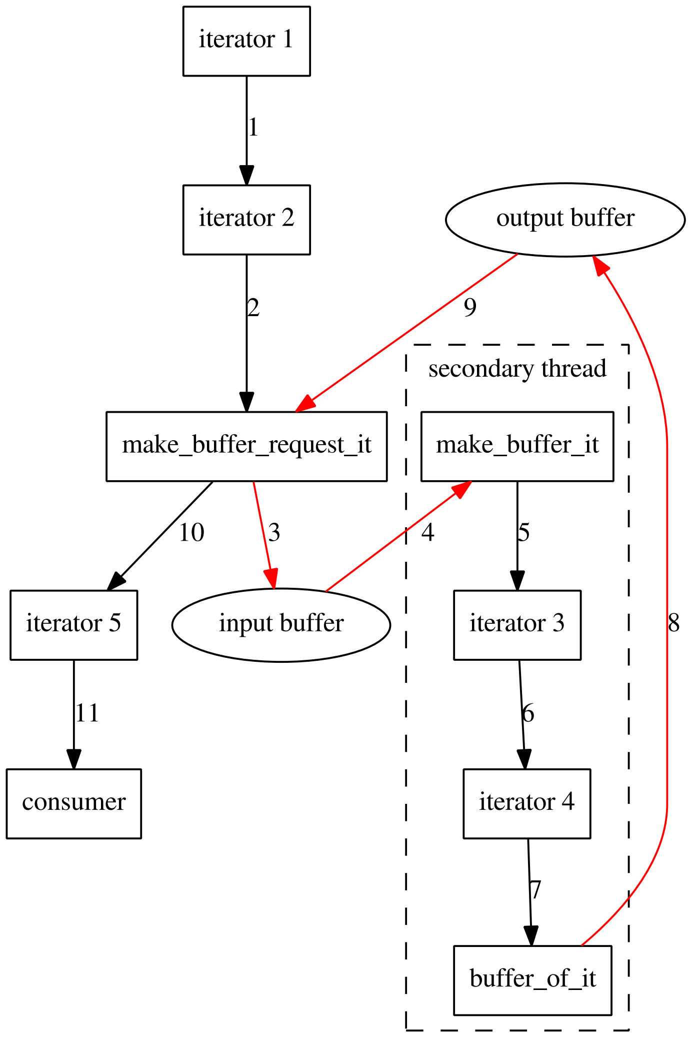


Figure 5: In this example the iterators 3 and 4 are moved to another thread and connected via buffers.

- `make_buffer_requests_it`: the same as `make_buffer_request_it` except row groups grouped

as in make_map_period_it are sent in a bunch of requests to the secondary thread before retrieving the answers. The secondary thread must produce exactly one answer for each input row.
row types: darr (so far).

13.4 Snapshots

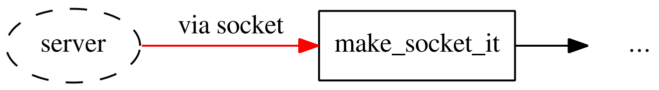
A snapshot is a matrix containing a copy of the rows of the current row group of some iterator chain.

- make_snapshot_of_it, snapshot_of_it: Whenever a new row arrives, it overwrites the corresponding row in the snapshot matrix synchronously (i.e. using a mutex). Another thread may synchronously retrieve rows from the snapshot.
  row types: darr (so far).
- make_join_snapshot_of_it: synchronously joins the right snapshot row corresponding to the incoming row in a way similar to join_ordered_it above.
  row types: darr (so far).

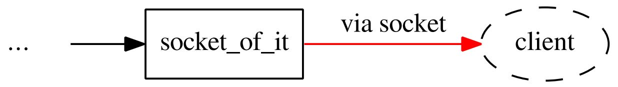
13.5 Socket pairs

Socket pairs may be used for communication between processes on the same or different computers on the intra- or internet. In this interface, socket messages always have a fixed length chosen by the user when creating the socket, i.e. the texts to be sent are always padded to this fixed length. Floating-point values are transmitted exactly, using a hex-float format if necessary.

- socket_server: a general socket server with the plugin functions on_open, on_close and on_request or on_input, on_output. The optional plugin function seed_fct specifies the seed for each connection number. The socket server optionally starts a new thread for accepting connections and always starts a new thread for serving each connection. In on_request the answer is directly calculated from the question whereas in on_input and on_output, the questions are gathered and the answers are retrieved in two separate threads.
- close_socket_server: close a socket server that has been started in a new thread.
- make_socket_it: a client iterator that produces rows received from a socket server.



  row types: sarr, darr (so far).
- socket_of_it: a server iterator offering the incoming rows on a socket to clients in a new thread.



row types: darr, sarr, garr.

- `push_and_pull_in_socket_server_it`: a socket server that lets an iterator chain act on the input rows received as socket requests from the client and returns the output rows to the client. Each connection gets its own iterator chain which is created by a user function. The iterator chain must produce exactly one row for each input row.
  Alternatively, two iterator chains may be given. They correspond to the arguments `on_input` and `on_ouput` in `socket_server` above.
  row types: darr, sarr, garr.

- `make_socket_request_it`: the equivalent of `push_and_pull_in_socket_server_it` on the client side: A gap in an iterator chain is filled by a call to a socket server which acts on another iterator chain.
  This way, a part of an iterator chain may easily be moved to another process (on the same or another machine), for instance because this part needs access to some restricted data or tool which is only available on a particular machine (see figure 6).
  row types: darr, sarr, garr (need not be the same for input and output).

- `make_socket_requests_it`: the same as `make_socket_request_it` except row groups grouped as in `make_map_period_it` are sent in a bunch of requests to the server before retrieving the answers. The socket server must produce exactly one answer for each request.
  row types: darr (so far).

13.6 Web-GUI utilities and XMLHttpRequests

These are components for writing Web-GUIs.

XMLHttpRequests are the standard form of communication between client and server for active Webpages using Ajax (Javascript) or Flash. XMLHttpRequests avoid the slow response of traditional CGI scripts which are forced to rebuild the whole Webpage and restart the server for each request.

- `read_cgi_parameters` (planned) : reads CGI parameters and returns them as a garr of keys and values. Upload parameters are returned as C file descriptors.

- `read_cgi_cookies` (planned) : reads Web Cookies in a CGI script and returns them as a garr of keys and values.

- `print_cgi_answer_preamble` (planned) : prints the HTML content-type, charset and the doctype preamble needed for CGI answers to `stdout`. Optionally, Web Cookies may be added as a garr argument. The rest of the CGI answer which may include Javascript code and references to Flash code is to be printed to `stdout`. Client XMLHttpRequests should be handled in a separate server program using the component `push_and_pull_in_xml_http_server_it` (see below).

before:

after:

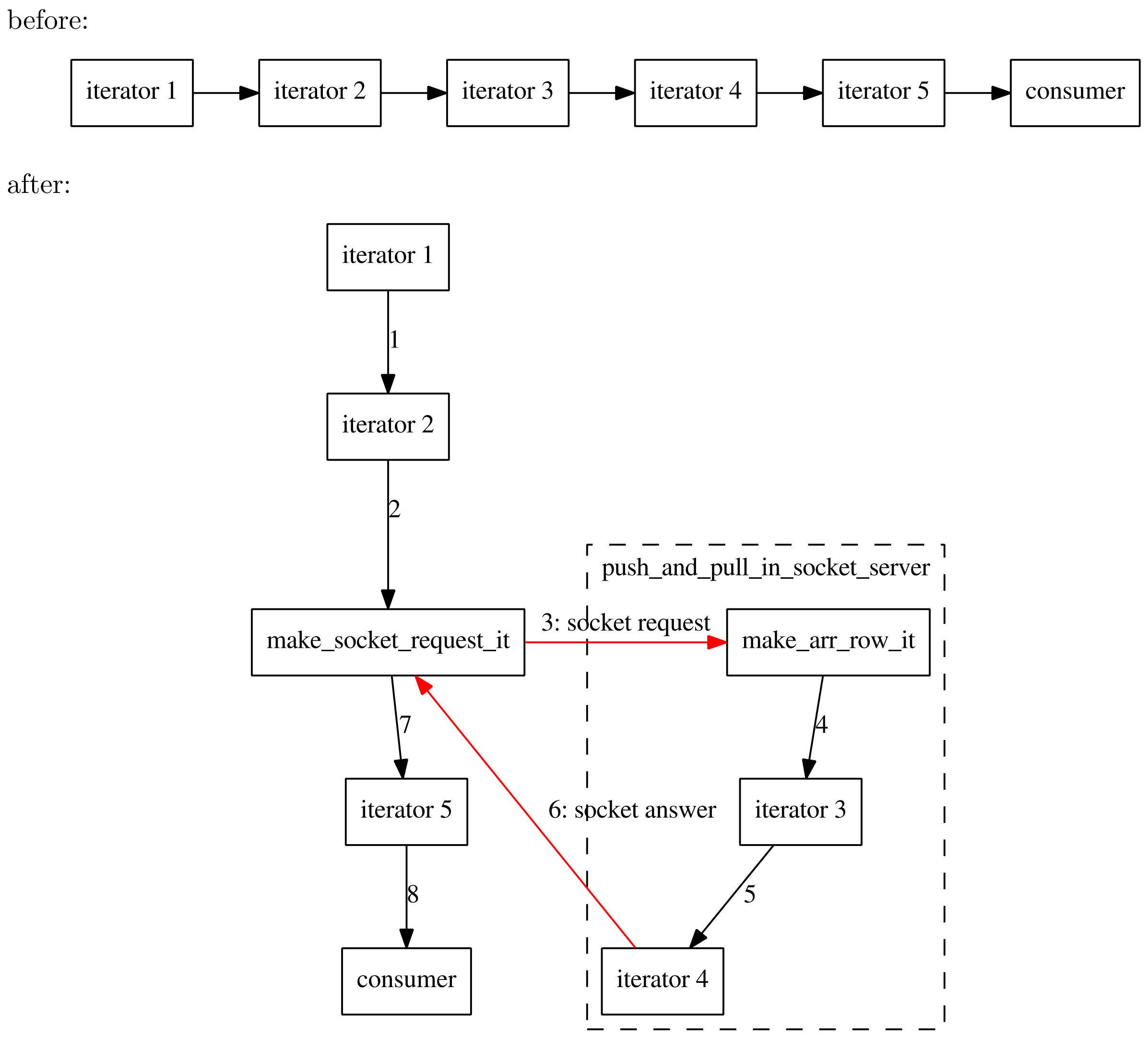


Figure 6: In this example the iterators 3 and 4 are moved to a socket server.

- `push_and_pull_in_xml_http_server_it` (planned) : an XMLHttpRequest server that lets an iterator chain act on the input rows received as XMLHttpRequests from a Webpage client and returns the output rows to the client (just like `push_and_pull_in_socket_server_it` for socket servers).
  Each connection gets its own iterator chain which is created by a user function. The iterator chain must produce exactly one row for each input row. The request (and the answer) may automatically be converted to (and from) a numeric or generic array.
  row types: darr, sarr, garr.
- `make_xml_http_request_it` (planned) : fills a gap in an iterator chain by XMLHttpRequests to a Webpage client (just like `make_socket_request_it` and `make_buffer_request_it`).

The request (and the answer) may automatically be converted from (and to) a numeric or generic array.
row types: darr, sarr, garr.

- an Ajax and Flash library (planned) for sending and receiving garrs as XMLHttpRequests and for displaying data in plots, diagrams and tables.

13.7 Other components for live applications

- `make_backup_for_interrupt_it`: If live programs are interrupted, the iterator row history needed for timeseries, interpolation etc. is lost. This iterator fills this need by saving rows on the fly in a file. If the program is restarted and the option `was_interrupted` is set to `true`, the old rows in the backup-file are offered before the new live rows received from the previous iterator.
  row types: darr (so far).
- `make_dummy_rows_for_groups_it`: adds dummy rows to the start of each group of rows. This is sometimes needed to support the handling of row groups in the live case.
  row types: darr.
- `acquire_debug_mutex`: lock the global mutex for debugging purposes, e.g. in order to avoid intermingling of debugging outputs.
- `release_debug_mutex`: unlock the global mutex for debugging purposes.

14 Performance optimizations

14.1 Sorting tricks

Even though the sorting components are highly optimized, resorting data is often the slowest part of an iterator chain. So it's usually more important to optimize the need for and the way of sorting instead of other parts of the application.

- Please consider dropping columns that are unnecessary for later processing (”passive columns”) before sorting by using the mechanism in section 8 or the iterators
  `make_drop_columns_it` or `make_column_sel_it`. The last step of the sorting algorithm (i.e. the application of the permutation vector) will be proportionally faster with fewer columns.
- `make_resort_groups_it` is faster than `make_resort_it` if applicable.
- If the sort is used for a group-by-aggregation, it's sometimes faster to perform the aggregation in two steps.
  Say, the data has columns (k_1, k_2, k_3, a) and is sorted by the column series (k_1, k_2) and an aggregation of a for each k_2 value is desired. Then instead of resorting all data by k_2 and then aggregating over a for each k_2 it's sometimes faster to aggregate the original data over a for each (k_1, k_2), then resort the much smaller result by k_2 and finally aggregate over a for each k_2.
  If the original data is only sorted by k_1 instead of (k_1, k_2), `make_resort_groups_it` could be used to extend the sorting.

- If make_join_to_prev_it_after_reset_it is used for a group-by-aggregation, it's sometimes faster to sort the smaller table twice instead of the complete data once.
  Say, the data has columns $(k_1, k_2, a, p_1, p_2, \ldots\ldots)$ with the passive columns $p_1, p_2, \ldots$ and is sorted by k_1 and an aggregation of a for each k_2 is desired. For the aggregation, the columns should be restricted to (k_1, k_2, a) (passive columns removed). Instead of resorting the original data by k_2 before the aggregation, it's faster to sort the data without the passive columns by k_2, then aggregate over k_2, then resort the much smaller aggregation by k_1 and finally use make_join_to_prev_it_after_reset_it to join the result to the unchanged original data.

15 Links to further documentation and examples

Further documentation may be found in the C header files of the libraries
(all found in $NEUROBAYES/include):

- dsa_basics.h: fundamental mechanisms, numeric utilities.
- dsa_tests.h: utilities for unit tests.
- dsa_string_utils.h: string and string buffer utilities.
- dsa_arrays.h: safe and expandable arrays and matrices, utilities for nonscalar numericals.
- dsa_statistics: statistics functions.
- dsa_hash_tables.h: hash tables in C.
- dsa_system.h: file system utilities, time and date functions, miscellaneous system utilities.
- dsa_generics.h: generic values (tagged union) and dynamic variables.
- dsa_iterators.h: iterator type, method and utilities, fundamental iterators.
- dsa_data_flow.h: non-fundamental data-flow iterators.
- dsa_linear_algrebra.h: Linear Algebra functions, Matlab-like functions.
- dsa_plots.h: plotprintf and special plot functions.
- dsa_colindices.h: column indices utilities.
- dsa.h: all of the above: fundamental data structures and algorithms.
- hdf5_utils.h: code for using the HDF5 format.
- root_utils.h: code for using the ROOT format.
- pcrec.h: code for using regular expressions in C.
- mysql_utils.h: code for using MySQL.
- prediction_scheme.h: iterator code for Neurobayes and Plots.

- pyembed.h and pyembed.py: utilities for embedding a Python interpreter in C code and for easy use of C functions in Python and of Python functions in C; includes also make_socket_it and socket_of_it.
- algits.h: algorithmic iterators.
- timeseries.h: timeseries algorithms and GSL-dependent code (GNU scientific library).
- live_utils.h: live utilities
- monlog.h: monitoring, logging and debugging utilities
- excel_utils.h: Excel utilities
- sqlite_utils.h: Sqlite utilities
- dsa_branches.h: iterators for branches in iterator chains

The Python wrappers for these libraries are the modules pydsa, pyhdf5_utils, pyroot_utils, pymysql_utils. and pygsl_utils. There is also pympl_utils which contains some Matplotlib utilities. All iterators wrapped for Python pass the data without copying across the language barrier (forth and back).

15.1 Unit tests as examples

All iterators are extensively tested in unit tests. Tests are usually written before implementation to experiment and find a good interface from the perspective of the user. Tests also serve as usage examples which are often much easier to understand than an abstract documentation.

Neurobayes' unit tests are found in the following files
(all in subdirectories of $NEUROBAYES/examples):

- iterator_tests.c: unit tests for fundamental and data flow iterators (functions mostly from dsa_iterators.h and dsa_data_flow.h).
- safe_array_tests.c: unit tests for safe arrays, matrices and other fundamental data structures in C (functions and macros mostly from dsa_arrays.h)
- matlab_like_tests.c: unit tests for functions on matrices and vectors analogical to Matlab (functions from dsa_linear_algebra.h).
- mysql_example.c: unit tests for MySQL utilities (functions from mysql.h).
- regex_tests.c: unit tests for regular expression utilities (functions from pcre.h).
- string_utils_tests.c: unit tests for string utilities (functions from dsa_string_utils.h and dsa_arrays.h).
- hash_table_tests.c: unit tests for hash tables (functions from dsa_hash_tables.h and dsa_arrays.h).
- stats_tests.c: unit tests for statistics functions (functions from dsa_statistics.h).
- parallelization.c: test for parallelization by fork (functions from prediction_utils.h).

- `clustering.c`: tests for clustering algorithms (functions from `dsa_linear_algebra.h`).
- `date_time_tests.c`: tests for date and time functions (functions from `dsa_system.h`).
- `exception_tests.c`: tests for exception handling in C (functions and macros from `dsa_basics.h`).
- `unix_shell_examples.c`: tests for Unix-shell programming with iterators.
- `column_indices_tests.c1`: tests for the handling of column indices (functions and macros from `dsa_colindices.h` and `codegen2.py`).
- `timeseries_tests.c`: tests for timeseries functions and gsl utilities (functions from `timeseries.h`).
- `generic_tests.c`: tests for generics, dynamic variables and some algorithmic iterators.
- `monlog_tests.c`: tests for monitoring and logging utilities.
- `algits_tests.c`: tests for algorithmic iterators (functions from `algits.h`).
- `live_utils_tests.c`: tests for live utilities.
- `branch_tests.c`: tests for branches of iterator chains: branches, switches and column-splits.

Unit tests for Python code (in `$NEUROBAYES/examples/python`):

- `pydsa_tests.py`: tests for Python wrappers of iterators (functions from the modules `pydsa`, `pyhdf5_utils`, `pymysql_utils`, `pyroot_utils`).
- `python_embedding_test.c` and `python_embedding_test.py`: tests for embedding the Python interpreter in C (functions and macros from `pyembed.h` and `dsa.h`).

16 Installation

The environment variable `$NEUROBAYES` must be set to the directory of the Neurobayes installation. Please include `$NEUROBAYES/lib` in your `$LD_LIBRARY_PATH` and your `$PYTHONPATH` and `$NEUROBAYES/external` in your `$PATH`. Please avoid inconsistent settings of these variables!

In order to avoid inconsistent settings, it's preferable to use

```
source <Neurobayes-dir>/setup_neurobayes.sh
```

for setting these variables in the shell (or in its initialization file `.bashrc` or `.cshrc` etc.).

The most recent version of this document is always found in `$NEUROBAYES/doc/neurobayes_iterators.pdf`.

17 Frequently asked questions

- How to append new rows (produced by some iterator) to an existing data sink? – This is currently possible for `csv_of_it`, `mysql_of_it` and `h5_of_it` by the option `append = true`, see the test "`h5_of_it` with or without append = true" in `iterator_tests.c`.
  `root_of_it` can't support this, because the ROOT library doesn't allow to change ROOT files.

A Appendix: Software architecture of the Neurobayes-tools (Bottom-up design)

- **Layer 1** (lowest layer): fundamental mechanisms and data structures (keyword argument handling in C, exception-handling in C; bounds-checked and expandable arrays and matrices of different types in C, Matlab-like functions; string-buffers; hash-tables for C; regular expressions in C); convenience functions for system calls (date and time functions, file handling etc.).

  Files: `dsa.{h,c}`, `dsa_basics.h`, `dsa_tests.h`, `dsa_string_utils.h`, `dsa_arrays.h`, `dsa_statistics.h`, `dsa_hash_tables.h`, `dsa_system.h`, `dsa_linear_algebra.h`, `dsa_plots.h`, `dsa_colindices.h`.
  Tests/examples: `safe_array_tests.c`, `date_time_tests.c`, `stats_tests.c`, `hash_table_tests.c`, `matlab_like_tests.c`, `exception_tests.c`, `regex_tests.c`, `string_utils_tests.c` (all tests are found in subdirectories of `$NEUROBAYES/examples`).
  State: fairly complete, some cleanup needed.

- **Layer 2**: HDF5 utilities and codegen2-backend, i.e. handling of column indices, code-generator for column indices and enums (`cg_declare_columns`, `enum_to_string` and `string_to_enum`), `nb`-macros; efficient access to data in HDF5 files (memory-buffers, column-wise and chunk-wise access).

  Files: `dsa.{h,c}`, `hdf5_utils.{h,c}`, `codegen2.py`.
  Tests/examples: `examples/hdf5/*` in several programming languages; `column_indices_tests.c`, `examples/trainings/*` – training examples using codegen2.
  State: fairly complete, cleanup needed, string handling in HDF5 files needed.

- **Layer 3**: iterator data structure; non-learning iterators (see sections 2 to 7), i.e. iterators without mode and without expertises or monitoring data; data sources, data sinks, filters, maps, data-flow handling analogous to SQL functionality; data-flow handling of timeseries and more complex data-shuffling; on-the-fly iterators; parking iterators; `it_type_t` (for row types) and converting iterators; using iterator chains as functions; foreign-language interfaces for our libraries: Python-interface to iterators and C data structures, embedding Python in C-programs and vice versa.

  Files: `dsa.h`, `dsa_iterators.h`, `dsa_data_flow.h`, `dsa_branches.h` `hdf5_utils.h`, `root_utils.h`, `mysql_utils.h`, `pcrec.h`, `excel_utils.h`,
  `sqlite_utils.h`.

Tests/examples: `iterator_tests.c`, `mysql_example`, `regex_tests.c`, `unix_shell_examples.c`, `branch_tests.c`, `excel_utils_tests.c`, `sqlite_utils_tests.c`.
Python-interface: `pydsa.py`, `pyhdf5_utils.py`, `pymysql_utils.py`, `pyroot_utils.py`, `pydsa_tests.py`, `python_embedding_test.c`.
State: rich functionality implemented, but many ideas for new iterators not realized, yet; some further foreign-language interfaces and converters from and to other data sources may be needed.

- **Layer 4**: utilities for live applications (see section 13); threads, mutexes, thread conditions, microsecond-precise sleep, iterator buffers, sockets, socket-iterator and consumer, socket-server acting on an iterator chain, socket-client-iterator; parallelization utilities; thread-safety of our libraries; a Web-GUI library.

  Files: `live_utils.{h,c}`.
  Tests/examples: `live_utils_tests.c`.
  State: fairly complete except for the Web-GUI library; thread-safety of Fortran-code is still a problem.

- **Layer 5**: dynamic typing and data structures for the expertise data of learning iterators and the monitoring data in monitoring iterators (see section 9.1). tagged-union data structure `gen_t`, dynamic variable data structure `dynvars_t`, conversion of `gen_t` to and from strings, basic functions for `gen_t` (comparison, deep copy, ...), persistence for dynvars (csv, xml), handling of dynvars in iterators.

  Files: `dsa.{h,c}`, `dsa_generics.h`, .
  Tests/examples: `generic_tests.c`.
  Sate: fairly complete, maybe XML format and some other convenience functions needed.

- **Layer 6**: Neurobayes, other learning iterators and monitoring iterators (see sections 9 to 12). These algorithmic iterators have a mode and/or access to dynvars. `promote_it`, prediction algorithms, clustering algorithms, simple text algorithms; iterators that perform measurements on the data, evaluate the measurements or display the evaluations on the fly.

  Files: `dsa.{h,c}`, `algits.{h,c}`, `monlog.{h,c}`.
  Tests/examples: `generic_tests.c`, `algits_tests.c`, `monlog_tests.c`.
  State: some algorithms implemented, design fairly complete, much interesting work to do.

- **Layer 7**: garr-datatype and algorithms. garrs are a very general datatype based on the tagged-union data structure above. garrs are analogous to s-expressions in Lisp but based on expandable arrays instead of linked lists and thus more efficient in many applications. They are adequate for more complex algorithms like those used in AI and they will be used as the lingua franca of data structures here.
  garrs in fundamental iterators, persistence for garrs, interpreter of garr-expressions (a.k.a. Lisp-interpreter), sending garrs over a socket; using garr-interpreters for GUIs and for efficient device- and library-independent drawing.

Files: `dsa.{h,c}`, `gen_utils.h`.
Tests/examples: `generic_tests.c`, `live_utils_tests.c`.
State: fundamentals established; garr-interpreter and garr-GUI designed.

- **Layer 8**: garr-iterators for natural-language algorithms and other AI topics. They will be based on the data structures of Layer 7.
State: planned, some software design ideas established.

- **Layer 9**: higher-order learning iterators (”learn-learners”), i.e. learning algorithms that learn how to choose and configure other iterators or iterator chains.
State: planned, some software design ideas established.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns. Elements of Reusable Object-Oriented Software*, Reading, Mass, Addison-Wesley.

[2] http://www.hdfgroup.org

[3] http://root.cern.ch

[4] Feindt, Michael (2004). ”A Neural Bayesian Estimator for Conditional Probability Densities”. arXiv:physics/0402093v1 [physics.data-an]

[5] Oja, Erkki (1982). ”Simplified neuron model as a principal component analyzer”. *Journal of Mathematical Biology* **15** (3)

[6] Haykin, Simon (1998). *Neural Networks: A Comprehensive Foundation* (2 ed.). Prentice Hall.

[7] Sanger, Terence D. (1989). ”Optimal unsupervised learning in a single-layer linear feedforward neural network”. *Neural Networks* **2** (6)

[8] Rosenblatt, Frank (1957), The Perceptron – a perceiving and recognizing automaton. Report 85-460-1, Cornell Aeronautical Laboratory.

[9] MacQueen, J. B. (1967). ”Some Methods for classification and Analysis of Multivariate Observations”. **1.** Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability. University of California Press. pp. 281-297.