mist

Andrew Banman

Mar 22, 2022

CONTENTS

CHAPTER

ONE

MIST

MIST is a **M**ultivariable **I**nformation Theory-based dependence **S**earch **T**ool. The Mist library computes entropy-based measures that detect functional dependencies between variables. Mist provides the **libmist** library and **mistcli** Linux command line tool.

- Mist source is hosted on Github.
- Mist for Python is available on PyPi.
- Mist documentation is hosted on ReadTheDocs.

1.1 Background

A biological system is intrinsically complex and can be viewed as a large set of components, variables, and attributes that store and transmit information from one another. This information depends on how each component interacts with, and is related to, other components of the system. Handling the problem of representing and measuring the information is the goal of Mist.

A central question of this problem is: How can we fully describe the joint probability density of the N variables that define the system? Characterization of the joint probability distribution is at the heart of describing the mathematical dependency among the variables. Mist provides a number of tools that are useful in the pursuit for the description and quantitation of dependences in complex biological systems.

A function between variables defines a deterministic relationship between them, a dependency; it can be as simple as *if X then Y* or something more complicated involving many variables. Thus, a functional dependency among variables implies the existence of a function. See [Galas2014]. Here we focus on the task of finding a functional dependency without concerning ourselves with the nature of the underlying function.

Mist is designed to quickly find functional dependencies among many variables. It uses model-free Information Theory measures based on entropy to compute the strength of the dependence. Mist allows us to detect functional dependencies for any function, involving any number of variables, limited only by processing capabilities and statistical power. This makes Mist a great tool for paring down a large set of variables into an interesting subset of dependencies, which may then be studied by other methods. This may be seen as compression of data by identifying redundant variables.

1.2 Quick Start

The easiest way to run Mist is by using the **libmist** Python module. The following minimal example sets up an exhaustive search for dependencies between two variables, estimated with the default measurement

```
import libmist
search = libmist.Search()
search.load_file('/path/to/data.csv')
search.outfile = '/dev/stdout'
search.start()
```

There are numerous functions to configure Mist – below are some of the most important. For a full list see Mist documentation and consult the User Guide.

```
search.load_ndarray()     # load data from a Python.Numpy.ndarray (see docs for␣
↪restrictions)
search.tuple_size         # set the number of variables in each tuple
search.measure            # set the Information Theory Measure
search.threads            # set the number of computration threads
```

This Python syntax is virtually identical to the C++ code you would write for a program using the Mist library, as you can see in the examples directory.

1.3 Installation

1.3.1 Docker

Mist can be built into a Docker image with the included docker file

```
cd /path/to/mist
docker image build . -t mist
docker run --rm -v ./:/mist mist
```

The default command builds the Mist python module, which can then be run in an interactive session or with python script, e.g.

```
docker run --it --rm -v ./:/mist mist python3
```

1.3.2 mist

These packages are required to build libmist and mistcli:

- CMake (minimum version 3.5)
- Boost (minimum version 1.58.0)

Run *cmake* in out-of-tree build directory:

```
mkdir /path/to/build
cd /path/to/build
cmake /path/to/mist
make install
```

1.3.3 mist python library

Use pip package manager to install libmist:

```
pip install libmist
```

Or build and install from source.

Additional build requirements:

- Python development packages (python3-dev or python-dev).
- Boost Python and Numpy components. For Boost newer than 1.63 use the integrated Boost.Numpy (libboost-numpy) package. For earlier versions install ndarray/Boost.Numpy.

Run *cmake* with *BuildPython* set to *ON*:

```
mkdir /path/to/build
cd /path/to/build
cmake -DBuildPython:BOOL=ON /path/to/mist
make install
```

Note: both the mist and ndarray/Boost.numpy builds use the default python version installed on the system. To use a different python version, change the FindPythonInterp, FindPythonLibs, and FindNumpy invocations in both packages to use the same python version.

1.3.4 Documentation

Additional Requirements

- Doxygen
- Sphinx
- Breathe
- sphinx_rtd_theme

Run *cmake* with *BuildDoc* set to *ON*:

```
mkdir /path/to/build
cd /path/to/build
cmake -DBuildDoc:BOOL=ON /path/to/mist
make Sphinx
```

And then run the build as above.

1.4 For Developers

This project follows the Pitchfork Layout. Namespaces are encapsulated in separate directories. Any physical unit must only include headers within its namespace, the root namespace (core), or interface headers in other namespaces. The build system discourages violations by making it awkward to link objects across namespaces.

Documentation for this project is dynamically generated with Doxygen and Sphinx. Comments in the source following Javadoc style are included in the docs. Non-documented comments, e.g. implementation notes, developer advice, etc. follow standard C++ comment style.

The included `.clang-format` file defines the code format, and it can should applied with the `tools/format.sh` script.

1.5 Credits

Mist is written by Andrew Banman. It is based on software written by Nikita Sakhanenko. The ideas behind entropy-based functional dependency come from Information Theory research by David Galas, Nikita Sakhanenko, and James Kunert.

For copyright information see the LICENSE.txt file included with the source.

1.6 References

CHAPTER

TWO

USER GUIDE

This guide explains how to prepare data for Mist, set up and execute a search, and tune the algorithm for best performance. The basic steps are:

1. Prepare the data
2. Select an appropriate IT Measure
3. Define the search space
4. Compute

Here we assume that we have a set of variables representing the components of a system we study and a set of samples representing multiple measurements of these variables. So, input data is represented by a matrix, where each row is a variable and each row is a sample, e.g., a measurement or subject. A variable tuple is a small combination of variables. The set of all variable tuples is the search space. Mist efficiently traverses the search space and computes the Information Theory measure for each tuple, which in turn allows us to estimate the strength of the dependence among variables in the tuple.

The procedure of defining a search space and computing the IT measure for each tuple in the space is simply called a search. Mist uses a parallel algorithm to divide the search among computing threads. The algorithm can be tuned to improve performance for different kinds of searches.

2.1 Run modes

There are three ways to run Mist. They all use the same Mist C++ library.

2.1.1 Python Module

The libmist Python module is the recommended way to run Mist searches. All of the examples in this guide use the Python module.

1. Download libmist on PyPi to use the python module.
2. Import the libmist module.

```
import libmist
```

All of the classes needed to execute searches are extended to the Python module. For custom applications that need the full API, use the C++ library directly.

2.1.2 mistcli

Mist provides a statically-compiled Linux command line tool called **mistcli**. It includes enough Mist features to run basic searches. This tool is a good option if Python is unavailable or the system is too old to run with standard libraries.

1. Download mistcli from the release page.
2. Run on recent Linux system

Consult the help output for mistcli for instructions.

```
mistcli -h
```

2.1.3 C++ Library

Application developers can develop new programs with the Mist API in C++. The mistcli program is the reference example.

1. Download libmist from the release page or compile from source.
2. Install development headers
3. Link program against libmist

A good procedure is to compile and install from source:

```
git clone https://github.com/andbanman/mist
mkdir mist/build
cd mist/build
cmake ../
make install
```

And then add the appropriate linker flags, e.g.

```
-std=c++14 -llibmist.so
```

2.2 Prepare the Data

Data should be prepared to meet these requirements:

- Arranged as *NxM* matrix of 8bit signed integer values, typically with each row a variable.
- Continuous variables discretized into non-negative integer bins (for best performance, bins should be contiguous and start at 0).
- Missing values represented by a negative integer.

Data can be parsed in row-major (the default, preferred) or column-major order. In row-major order each row is a variable; in column-major order each column is a variable.

Data can be read from a CSV file, and the parse order is set explicitly

```
import libmist
search = libmist.Search()
search.load_file('/path/to/data.csv')
```

(continues on next page)

(continued from previous page)

```
# parse order explicitly set with these methods
# search.load_file_row_major('/path/to/data.csv')
# search.load_file_column_major('/path/to/data.csv')
```

or a Python Numpy ndarray. The parse order is determined by the memory layout: if the array is `C_CONTIGUOUS` (the default) then it is parsed in row-major order; if the array is `F_CONTIGUOUS` then it is parsed in column-major order.

```
x = numpy.ndarray(...)
search.load_ndarray(x)

# parse order determined by x.flags
```

The ndarray is not copied, and so it must exactly match the expected format. Mist with an exception if a requirement is not met[1].

2.2.1 Missing values

During the computation of an Information measure of a tuple, Mist omits rows (samples) that have a missing value in any of the variables of the tuple. Thus, the effective sample size of the tuple used to calculate the Information measure is less than or equal to the sample size of each variable. The effective sample size may vary widely depending on the missing values pattern.

For example, you mave have a missing value rate of about 50% for each variable, but the effective sample size for a pair of variables may be much smaller than the others.

```
Variable Tuple   Missing or Present?    Effective Sample Size
(V0)             *****-----             5
(V1)             ----******             6
(V2)             ----******             6
(V0,V1)          ----*-----             1
(V0,V2)          ----*-----             1
(V1,V2)          ----******             6
```

In this contrived example, pairs involving *V0* have a much smaller effective sample size because its missing value pattern is opposite to that of the other variables. A similar situation can arise in real data, say when one variable systematically missed one half the sample population while another variable systematically missed the other half.

Under the hood, Mist computes joint entropy estimations that are sensitive to small sample size. If the effective sample size is very small, the estimate can have large fluctuations from the true entropy value. Since joint entropy estimations are used to calculate higher-order measures, such as Symmetric Delta, these fluctuation could lead to spurious results. That is why you should always check the effective sample size of any tuples with interesting signals, such as potential outliers or candidates for a functional dependence.

[1] Mist does not modify the input data to fit the requirements. We don't wish to make any invisible changes to the data that could a) inadvertently introduce bias into the data, or b) make it difficult to reproduce or validate results outside Mist.

2.3 Select an appropriate Information Theory Measure

Select the measure you want to compute with Mist::set_measure.

```
import libmist
search = libmist.Search()
search.measure = "SymmetricDelta"
```

The appropriate measure depends on the data and the question you are trying to answer. Currently, there are two measures available: Joint Entropy and Symmetric Delta.

2.3.1 Joint Entropy

An estimate of the joint entropy of two or more variables, computed using the naive approach [Shannon1949].

2.3.2 Symmetric Delta

A novel symmetric measure of functional dependence constructed from joint entropies [Galas2014]. Values are always reported as positive real numbers[2], with larger values indicating stronger signal. Missing values may cause a sign change for low-signal tuples, but these can be ignored.

2.4 Define the search Space

Mist computes the IT Measure for each tuple in the search space. Currently Mist recognizes two types of search space, Exhaustive and Custom.

2.4.1 Exhaustive (default) search space

The default search space is the set of all variable tuples. For N variables and tuples size T, the default space contains *(N choose T)* tuples. This space is called "exhaustive" or "complete" because it contains all possible unique tuples for a set of variables.

Set the size of tuples in the default space with Mist::set_tuple_size.

```
search.tuple_size = 3
```

Beware of the size of the exhaustive space: a large number of variables and tuple size 3 and greater leads to combinatorial explosion, e.g., the exhaustive search space of 5000 variables in 3-tuples is over 20 billion tuples!

[2] Symmetric Delta, as described in [Galas2014], has negative sign for odd-dimension tuples. In Mist we give the magnitude always so it is clear what tail of the distribution holds the signal.

2.4.2 Custom search space

There are many search problems where you do not need to compute all possible tuples. Perhaps you're only interested in functional relationships involving a specific variable, and so you'd like to skip tuples that do not include it.

You can define a smaller search space using the TupleSpace class. A tuple space is made by defining groups of variables, and then specifying how variables from each group should combine to form the tuples. Follow these steps to define the custom search space:

1. Create a TupleSpace object

```
import libmist
ts = libmist.TupleSpace()
```

2. Define Variable Groups

A *Variable Group* is simply a named set of variables. Variables are referenced by their position in the matrix, *[0,N-1]*. Add a group with TupleSpace::addVariableGroup. Variable groups are usually disjoint, but they do not need to be ordered or contiguous.

```
ts.addVariableGroup("A", [0,1,2,9])
ts.addVariableGroup("B", [4])
```

Note that the size of variable groups may impact performance, see below.

3. Define Variable Group Tuples

A *Variable Group Tuple* (or a group tuple for short) is a set of Variable Groups that define the tuples in the search space. Add a group tuple with TupleSpace::addVariableGroupTuple.

The group tuple is the blueprint for the variable tuples. An algorithm generates variable tuples by replacing the group name with variables from that group. Through iteration it generates all variables tuples, e.g. for variable groups *A=[a1,a2,...,aN]* and *B=[b1,b2,...,bM]*, the group tuple *[A,B]* would generate *N*M* variable tuples *[a1,b1]*, *[a1,b2]*, ..., *[aN,bM]*.

Let us illustrate the algorithm through an example:

```
ts.addVariableGroupTuple(["A", "B"])

# this group tuple generates variable tuples:
# 0,4
# 1,4
# 2,4
# 4,9
```

You can list a variable group any number of times, in any order:

```
ts.addVariableGroupTuple(["A", "B", "A"])

# this group tuple generates variable tuples:
# 0,1,4
# 0,2,4
# 0,3,4
# 1,2,4
# 1,4,9
# 2,4,9
```

Note that the order in a group tuple is not important, so the group tuples "A,B" and "B,A" result in the same set of variable tuples.

4. Set the TupleSpace

Finally, load the TupleSpace object to set the tuple space. Now, when you run the computation, only the desired tuples will be included.

```
search.tuple_space = ts
```

Note: tuple_space and tuple_size parameters are mutually exclusive. The tuple_space parameter takes precedence.

2.4.3 Preview search space size

You can count the number of tuples contained the tuple space with TupleSpace::count_tuples

```
search.tuple_space = libmist.TupleSpace(5000, 3)
search.tuple_space.count_tuples()
# returns 20820835000
```

2.4.4 Genetics Example

Consider a more realistic example in genetics. Suppose we have a single phenotype of interest and 5000 single nucleotide polymorphisms (SNPs) that might be related. If we are interested only in finding functional dependencies between two SNPs and the single phenotype, then we should exclude tuples containing only SNPs. The following few lines of code specifies this example, assuming our phenotype variable is in position 0 with all other variables being SNPs

```
ts = libmist.TupleSpace()
ts.addVariableGroup("phenotype", [0])
ts.addVariableGroup("genotypes", list(range(1, 5001)))
ts.addVariableGroupTuple(["genotypes", "phenotype"])
search.tuple_space = ts

ts.count_tuples()
# returns 12497500
```

This custom search space reduces the size from roughly 20 billion tuples to 12.5 million.

2.5 Compute

Before starting the computation of information measures you should configure the output file with Mist::set_outfile. For small search spaces this could be the `stdout` stream, but more often you will pick a file destination.

```
search.outfile = "/dev/stdout"
```

Finally run the computation.

```
search.start()
```

This may take anywhere from seconds to days depending on the size of the search space. It is a good idea to start small to get an idea of the runtime. Start with tuples size 2 based on a set of less than 1000 variables and then increase the search space.

2.6 Performance Tuning

The most important factors affecting the overall runtime of a search are the size of the search space and the number of threads. We already covered how to narrow the search space in the previous section. Set the number of threads with Mist::set_threads.

```
search.threads = 10
```

The default number of threads is the maximum allowed by the system (e.g. what you get from the `nproc` command). Setting threads equal to 0 implies the maximum allowed.

2.6.1 Advanced

The following are more fine-tuned options that should be considered for advanced uses.

Probability Distribution Algorithms

Counting probability distributions is the most time-consuming part of computing an IT Measure. See Mist::set_probability_algorithm for a list of available algorithms.

For very "tall" data (many rows for each variable) we can speed up the algorithm by casting each variable as series of bitsets, rather than using the typical vector representation. This allows faster entropy calculation at the cost of some memory and computation overhead. This option is not advantageous for "short" data, and disastrous if variables have many value bins.

It's worth experimenting with this option if your variable have three or fewer bins, and/or your variables have thousands or ten's of thousands of rows.

2.7 Notes

CHAPTER

THREE

API

mist is comprised of logically distinct components encapsulated by namespaces. Classes access other namespaces via an interface class. Users typically only need to be concerned with classes in the root namespace, whereas developers will need the rest.

3.1 mist

The root namespace includes composition classes and classes common to the sub-namespaces.

class `mist::Search`
Main user interface for mist runtime.

CPP and Python users instantiate this class, load data, and optionally call various configuration methods to define the computation. Computations begin with *start()*. Maintains state in between runs, such as intermediate value caches for improved performance.

Public Functions

void **set_measure**(std::string const &measure)
Set the IT Measure to be computed.

- Entropy : Compute only combined entropy.
- SymmetricDelta (default) : A novel symmetric measure of shared information. See Sakhanenko, Galas in the literature.

void **set_cutoff**(*it*::*entropy_type* cutoff)
Set the minimum IT Measure value to keep in results.

This option is most useful for dealing with very large TupleSpaces, the results for which cannot be stored in memory or on disk.

void **set_probability_algorithm**(std::string const &algorithm)
Set the algorithm for generating probability distributions.

- Vector (default) : Process each *Variable* as a vector. Gives best performance when *Variable* size is small or when there are many value bins.
- Bitset : Convert each distinct *Variable* value into a bitset to leverage bitwise operations. Gives best performance when *Variable* size is large and the number of value bins is small.

Performance of each algorithm depends strongly on the problem, i.e. the data, and potentially also on the system. After the number of threads, this parameter has the largest effect on runtime since distribution generation dominates the computation.

void **set_outfile**(std::string const &filename)
Set output CSV file.

void **set_ranks**(int ranks)
Set number of concurrent ranks to use in this *Search*.

A rank on a computation node is one execution thread. The default ranks is the number of threads allowed by the node. Setting ranks to 0 causes the system to use the maximum.

void **set_start_rank**(int rank)
Set the starting rank for this *Search*.

A Mist search can run in parallel on multiple nodes in a system. For each node, configure a *Search* with the starting rank, number of ranks (ie threads) on the node, and total ranks among all nodes. In this way you can divide the search space among nodes in the system.

The starting rank is the zero-indexed rank number, valid over range [0,total_ranks].

Parameters **rank** – Zero-indexed rank number

void **set_total_ranks**(int ranks)
Set the total number of ranks among all participating Searches.

Each thread on each node is counted as a rank. So the total_ranks is the sum of configured ranks (threads) on each node.

void **set_tuple_size**(int size)
Set the number of Variables to include in each IT measure computation.

void **set_tuple_space**(*algorithm*::*TupleSpace* const &ts)
Set the custom tuple space for the next computation

Side effects: sets the thread algorithm to TupleSpace so that the tuple space becomes effective immediately.

void **set_tuple_limit**(long limit)
Set the maximum number of tuples to process. The default it 0, meaning unlimited.

void **set_show_progress**(bool)
Toggle whether to write program progress to stderr.

When true, an extra thread will be made to watch progress through the TupleSpace. This option is especially useful for large searches to estimate how long the run will take.

void **set_output_intermediate**(bool)
Include all subcalculations in the output

void **set_cache_enabled**(bool)
Enable caching intermediate entropy calculation

void **set_cache_size_bytes**(unsigned long)
Set maximum size of entropy cache in bytes

void **load_file**(std::string const &filename)
Load Data from CSV or tab-separated file.

By defualt, the file is loaded in row-major order, i.e. each row is a variable.

Parameters

- **filename** – path to file
- **is_row_major** – Set to true for row-major variables

Pre each row has an equal number of columns. Load Data from CSV or tab-separated file.

void **load_ndarray**(*np*::ndarray const &np)
Load Data from Python Numpy::ndarray.

Data is loaded into the library following a zero-copy guarantee.

Parameters **np** – ndarray

Pre Array is NxM matrix of the expected dtype and C memory layout.

np::ndarray **python_get_results**()
Return a Numpy ndarray copy of all results

np::ndarray **python_start**()
Start search.

Compute the configured IT measure for all *Variable* tuples in the configured search space. And return up to tuple_limit number of results.

void **start**()
Begin computation.

Compute the configured IT measure for all *Variable* tuples in the configured search space.

std::vector<*it*::*entropy_type*> const &**get_results**()
Return a copy of all results

void **printCacheStats**()
Print cache statistics for each cache in each thread to stdout.

std::string **version**()
Return the *Search* library Version string

class mist::**Variable**
Variable wraps a pointer to a data column.

Public Types

using **data_t** = std::int8_t
Variable values must be signed so that negative values can represent missing data, and should be as small as possible to save space for very large data sets.

Public Functions

Variable(data_ptr src, std::size_t size, std::size_t index, std::size_t bins)
Variable constructor.

Wrap a shared pointer to column data along with metadata.

Parameters

- **src** – Shared pointer to memory allocated for the data column
- **size** – Number of rows in the data column
- **index** – Identifying column index into data matrix

- **bins** – Number of data value bins

Throws `invalid_argument` – data stored ptr, size, or bin argument is zero.

Pre src data has been allocated memory for at least size elements.

Pre src data values are binned to a contiguous non-negative integer array starting at 0.

Pre src missing data values are represented by negative integers.

inline bool **missing**(std::size_t pos) const
Test if data at position is missing.

Throws `std::out_of_range` –

data_t &**at**(std::size_t const pos)

Throws `out_of_range` –

Variable **deepCopy**()
Variable uses default move and copy constructors that are shallow and maintain const requirement on underlying data. A deep copy made with this extra method.

bool **operator==**(*Variable* const &other) const noexcept
Will resort to a deep inspection so two Variables with identical content in different memory locations are equivalent. Returns false if either *Variable* has invalid data, e.g. as a sideeffect of std::move.

bool **operator!=**(*Variable* const &other) const noexcept
Variable inequality test.

Public Static Functions

static bool **missingVal**(*data_t* const val)
Test if value is classified as missing.

3.2 mist::algorithm

Algorithms to divide and conquer Information Theory computations.

namespace `mist::algorithm`

class **TupleSpace**
#include <TupleSpace.hpp> Tuple Space defines the set of tuples over which to run a computation search.

Public Functions

int **addVariableGroup**(std::string const &name, tuple_t const &vars)
Define a named logical group of variables

Parameters

- **name** – group name
- **vars** – set of variables in the group, duplicates will be ignored

Throws *TupleSpaceException* – variable already listed in existing variable group

Returns index of created variable group

void **addVariableGroupTuple**(std::vector<std::string> const &groups)
Add a variable group tuple

The cross product of groups in the group tuple generates a set of variable tuples that will be added to the *TupleSpace* by TupleSpaceTupleProducer.

Parameters **groups** – Array of group names
Throws *TupleSpaceException* – group does not exists

void **addVariableGroupTuple**(tuple_t const &groups)
Add a variable group tuple

The cross product of groups in the group tuple generates a set of variable tuples that will be added to the *TupleSpace* by TupleSpaceTupleProducer.

Parameters **groups** – Array of group indexed by order created
Throws *TupleSpaceException* – group index out of range

std::vector<std::string> **names**() const
Get variable names

void **set_names**(std::vector<std::string> const &names)
Set variable names

count_t **count_tuples**() const
Calculate the size of the tuple space, i.e. count the generated number of tuples.

void **traverse**(*TupleSpaceTraverser* &traverser) const
Walk through all tuples in the tuple space

Parameters **traverser** – Process each tuple with methods defined in specialization

void **traverse**(count_t start, count_t stop, *TupleSpaceTraverser* &traverser) const
Walk through as subset of tuples in the tuple space

TupleSpace generates an ordered list of tuples, that can begin at any position in the list.

Parameters

- **start** – Begin the walk at tuple in position start
- **stop** – End the walk at tuple in position stop
- **traverser** – Process each tuple with methods defined in specialization

void **traverse_entropy**(*it*::*EntropyCalculator* &ecalc, *TupleSpaceTraverser* &traverser) const
Walk through all tuples in the tuple space, computing entropy values as you go.

Some *it::Measure* classes compute the entropy values of sub-tuples. It is most efficient to compute these as you walk through the tuple space so intermediary values can be reused many times.

Parameters

- **ecalc** – *it::EntropyCalculator* object to perform entropy computations
- **traverser** – Process each tuple with methods defined in specialization

void **traverse_entropy**(count_t start, count_t stop, *it*::*EntropyCalculator* &ecalc, *TupleSpaceTraverser* &traverser) const
Walk through a subset of tuples in the tuple space, computing entropy values as you go.

class **TupleSpaceException** : public exception

class **TupleSpaceTraverser**
#include <TupleSpace.hpp> Interface for processing tuples in the *TupleSpace*

A class can specialize the *TupleSpaceTraverser* to gain access to the stream of tuples generated by *TupleSpace::traverse* family of functions.

Subclassed by *mist::algorithm::Worker*

class **Worker** : public mist::*algorithm*::*TupleSpaceTraverser*

#include <Worker.hpp> The *Worker* class divides and conquers the tuple search space.

The *Worker* processes each tuple in the configured search space, or a portion of the search space depending on the rank parameters. It is common for each computing thread on the system to have a unique *Worker* instance.

Public Functions

Worker(tuple_space_ptr const &ts, count_t start, count_t stop, result_t cutoff, entropy_calc_ptr &calc, std::vector<output_stream_ptr> const &out_streams, measure_ptr const &measure)

Construct and configure a *Worker* instance.

Parameters

- **ts** – *TupleSpace* that defines the tuple search space
- **start** – Start processing at start tuple number
- **stop** – Stop processing when stop tuple number is reached
- **cutoff** – Discard all tuples from output with a measure less than cutoff
- **out_streams** – Collection OutputStream pointers to send results
- **measure** – The *it::Measure* to calculate the results

Worker(tuple_space_ptr const &ts, count_t start, count_t stop, entropy_calc_ptr &calc, std::vector<output_stream_ptr> const &out_streams, measure_ptr const &measure)

Construct and configure a *Worker* instance.

Cutoff is not used in the this instance.

void **start**()

Start the *Worker* search space execution. Returns when all tuples in the search space have been processed.

class **WorkerException** : public exception

3.3 mist::cache

Cache intermediate results for performance improvement.

namespace `mist::cache`

Typedefs

using **K** = *Variable*::indexes

using **V** = *it*::*entropy_type*

class **Cache**

#include <Cache.hpp> *Cache* interface

Subclassed by *mist::cache::Flat1D*, *mist::cache::Flat2D*

Public Functions

virtual bool **has**(*K* const&) = 0
Test that key is in table

virtual void **put**(*K* const&, *V* const&) = 0
Insert value at key.

virtual *V* **get**(*K* const&) = 0
Return value at key.

out_of_range Key not in table

virtual std::size_t **size**() = 0
Number of entries in table

virtual std::size_t **bytes**() = 0
Size in bytes of table

inline std::size_t **hits**()
Number of cache hits

inline std::size_t **misses**()
Number of cache misses

inline std::size_t **evictions**()
Number of cache evictions

class **Flat1D** : public mist::*cache*::*Cache*

#include <Flat1D.hpp> Fixed sized associative cache

Public Functions

virtual bool **has**(key_type const &key)
Test that key is in table

virtual void **put**(key_type const &key, val_type const &val)
Insert value at key.

virtual val_type **get**(key_type const &key)
Return value at key.

out_of_range Key not in table

virtual std::size_t **size**()
Number of entries in table

virtual std::size_t **bytes**()
Size in bytes of table

class **Flat1DException** : public exception

class **Flat1DOutOfRange** : public out_of_range

class **Flat2D** : public mist::*cache*::*Cache*
#include <Flat2D.hpp> Fixed sized associative cache

Public Functions

virtual bool **has**(key_type const &key)
Test that key is in table

virtual void **put**(key_type const &key, val_type const &val)
Insert value at key.

virtual val_type **get**(key_type const &key)
Return value at key.

out_of_range Key not in table

virtual std::size_t **size**()
Number of entries in table

virtual std::size_t **bytes**()
Size in bytes of table

class **Flat2DException** : public exception

class **Flat2DOutOfRange** : public out_of_range

3.4 mist::io

Input/Output

namespace **mist::io**

class **DataMatrix**
#include <DataMatrix.hpp> N x M input data matrix.

Columns are interpreted as variables with each row a sample.

class **DataMatrixException** : public exception

class **FileOutputStream** : public mist::*io*::*OutputStream*

class **FileOutputStreamException** : public exception

class **FlatOutputStream** : public mist::*io*::*OutputStream*

Public Functions

void **relocate**(*FlatOutputStream* &other)
Move all data in other to this object

class **FlatOutputStreamException** : public exception

class **MapOutputStream** : public mist::*io*::*OutputStream*

class **OutputStream**
Subclassed by *mist::io::FileOutputStream*, *mist::io::FlatOutputStream*, *mist::io::MapOutputStream*

3.5 mist::it

Information Theory definitions and algorithms.

namespace `mist::it`

Typedefs

using **Bitset** = boost::dynamic_bitset<unsigned long long>

using **BitsetVariable** = std::vector<*Bitset*>

using **BitsetTable** = std::vector<*BitsetVariable*>

using **DistributionData** = double

using **entropy_type** = double

using **Entropy** = std::vector<*entropy_type*>

Enums

enum **d1**
Values:

enumerator **e0**

enumerator **size**

enum **d2**
Values:

enumerator **e0**

enumerator **e1**

enumerator **e01**

enumerator **size**

enum **d3**
Values:

enumerator **e0**

enumerator **e1**

enumerator **e2**

enumerator **e01**

enumerator **e02**

enumerator **e12**

enumerator **e012**

enumerator **size**

enum **d4**
Values:

enumerator **e0**

enumerator **e1**

enumerator **e2**

enumerator **e3**

enumerator **e01**

enumerator **e02**

enumerator **e03**

enumerator **e12**

enumerator **e13**

enumerator **e23**

enumerator **e012**

enumerator **e013**

enumerator **e023**

enumerator **e123**

enumerator **e0123**

enumerator **size**

class **BitsetCounter** : public mist::*it*::*Counter*
#include <BitsetCounter.hpp> Generates a ProbabilityDistribution from a *Variable* tuple.

Recasts each *Variable* as an array of bitsets, one for each bin value. Computes the ProbabilityDistribution using bitwise AND operation and bit counting algorithm.

class **BitsetCounterOutOfRange** : public out_of_range

class **Counter**
#include <Counter.hpp> Abstract class. Generates a Probability *Distribution* from a *Variable* tuple

Subclassed by *mist::it::BitsetCounter*, *mist::it::VectorCounter*

class **Distribution**
#include <Distribution.hpp> Joint probability array for N variables

Public Functions

template<class **Container**>
inline **Distribution**(*Container* const &strides)
Construct directly from dimension strides

inline **Distribution**(*Variable*::tuple const &vars)
Construct from a *Variable* tuple

inline void **scale**(double factor)
Multiply each value in distribution by factor

inline void **normalize**()
Normalize distribution

class **DistributionOutOfRange** : public out_of_range

class **EntropyCalculator**

class **EntropyCalculatorException** : public exception

class **EntropyMeasure** : public mist::*it*::*Measure*

Public Functions

virtual result_type **compute**(*EntropyCalculator* &ecalc, *Variable*::indexes const &tuple) const
Compute the information theory measure with the computation ecalc for the given variables.
Returns final result

virtual result_type **compute**(*EntropyCalculator* &ecalc, *Variable*::indexes const &tuple, *Entropy* const &e) const
Compute the information theory measure with the the given variables, using pre-computed entropies. Only useful for measures that use entropy sub calculations.

virtual std::string **header**(int d, bool full_output) const
Return a comma-separated header string corresponding to the full results
Parameters
- **d** – tuple size
- **full_output** – whether header should include all subcalculation names

Returns header string

virtual std::vector<std::string> const &**names**(int d, bool full_output) const
Return array of names for each column in the output
Parameters
- **d** – tuple size
- **full_output** – whether header should include all subcalculation names

Returns array of column names in the output

inline virtual bool **full_entropy**() const
Whether this measure uses intermediate entropy calculations

class **EntropyMeasureException** : public exception

class **Measure**
Subclassed by *mist::it::EntropyMeasure*, *mist::it::SymmetricDelta*

Public Functions

virtual result_type **compute**(*EntropyCalculator* &ecalc, *Variable*::indexes const &tuple) const = 0
Compute the information theory measure with the computation ecalc for the given variables.
Returns final result

virtual result_type **compute**(*EntropyCalculator* &ecalc, *Variable*::indexes const &tuple, *Entropy* const &entropy) const = 0
Compute the information theory measure with the the given variables, using pre-computed entropies. Only useful for measures that use entropy sub calculations.

virtual std::string **header**(int d, bool full_output) const = 0
Return a comma-separated header string corresponding to the full results
Parameters
- **d** – tuple size
- **full_output** – whether header should include all subcalculation names

Returns header string

virtual std::vector<std::string> const &**names**(int d, bool full_output) const = 0
Return array of names for each column in the output
Parameters
- **d** – tuple size

- **full_output** – whether header should include all subcalculation names

Returns array of column names in the output

virtual bool **full_entropy**() const = 0
Whether this measure uses intermediate entropy calculations

class **SymmetricDelta** : public mist::*it*::*Measure*

Public Functions

virtual result_type **compute**(*EntropyCalculator* &ecalc, *Variable*::indexes const &tuple) const
Compute the information theory measure with the computation ecalc for the given variables.

Returns final result

virtual result_type **compute**(*EntropyCalculator* &ecalc, *Variable*::indexes const &tuple, *Entropy* const &e) const
Compute the information theory measure with the the given variables, using pre-computed entropies. Only useful for measures that use entropy sub calculations.

virtual std::string **header**(int d, bool full_output) const
Return a comma-separated header string corresponding to the full results

Parameters

- **d** – tuple size
- **full_output** – whether header should include all subcalculation names

Returns header string

virtual std::vector<std::string> const &**names**(int d, bool full_output) const
Return array of names for each column in the output

Parameters

- **d** – tuple size
- **full_output** – whether header should include all subcalculation names

Returns array of column names in the output

inline virtual bool **full_entropy**() const
Whether this measure uses intermediate entropy calculations

class **SymmetricDeltaException** : public exception

class **VectorCounter** : public mist::*it*::*Counter*
#include <VectorCounter.hpp> Generates a ProbabilityDistribution from a *Variable* tuple.

Counts using standard algorithm.

class **VectorCounterException** : public exception

BIBLIOGRAPHY

[Galas2014] Galas, David J et al. “Describing the complexity of systems: multivariable “set complexity” and the information basis of systems biology.” Journal of computational biology : a journal of computational molecular cell biology vol. 21,2 (2014): 118-40. doi:10.1089/cmb.2013.0039 PMC

[Shannon1949] Shannon, Claude Elwood, and Warren Weaver. The Mathematical Theory of Communicaton. University of Illinois Press, 1949.

INDEX

M