dit Documentation

Release 1.2.1

dit Contributors

May 31, 2018

Contents

dit is a Python package for discrete information theory.

CHAPTER 1

Introduction

Information theory is a powerful extension to probability and statistics, quantifying dependencies among arbitrary random variables in a way tha tis consistent and comparable across systems and scales. Information theory was originally developed to quantify how quickly and reliably information could be transmitted across an arbitrary channel. The demands of modern, data-driven science have been coopting and extending these quantities and methods into unknown, multivariate settings where the interpretation and best practices are not known. For example, there are at least four reasonable multivariate generalizations of the mutual information, none of which inherit all the interpretations of the standard bivariate case. Which is best to use is context-dependent. `dit` implements a vast range of multivariate information measures in an effort to allow information practitioners to study how these various measures behave and interact in a variety of contexts. We hope that having all these measures and techniques implemented in one place will allow the development of robust techniques for the automated quantification of dependencies within a system and concrete interpretation of what those dependencies mean.

For a quick tour, see the *Quickstart*. Otherwise, work your way through the various sections. Note that all code snippets in this documentation assume that the following lines of code have already been run:

```
In [1]: from __future__ import division # true division for Python 2.7

In [2]: import dit

In [3]: import numpy as np
```

Contents:

1.1 General Information

Documentation: http://docs.dit.io

Downloads: https://pypi.org/project/dit/

Dependencies:

- Python 2.7, 3.3, 3.4, 3.5, or 3.6
- boltons

- contextlib2
- debtcollector
- networkx
- numpy
- prettytable
- scipy
- six

1.1.1 Optional Dependencies

- colorama: colored column heads in PID indicating failure modes
- cython: faster sampling from distributions
- hypothesis: random sampling of distributions
- matplotlib, python-ternary: plotting of various information-theoretic expansions
- numdifftools: numerical evaluation of gradients and hessians during optimization
- pint: add units to informational values
- scikit-learn: faster nearest-neighbor lookups during entropy/mutual information estimation from samples

Mailing list: None

Code and bug tracker: https://github.com/dit/dit

License: BSD 3-Clause, see LICENSE.txt for details.

Quickstart

The basic usage of `dit` corresponds to creating distributions, modifying them if need be, and then computing properties of those distributions. First, we import:

```
In [1]: import dit
```

Suppose we have a really thick coin, one so thick that there is a reasonable chance of it landing on its edge. Here is how we might represent the coin in `dit`.

```
In [2]: d = dit.Distribution(['H', 'T', 'E'], [.4, .4, .2])

In [3]: print(d)
Class:          Distribution
Alphabet:       ('E', 'H', 'T') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 1
RV Names:       None

x   p(x)
E   1/5
H   2/5
T   2/5
```

Calculate the probability of H and also of the combination: H **or** T.

```
In [4]: d['H']
Out[4]: 0.4

In [5]: d.event_probability(['H','T'])
Out[5]: 0.8
```

Calculate the Shannon entropy and extropy of the joint distribution.

```
In [6]: dit.shannon.entropy(d)
Out[6]: 1.5219280948873621

In [7]: dit.other.extropy(d)
Out[7]: 1.1419011889093373
```

Create a distribution representing the **xor** logic function. Here, we have two inputs, X and Y, and then an output $Z = \textbf{xor}(X, Y)$.

```
In [8]: import dit.example_dists

In [9]: d = dit.example_dists.Xor()

In [10]: d.set_rv_names(['X', 'Y', 'Z'])

In [11]: print(d)
Class:          Distribution
Alphabet:       ('0', '1') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 3
RV Names:       ('X', 'Y', 'Z')

x     p(x)
000   1/4
011   1/4
101   1/4
110   1/4
```

Calculate the Shannon mutual informations $I[X : Z]$, $I[Y : Z]$, and $I[X, Y : Z]$.

```
In [12]: dit.shannon.mutual_information(d, ['X'], ['Z'])
Out[12]: 0.0

In [13]: dit.shannon.mutual_information(d, ['Y'], ['Z'])
Out[13]: 0.0

In [14]: dit.shannon.mutual_information(d, ['X', 'Y'], ['Z'])
Out[14]: 1.0
```

Calculate the marginal distribution $P(X, Z)$. Then print its probabilities as fractions, showing the mask.

```
In [15]: d2 = d.marginal(['X', 'Z'])

In [16]: print(d2.to_string(show_mask=True, exact=True))
Class:          Distribution
Alphabet:       ('0', '1') for all rvs
Base:           linear
```

```
Outcome Class:  str
Outcome Length: 2 (mask: 3)
RV Names:       ('X', 'Z')

x      p(x)
0*0    1/4
0*1    1/4
1*0    1/4
1*1    1/4
```

Convert the distribution probabilities to log (base 3.5) probabilities, and access its probability mass function.

```
In [17]: d2.set_base(3.5)

In [18]: d2.pmf
Out[18]: array([-1.10658951, -1.10658951, -1.10658951, -1.10658951])
```

Draw 5 random samples from this distribution.

Enjoy!

1.2 Notation

`dit` is a scientific tool, and so, much of this documentation will contain mathematical expressions. Here we will describe this notation.

1.2.1 Basic Notation

A random variable X consists of *outcomes* x from an *alphabet* $\mathcal{X}$. As such, we write the entropy of a distribution as $HX = \sum_{x \in \mathcal{X}} p(x) \log_2 p(x)$, where $p(x)$ denote the probability of the outcome x occuring.

Many distributions are *joint* distribution. In the absence of variable names, we index each random variable with a subscript. For example, a distribution over three variables is written $X_0 X_1 X_2$. As a shorthand, we also denote those random variables as $X_{0:3}$, meaning start with X_0 and go through, but not including X_3 — just like python slice notation.

If a set of variables $X_{0:n}$ are independent, we will write $\perp\!\!\!\perp X_{0:n}$. If a set of variables $X_{0:n}$ are independent conditioned on V, we write $\perp\!\!\!\perp X_{0:n} \mid V$.

If we ever need to describe an infinitely long chain of variables we drop the index from the side that is infinite. So $X_{:0} = \ldots X_{-3} X_{-2} X_{-1}$ and $X_{0:} = X_0 X_1 X_2 \ldots$. For an arbitrary set of indices A, the corresponding collection of random variables is denoted X_A. For example, if $A = \{0, 2, 4\}$, then $X_A = X_0 X_2 X_4$. The complement of A (with respect to some universal set) is denoted $\overline{A}$.

Furthermore, we define $0 \log_2 0 = 0$.

1.2.2 Advanced Notation

When there exists a function $Y = f(X)$ we write $X \succeq Y$ meaning that X is *informationally richer* than Y. Similarly, if $f(Y) = X$ then we write $X \preceq Y$ and say that X is *informationally poorer* than Y. If $X \preceq Y$ and $X \succeq Y$ then we write $X \cong Y$ and say that X is *informationally equivalent* to Y. Of all the variables that are poorer than both X and Y, there is a richest one. This variable is known as the *meet* of X and Y and is denoted $X \curlywedge Y$. By definition, $\forall Z s.t. Z \preceq X$ and $Z \preceq Y, Z \preceq X \curlywedge Y$. Similarly of all variables richer than both X and Y, there is a poorest. This variable is known as the *join* of X and Y and is denoted $X \curlyvee Y$. The joint random variable (X, Y) and the join are informationally equivalent: $(X, Y) \cong X \curlyvee Y$.

Lastly, we use $X \searrow Y$ to denote the minimal sufficient statistic of X about the random variable Y.

1.3 Distributions

Distributions in *dit* come in two different flavors: `ScalarDistribution` and `Distribution`. `ScalarDistribution` is used for representing distributions over real numbers, and have many features related to that. `Distribution` is used for representing joint distributions, and therefore has many features related to marginalizing, conditioning, and otherwise exploring the relationships between random variables.

1.3.1 Numpy-based ScalarDistribution

ScalarDistributions are used to represent distributions over real numbers, for example a six-sided die or the number of heads when flipping 100 coins.

Playing with ScalarDistributions

First we will enable two optional features: printing fractions by default, and using `__str__()` as `__repr__()`. Be careful using either of these options, they can incur significant performance hits on some distributions.

```
In [1]: dit.ditParams['print.exact'] = dit.ditParams['repr.print'] = True
```

We next construct a six-sided die:

```
In [2]: from dit.example_dists import uniform

In [3]: d6 = uniform(1, 7)

In [4]: d6
Out[4]:
Class:    ScalarDistribution
Alphabet: (1, 2, 3, 4, 5, 6)
Base:     linear

x   p(x)
1   1/6
2   1/6
3   1/6
4   1/6
5   1/6
6   1/6
```

We can perform standard mathematical operations with scalars, such as adding, subtracting from or by, multiplying, taking the modulo, or testing inequalities.

```
In [5]: d6 + 3
Out[5]:
Class:    ScalarDistribution
Alphabet: (4, 5, 6, 7, 8, 9)
Base:     linear

x   p(x)
```

```
4   1/6
5   1/6
6   1/6
7   1/6
8   1/6
9   1/6

In [6]: d6 - 1
Out[6]:
Class:    ScalarDistribution
Alphabet: (0, 1, 2, 3, 4, 5)
Base:     linear

x   p(x)
0   1/6
1   1/6
2   1/6
3   1/6
4   1/6
5   1/6

In [7]: 10 - d6
Out[7]:
Class:    ScalarDistribution
Alphabet: (4, 5, 6, 7, 8, 9)
Base:     linear

x   p(x)
4   1/6
5   1/6
6   1/6
7   1/6
8   1/6
9   1/6

In [8]: 2 * d6
Out[8]:
Class:    ScalarDistribution
Alphabet: (2, 4, 6, 8, 10, 12)
Base:     linear

x    p(x)
2    1/6
4    1/6
6    1/6
8    1/6
10   1/6
12   1/6

In [9]: d6 % 2
Out[9]:
Class:    ScalarDistribution
Alphabet: (0, 1)
Base:     linear

x   p(x)
0   1/2
1   1/2
```

```
In [10]: (d6 % 2).is_approx_equal(d6 <= 3)
Out[10]: True
```

Furthermore, we can perform such operations with two distributions:

```
In [11]: d6 + d6
Out[11]:
Class:    ScalarDistribution
Alphabet: (2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
Base:     linear

x    p(x)
2    1/36
3    1/18
4    1/12
5    1/9
6    5/36
7    1/6
8    5/36
9    1/9
10   1/12
11   1/18
12   1/36

In [12]: (d6 + d6) % 4
Out[12]:
Class:    ScalarDistribution
Alphabet: (0, 1, 2, 3)
Base:     linear

x   p(x)
0   1/4
1   2/9
2   1/4
3   5/18

In [13]: d6 // d6
Out[13]:
Class:    ScalarDistribution
Alphabet: (0, 1, 2, 3, 4, 5, 6)
Base:     linear

x   p(x)
0   5/12
1   1/3
2   1/9
3   1/18
4   1/36
5   1/36
6   1/36

In [14]:  d6 % (d6 % 2 + 1)
Out[14]:
Class:    ScalarDistribution
Alphabet: (0, 1)
Base:     linear
```

```
x   p(x)
0   3/4
1   1/4
```

There are also statistical functions which can be applied to `ScalarDistributions`:

```
In [15]: from dit.algorithms.stats import *

In [16]: median(d6+d6)
Out[16]: 7.0

In [17]: from dit.example_dists import binomial

In [18]: d = binomial(10, 1/3)

In [19]: d
Out[19]:
Class:    ScalarDistribution
Alphabet: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
Base:     linear

x    p(x)
0    409/23585
1    4302/49615
2    1280/6561
3    5120/19683
4    4480/19683
5    896/6561
6    1120/19683
7    320/19683
8    20/6561
9    9/26572
10   1/59046

In [20]: mean(d)
Out[20]: 3.3333333333333335

In [21]: median(d)
Out[21]: 3.0

In [22]: standard_deviation(d)
Out[22]: 1.4907119849998596
```

API

`ScalarDistribution.`**`__init__`**(*outcomes*, *pmf=None*, *sample_space=None*, *base=None*, *prng=None*, *sort=True*, *sparse=True*, *trim=True*, *validate=True*)

Initialize the distribution.

Parameters

- **outcomes** (*sequence, dict*) – The outcomes of the distribution. If *outcomes* is a dictionary, then the keys are used as *outcomes*, and the values of the dictionary are used as *pmf* instead. Note: an outcome is any hashable object (except *None*) which is equality comparable. If *sort* is *True*, then outcomes must also be orderable.
- **pmf** (*sequence*) – The outcome probabilities or log probabilities. If *None*, then *outcomes*

is treated as the probability mass function and the outcomes are consecutive integers beginning from zero.

- **sample_space** (*sequence*) – A sequence representing the sample space, and corresponding to the complete set of possible outcomes. The order of the sample space is important. If *None*, then the outcomes are used to determine the sample space instead.
- **base** (*float, None*) – If *pmf* specifies log probabilities, then *base* should specify the base of the logarithm. If 'linear', then *pmf* is assumed to represent linear probabilities. If *None*, then the value for *base* is taken from ditParams['base'].
- **prng** (*RandomState*) – A pseudo-random number generator with a *rand* method which can generate random numbers. For now, this is assumed to be something with an API compatible to NumPy's RandomState class. This attribute is initialized to equal dit.math.prng.
- **sort** (*bool*) – If *True*, then the sample space is sorted before finalizing it. Usually, this is desirable, as it normalizes the behavior of distributions which have the same sample space (when considered as a set). Note that addition and multiplication of distributions is defined only if the sample spaces (as tuples) are equal.
- **sparse** (*bool*) – Specifies the form of the pmf. If *True*, then *outcomes* and *pmf* will only contain entries for non-null outcomes and probabilities, after initialization. The order of these entries will always obey the order of *sample_space*, even if their number is not equal to the size of the sample space. If *False*, then the pmf will be dense and every outcome in the sample space will be represented.
- **trim** (*bool*) – Specifies if null-outcomes should be removed from pmf when *make_sparse()* is called (assuming *sparse* is *True*) during initialization.
- **validate** (*bool*) – If *True*, then validate the distribution. If *False*, then assume the distribution is valid, and perform no checks.

Raises

- `InvalidDistribution` – If the length of *values* and *outcomes* are unequal.
- See `validate()` for a list of other potential exceptions.

1.3.2 Numpy-based Distribution

The primary method of constructing a distribution is by supplying both the outcomes and the probability mass function:

```
In [1]: from dit import Distribution

In [2]: outcomes = ['000', '011', '101', '110']

In [3]: pmf = [1/4]*4

In [4]: xor = Distribution(outcomes, pmf)

In [5]: print(xor)
Class:          Distribution
Alphabet:       ('0', '1') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 3
RV Names:       None

x     p(x)
000   0.25
```

```
011   0.25
101   0.25
110   0.25
```

Another way to construct a distribution is by supplying a dictionary mapping outcomes to probabilities:

```
In [6]: outcomes_probs = {'000': 1/4, '011': 1/4, '101': 1/4, '110': 1/4}

In [7]: xor2 = Distribution(outcomes_probs)

In [8]: print(xor2)
Class:          Distribution
Alphabet:       ('0', '1') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 3
RV Names:       None

x     p(x)
000   0.25
011   0.25
101   0.25
110   0.25
```

Yet a third method is via an ndarray:

```
In [9]: pmf = [[0.5, 0.25], [0.25, 0]]

In [10]: d = Distribution.from_ndarray(pmf)

In [11]: print(d)
Class:          Distribution
Alphabet:       (0, 1) for all rvs
Base:           linear
Outcome Class:  tuple
Outcome Length: 2
RV Names:       None

x         p(x)
(0, 0)    0.5
(0, 1)    0.25
(1, 0)    0.25
```

`Distribution.`**`__init__`**(*outcomes*, *pmf=None*, *sample_space=None*, *base=None*, *prng=None*, *sort=True*, *sparse=True*, *trim=True*, *validate=True*)

Initialize the distribution.

Parameters

- **`outcomes`** (*sequence*, *dict*) – The outcomes of the distribution. If *outcomes* is a dictionary, then the keys are used as *outcomes*, and the values of the dictionary are used as *pmf* instead. The values will not be used if probabilities are passed in via *pmf*. Outcomes must be hashable, orderable, sized, iterable containers. The length of an outcome must be the same for all outcomes, and every outcome must be of the same type.
- **`pmf`** (*sequence*, *None*) – The outcome probabilities or log probabilities. *pmf* can be None only if *outcomes* is a dict.
- **`sample_space`** (*sequence*, *CartesianProduct*) – A sequence representing the

sample space, and corresponding to the complete set of possible outcomes. The order of the sample space is important. If *None*, then the outcomes are used to determine a Cartesian product sample space instead.

- **base** (*float, str, None*) – If *pmf* specifies log probabilities, then *base* should specify the base of the logarithm. If 'linear', then *pmf* is assumed to represent linear probabilities. If *None*, then the value for *base* is taken from ditParams['base'].
- **prng** (*RandomState*) – A pseudo-random number generator with a *rand* method which can generate random numbers. For now, this is assumed to be something with an API compatibile to NumPy's RandomState class. This attribute is initialized to equal dit.math.prng.
- **sort** (*bool*) – If *True*, then each random variable's alphabets are sorted before they are finalized. Usually, this is desirable, as it normalizes the behavior of distributions which have the same sample spaces (when considered as a set). Note that addition and multiplication of distributions is defined only if the sample spaces are compatible.
- **sparse** (*bool*) – Specifies the form of the pmf. If *True*, then *outcomes* and *pmf* will only contain entries for non-null outcomes and probabilities, after initialization. The order of these entries will always obey the order of *sample_space*, even if their number is not equal to the size of the sample space. If *False*, then the pmf will be dense and every outcome in the sample space will be represented.
- **trim** (*bool*) – Specifies if null-outcomes should be removed from pmf when *make_sparse()* is called (assuming *sparse* is *True*) during initialization.
- **validate** (*bool*) – If *True*, then validate the distribution. If *False*, then assume the distribution is valid, and perform no checks.

Raises

- `InvalidDistribution` – If the length of *values* and *outcomes* are unequal. If no outcomes can be obtained from *pmf* and *outcomes* is *None*.
- See `validate()` for a list of other potential exceptions.

To verify that these two distributions are the same, we can use the *is_approx_equal* method:

```
In [12]: xor.is_approx_equal(xor2)
Out[12]: True
```

`Distribution.`**`is_approx_equal`**(*other*, *rtol=None*, *atol=None*)

Returns *True* is *other* is approximately equal to this distribution.

For two distributions to be equal, they must have the same sample space and must also agree on the probabilities of each outcome.

Parameters

- **other** (*distribution*) – The distribution to compare against.
- **rtol** (*float*) – The relative tolerance to use when comparing probabilities.
- **atol** (*float*) – The absolute tolerance to use when comparing probabilities.

Notes

The distributions need not have the length, but they must have the same base.

1.4 Operations

There are several operations possible on joint random variables. Let's consider the standard `xor` distribution:

```
In [1]: d = dit.Distribution(['000', '011', '101', '110'], [1/4]*4)

In [2]: d.set_rv_names('XYZ')
```

1.4.1 Marginal

`dit` supports two ways of selecting only a subset of random variables. *`marginal()`* returns a distribution containing only the random variables specified, whereas *`marginalize()`* return a distribution containing all random variables *except* the ones specified:

```
In [3]: print(d.marginal('XY'))
Class:          Distribution
Alphabet:       ('0', '1') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 2
RV Names:       ('X', 'Y')

x    p(x)
00   1/4
01   1/4
10   1/4
11   1/4

In [4]: print(d.marginalize('XY'))
Class:          Distribution
Alphabet:       ('0', '1') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 1
RV Names:       ('Z',)

x   p(x)
0   1/2
1   1/2
```

`Distribution.`**`marginal`**`(`*rvs*, *rv_mode=None*`)`

Returns a marginal distribution.

Parameters

- **`rvs`** (*`list`*) – The random variables to keep. All others are marginalized.
- **`rv_mode`** (*`str, None`*) – Specifies how to interpret the elements of *rvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *self._rv_mode* is consulted.

Returns **d** – A new joint distribution with the random variables in *rvs* kept and all others marginalized.

Return type joint distribution

Distribution.**marginalize**(*rvs*, *rv_mode=None*)

Returns a new distribution after marginalizing random variables.

Parameters

- **rvs** (*list*) – The random variables to marginalize. All others are kept.
- **rv_mode** (*str, None*) – Specifies how to interpret the elements of *rvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *self._rv_mode* is consulted.

Returns **d** – A new joint distribution with the random variables in *rvs* marginalized and all others kept.

Return type joint distribution

1.4.2 Conditional

We can also condition on a subset of random variables:

```
In [5]: marginal, cdists = d.condition_on('XY')

In [6]: print(marginal)
Class:          Distribution
Alphabet:       ('0', '1') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 2
RV Names:       ('X', 'Y')

x    p(x)
00   1/4
01   1/4
10   1/4
11   1/4

In [7]: print(cdists[0]) # XY = 00
Class:          Distribution
Alphabet:       ('0', '1') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 1
RV Names:       ('Z',)

x   p(x)
0   1

In [8]: print(cdists[1]) # XY = 01
Class:          Distribution
Alphabet:       ('0', '1') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 1
RV Names:       ('Z',)

x   p(x)
1   1
```

```
In [9]: print(cdists[2]) # XY = 10
Class:          Distribution
Alphabet:       ('0', '1') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 1
RV Names:       ('Z',)

x   p(x)
1   1

In [10]: print(cdists[3]) # XY = 11
Class:          Distribution
Alphabet:       ('0', '1') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 1
RV Names:       ('Z',)

x   p(x)
0   1
```

`Distribution.`**`condition_on`**(*crvs*, *rvs=None*, *rv_mode=None*, *extract=False*)

Returns distributions conditioned on random variables `crvs`.

Optionally, `rvs` specifies which random variables should remain.

NOTE: Eventually this will return a conditional distribution.

Parameters

- **`crvs`** (*`list`*) – The random variables to condition on.
- **`rvs`** (*`list, None`*) – The random variables for the resulting conditional distributions. Any random variable not represented in the union of `crvs` and `rvs` will be marginalized. If `None`, then every random variable not appearing in `crvs` is used.
- **`rv_mode`** (*`str, None`*) – Specifies how to interpret `crvs` and `rvs`. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of `crvs` and `rvs` are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random varible names. If `None`, then the value of `self._rv_mode` is consulted, which defaults to 'indices'.
- **`extract`** (*`bool`*) – If the length of either `crvs` or `rvs` is 1 and `extract` is `True`, then instead of the new outcomes being 1-tuples, we extract the sole element to create scalar distributions.

Returns

- **cdist** (*dist*) – The distribution of the conditioned random variables.
- **dists** (*list of distributions*) – The conditional distributions for each outcome in `cdist`.

Examples

First we build a distribution P(X,Y,Z) representing the XOR logic gate.

```
>>> pXYZ = dit.example_dists.Xor()
>>> pXYZ.set_rv_names('XYZ')
```

We can obtain the conditional distributions P(X,Z|Y) and the marginal of the conditioned variable P(Y) as follows:

```
>>> pY, pXZgY = pXYZ.condition_on('Y')
```

If we specify `rvs='Z'`, then only 'Z' is kept and thus, 'X' is marginalized out:

```
>>> pY, pZgY = pXYZ.condition_on('Y', rvs='Z')
```

We can condition on two random variables:

```
>>> pXY, pZgXY = pXYZ.condition_on('XY')
```

The equivalent call using indexes is:

```
>>> pXY, pZgXY = pXYZ.condition_on([0, 1], rv_mode='indexes')
```

1.4.3 Join

We can construct the join of two random variables:

$$X \curlyvee Y = \min\{V | V \succeq X \land V \succeq Y\}$$

Where $\min$ is understood to be minimizing with respect to the entropy.

```
In [11]: from dit.algorithms.lattice import join

In [12]: print(join(d, ['XY']))
Class:    ScalarDistribution
Alphabet: (0, 1, 2, 3)
Base:     linear

x   p(x)
0   1/4
1   1/4
2   1/4
3   1/4
```

join (*dist*, *rvs*, *rv_mode=None*, *int_outcomes=True*)
Returns the distribution of the join of random variables defined by *rvs*.

Parameters

- **dist** (*Distribution*) – The distribution which defines the base sigma-algebra.
- **rvs** (*list*) – A list of lists. Each list specifies a random variable to be joined with the other lists. Each random variable can defined as a series of unique indexes. Multiple random variables can use the same index. For example, [[0,1],[1,2]].
- **rv_mode** (*str, None*) – Specifies how to interpret the elements of *rvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted.
- **int_outcomes** (*bool*) – If *True*, then the outcomes of the join are relabeled as integers instead of as the atoms of the induced sigma-algebra.

Returns **d** – The distribution of the join.

Return type ScalarDistribution

insert_join (*dist*, *idx*, *rvs*, *rv_mode=None*)

Returns a new distribution with the join inserted at index *idx*.

The join of the random variables in *rvs* is constructed and then inserted into at index *idx*.

Parameters

- **dist** (*Distribution*) – The distribution which defines the base sigma-algebra.
- **idx** (*int*) – The index at which to insert the join. To append the join, set *idx* to be equal to -1 or dist.outcome_length().
- **rvs** (*list*) – A list of lists. Each list specifies a random variable to be met with the other lists. Each random variable can defined as a series of unique indexes. Multiple random variables can use the same index. For example, [[0,1],[1,2]].
- **rv_mode** (*str, None*) – Specifies how to interpret the elements of *rvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted.

Returns **d** – The new distribution with the join at index *idx*.

Return type Distribution

1.4.4 Meet

We can construct the meet of two random variabls:

$$X \curlywedge Y = \max\{V | V \preceq X \wedge V \preceq Y\}$$

Where $\max$ is understood to be maximizing with respect to the entropy.

```
In [13]: from dit.algorithms.lattice import meet

In [14]: outcomes = ['00', '01', '10', '11', '22', '33']

In [15]: d2 = dit.Distribution(outcomes, [1/8]*4 + [1/4]*2, sample_space=outcomes)

In [16]: d2.set_rv_names('XY')

In [17]: print(meet(d2, ['X', 'Y']))
Class:    ScalarDistribution
Alphabet: (0, 1, 2)
Base:     linear

x   p(x)
0   1/4
1   1/4
2   1/2
```

meet (*dist*, *rvs*, *rv_mode=None*, *int_outcomes=True*)

Returns the distribution of the meet of random variables defined by *rvs*.

Parameters

- **dist** (*Distribution*) – The distribution which defines the base sigma-algebra.

- **rvs** (*list*) – A list of lists. Each list specifies a random variable to be met with the other lists. Each random variable can defined as a series of unique indexes. Multiple random variables can use the same index. For example, [[0,1],[1,2]].
- **rv_mode** (*str, None*) – Specifies how to interpret the elements of *rvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted.
- **int_outcomes** (*bool*) – If *True*, then the outcomes of the meet are relabeled as integers instead of as the atoms of the induced sigma-algebra.

Returns **d** – The distribution of the meet.

Return type ScalarDistribution

insert_meet (*dist, idx, rvs, rv_mode=None*)

Returns a new distribution with the meet inserted at index *idx*.

The meet of the random variables in *rvs* is constructed and then inserted into at index *idx*.

Parameters

- **dist** (*Distribution*) – The distribution which defines the base sigma-algebra.
- **idx** (*int*) – The index at which to insert the meet. To append the meet, set *idx* to be equal to -1 or dist.outcome_length().
- **rvs** (*list*) – A list of lists. Each list specifies a random variable to be met with the other lists. Each random variable can defined as a series of unique indexes. Multiple random variables can use the same index. For example, [[0,1],[1,2]].
- **rv_mode** (*str, None*) – Specifies how to interpret the elements of *rvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted.

Returns **d** – The new distribution with the meet at index *idx*.

Return type Distribution

1.4.5 Minimal Sufficient Statistic

This method constructs the minimal sufficient statistic of X about Y: $X \searrow Y$:

$$X \searrow Y = \min\{V|V \preceq X \wedge I[X:Y] = I[V:Y]\}$$

```
In [18]: from dit.algorithms import insert_mss

In [19]: d2 = dit.Distribution(['00', '01', '10', '11', '22', '33'], [1/8]*4 + [1/
↪4]*2)

In [20]: print(insert_mss(d2, -1, [0], [1]))
Class:          Distribution
Alphabet:       (('0', '1', '2', '3'), ('0', '1', '2', '3'), ('2', '0', '1'))
Base:           linear
Outcome Class:  str
Outcome Length: 3
RV Names:       None
```

```
x     p(x)
002   1/8
012   1/8
102   1/8
112   1/8
220   1/4
331   1/4
```

Again, min is understood to be over entropies.

mss (*dist*, *rvs*, *about=None*, *rv_mode=None*, *int_outcomes=True*)

Parameters

- **dist** (*Distribution*) – The distribution which defines the base sigma-algebra.
- **rvs** (*list*) – A list of random variables to be compressed into a minimal sufficient statistic.
- **about** (*list*) – A list of random variables for which the minimal sufficient static will retain all information about.
- **rv_mode** (*str, None*) – Specifies how to interpret the elements of *rvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted.
- **int_outcomes** (*bool*) – If *True*, then the outcomes of the minimal sufficient statistic are relabeled as integers instead of as the atoms of the induced sigma-algebra.

Returns **d** – The distribution of the minimal sufficient statistic.

Return type ScalarDistribution

Examples

```
>>> d = Xor()
>>> print(mss(d, [0], [1, 2]))
Class:    ScalarDistribution
Alphabet: (0, 1)
Base:     linear
x   p(x)
0   0.5
1   0.5
```

insert_mss (*dist*, *idx*, *rvs*, *about=None*, *rv_mode=None*)

Inserts the minimal sufficient statistic of *rvs* about *about* into *dist* at index *idx*.

Parameters

- **dist** (*Distribution*) – The distribution which defines the base sigma-algebra.
- **idx** (*int*) – The location in the distribution to insert the minimal sufficient statistic.
- **rvs** (*list*) – A list of random variables to be compressed into a minimal sufficient statistic.
- **about** (*list*) – A list of random variables for which the minimal sufficient static will retain all information about.
- **rv_mode** (*str, None*) – Specifies how to interpret the elements of *rvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *rvs* are interpreted as

random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted.

Returns d – The distribution *dist* modified to contain the minimal sufficient statistic.

Return type Distribution

Examples

```
>>> d = Xor()
>>> print(insert_mss(d, -1, [0], [1, 2]))
Class:          Distribution
Alphabet:       ('0', '1') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 4
RV Names:       None
x      p(x)
0000   0.25
0110   0.25
1011   0.25
1101   0.25
```

`dit.util.testing.`**`distributions`**`()`

1.5 Finding Examples

What if you'd like to find a distribution that has a particular property? For example, what if I'd like to find a distribution with a coinformation less that -0.5? This is where Hypothesis comes in:

```
In [1]: from hypothesis import find

In [2]: from dit.utils.testing import distributions

In [3]: find(distributions(3, 2), lambda d: dit.multivariate.coinformation(d) < -0.5)
Out[3]:
Class:          Distribution
Alphabet:       (0, 1) for all rvs
Base:           linear
Outcome Class:  tuple
Outcome Length: 3
RV Names:       None

x           p(x)
(0, 0, 0)   1/5
(0, 1, 1)   1/5
(1, 0, 0)   1/5
(1, 0, 1)   1/5
(1, 1, 0)   1/5
```

What hypothesis has done is use the `distributions()` *strategy* to randomly test distributions. Once it finds a distribution satisfying the criteria we specified (coinformation less than -0.5) it then simplifies the example as much as possible. Here, we see that even though it could have found any distribution, it found the exclusive or distribution, and simplified the probabilities to be uniform.

1.6 Optimization

It is often useful to construct a distribution d' which is consistent with some marginal aspects of d, but otherwise optimizes some information measure. For example, perhaps we are interested in constructing a distribution which matches pairwise marginals with another, but otherwise has maximum entropy:

```
In [1]: from dit.algorithms.distribution_optimizers import MaxEntOptimizer

In [2]: xor = dit.example_dists.Xor()

In [3]: meo = MaxEntOptimizer(xor, [[0,1], [0,2], [1,2]])

In [4]: meo.optimize()
Out[4]:
     fun: -3.000001614256971
     jac: array([-2.99999911, -3.00000122, -3.00000045, -2.99999905, -3.00000066,
       -2.99999931, -2.99999923, -3.0000006 ])
 message: 'Optimization terminated successfully.'
    nfev: 972
     nit: 88
    njev: 88
  status: 0
 success: True
       x: array([0.1250001 , 0.12500003, 0.12500002, 0.12500013, 0.12500003,
       0.12500011, 0.12500011, 0.12500001])

In [5]: dp = meo.construct_dist()

In [6]: print(dp)
Class:          Distribution
Alphabet:       ('0', '1') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 3
RV Names:       None

x     p(x)
000   381517/3052135
001   326484/2611873
010   338160/2705281
011   333148/2665183
100   388824/3110593
101   355162/2841295
110   315997/2527975
111   332113/2656905
```

1.6.1 Helper Functions

There are three special functions to handle common optimization problems:

```
In [7]: from dit.algorithms import maxent_dist, marginal_maxent_dists
```

The first is maximum entropy distributions with specific fixed marginals. It encapsulates the steps run above:

```
In [8]: print(maxent_dist(xor, [[0,1], [0,2], [1,2]]))
Class:          Distribution
```

```
Alphabet:       ('0', '1') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 3
RV Names:       None

x     p(x)
000   2180623/17444985
001   2540385/20323081
010   1299289/10394311
011   1/8
100   3550763/28406103
101   2408030/19264239
110   2243316/17946529
111   4961378/39691025
```

The second constructs several maximum entropy distributions, each with all subsets of variables of a particular size fixed:

```
In [9]: k0, k1, k2, k3 = marginal_maxent_dists(xor)
```

where `k0` is the maxent dist corresponding the same alphabets as `xor`; `k1` fixes $p(x_0)$, $p(x_1)$, and $p(x_2)$; `k2` fixes $p(x_0, x_1)$, $p(x_0, x_2)$, and $p(x_1, x_2)$ (as in the `maxent_dist` example above), and finally `k3` fixes $p(x_0, x_1, x_2)$ (e.g. is the distribution we started with).

1.7 Information Measures

`dit` supports many information measures, ranging from as standard as the Shannon entropy to as exotic as Gács-Körner common information (with even more esoteric measure coming soon!). We organize these quantities into the following groups.

We first have the Shannon-like measures. These quantities are based on sums and differences of entropies, conditional entropies, or mutual informations of random variables:

1.7.1 Basic Shannon measures

The information on this page is drawn from the fantastic text book **Elements of Information Theory** by Cover and Thomas *[CT06]*. Other good choices are **Information Theory, Inference and Learning Algorithms** by MacKay *[Mac03]* and **Information Theory and Network Coding** by Yeung *[Yeu08]*.

Entropy

The entropy measures how much information is in a random variable X.

$$HX = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x)$$

What do we mean by "how much information"? Basically, we mean the average number of yes-no questions one would have to ask to determine an outcome from the distribution. In the simplest case, consider a sure thing:

```
In [1]: d = dit.Distribution(['H'], [1])

In [2]: dit.shannon.entropy(d)
Out[2]: 0.0
```

So since we know that the outcome from our distribution will always be H, we have to ask zero questions to figure that out. If however we have a fair coin:

```
In [3]: d = dit.Distribution(['H', 'T'], [1/2, 1/2])

In [4]: dit.shannon.entropy(d)
Out[4]: 1.0
```

The entropy tells us that we must ask one question to determine whether an H or T was the outcome of the coin flip. Now what if there are three outcomes? Let's consider the following situation:

```
In [5]: d = dit.Distribution(['A', 'B', 'C'], [1/2, 1/4, 1/4])

In [6]: dit.shannon.entropy(d)
Out[6]: 1.5
```

Here we find that the entropy is 1.5 bits. How do we ask one and a half questions on average? Well, if our first question is "was it A?" and it is true, then we are done, and that occurs half the time. The other half of the time we need to ask a follow up question: "was it B?". So half the time we need to ask one question, and the other half of the time we need to ask two questions. In other words, we need to ask 1.5 questions on average.

Joint Entropy

The entropy of multiple variables is computed in a similar manner:

$$HX_{0:n} = -\sum_{x_{0:n} \in X_{0:n}} p(x_{0:n}) \log_2 p(x_{0:n})$$

Its intuition is also the same: the average number of binary questions required to identify a joint event from the distribution.

API

entropy(*dist*, *rvs=None*, *rv_mode=None*)

Returns the entropy H[X] over the random variables in *rvs*.

If the distribution represents linear probabilities, then the entropy is calculated with units of 'bits' (base-2). Otherwise, the entropy is calculated in whatever base that matches the distribution's pmf.

Parameters

- **dist** (`Distribution or float`) – The distribution from which the entropy is calculated. If a float, then we calculate the binary entropy.
- **rvs** (`list, None`) – The indexes of the random variable used to calculate the entropy. If None, then the entropy is calculated over all random variables. This should remain *None* for ScalarDistributions.
- **rv_mode** (`str, None`) – Specifies how to interpret the elements of *rvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted.

Returns **H** – The entropy of the distribution.

Return type float

Conditional Entropy

The conditional entropy is the amount of information in variable X beyond that which is in variable Y:

$$HX|Y = -\sum_{x\in X,y\in Y} p(x,y)\log_2 p(x|y)$$

As a simple example, consider two identical variables:

```
In [7]: d = dit.Distribution(['HH', 'TT'], [1/2, 1/2])

In [8]: dit.shannon.conditional_entropy(d, [0], [1])
Out[8]: 0.0
```

We see that knowing the second variable tells us everything about the first, leaving zero entropy. On the other end of the spectrum, two independent variables:

```
In [9]: d = dit.Distribution(['HH', 'HT', 'TH', 'TT'], [1/4]*4)

In [10]: dit.shannon.conditional_entropy(d, [0], [1])
Out[10]: 1.0
```

Here, the second variable tells us nothing about the first so we are left with the one bit of information a coin flip has.

API

conditional_entropy(*dist*, *rvs_X*, *rvs_Y*, *rv_mode=None*)

Returns the conditional entropy of H[X|Y].

If the distribution represents linear probabilities, then the entropy is calculated with units of 'bits' (base-2).

Parameters

- **dist** (*Distribution*) – The distribution from which the conditional entropy is calculated.
- **rvs_X** (*list, None*) – The indexes of the random variables defining X.
- **rvs_Y** (*list, None*) – The indexes of the random variables defining Y.
- **rv_mode** (*str, None*) – Specifies how to interpret the elements of *rvs_X* and *rvs_Y*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *rvs_X* and *rvs_Y* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted.

Returns **H_XgY** – The conditional entropy H[X|Y].

Return type float

Mutual Information

The mutual information is the amount of information shared by X and Y:

$$\begin{aligned} IX:Y &= HX,Y - HX|Y - HY|X \\ &= HX + HY - HX,Y \\ &= \sum_{x\in X,y\in Y} p(x,y)\log_2 \frac{p(x,y)}{p(x)p(y)} \end{aligned}$$

The mutual information is symmetric:

$$IX : Y = IY : X$$

Meaning that the information that X carries about Y is equal to the information that Y carries about X. The entropy of X can be decomposed into the information it shares with Y and the information it doesn't:

$$HX = IX : Y + HX|Y$$

See also:

The mutual information generalized to the multivariate case in three different ways:

Co-Information Generalized as the information which *all* variables contribute to.

Total Correlation Generalized as the sum of the information in the individual variables minus the information in the whole.

Dual Total Correlation Generalized as the joint entropy minus the entropy of each variable conditioned on the others.

CAEKL Mutual Information Generalized as the smallest quantity that can be subtracted from the joint, and from each part of a partition of all the variables, such that the joint entropy minus this quantity is equal to the sum of each partition entropy minus this quantity.

API

`mutual_information`(*dist*, *rvs_X*, *rvs_Y*, *rv_mode=None*)

Returns the mutual information I[X:Y].

If the distribution represents linear probabilities, then the entropy is calculated with units of 'bits' (base-2).

Parameters

- **dist** (*Distribution*) – The distribution from which the mutual information is calculated.
- **rvs_X** (*list, None*) – The indexes of the random variables defining X.
- **rvs_Y** (*list, None*) – The indexes of the random variables defining Y.
- **rv_mode** (*str, None*) – Specifies how to interpret the elements of *rvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted.

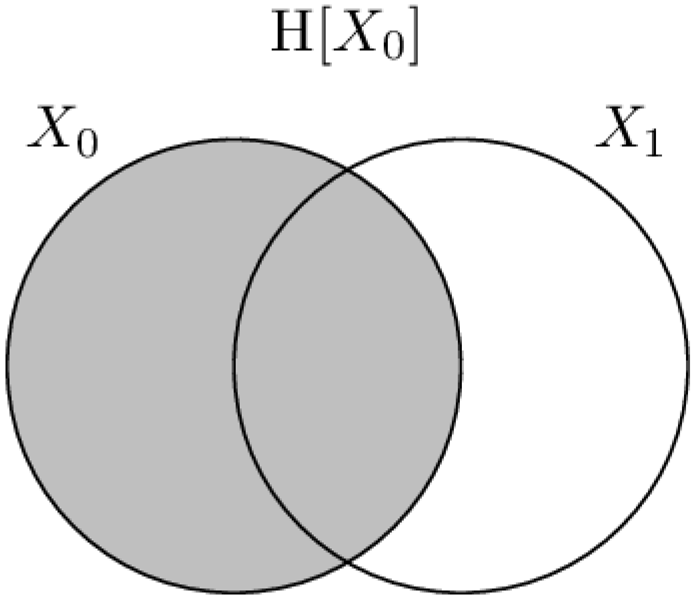
Returns **I** – The mutual information I[X:Y].

Return type float

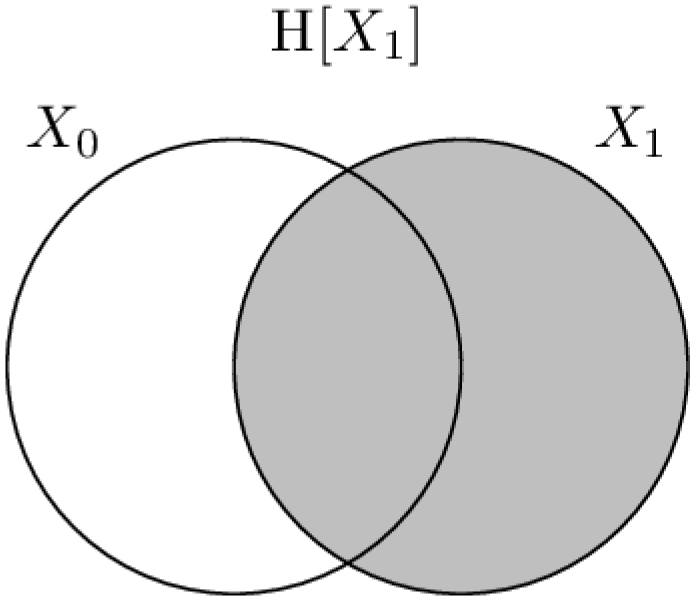
Visualization of Information

It has been shown that there is a correspondence between set-theoretic measures and information-theoretic measures. The entropy is equivalent to set cardinality, mutual information to set intersection, and conditional entropy to set difference. Because of this we can use Venn-like diagrams to represent the information in and shared between random variables. These diagrams are called *information diagrams* or i-diagrams for short.

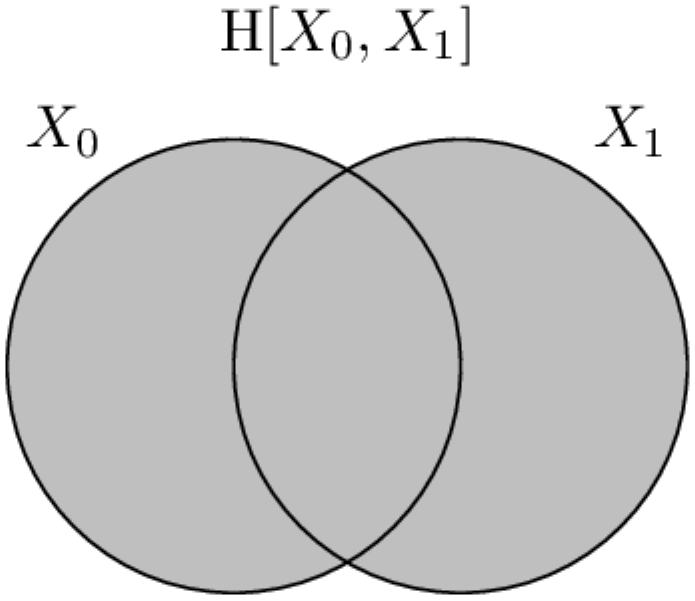
This first image pictographically shades the area of the i-diagram which contains the information corresponding to HX_0.



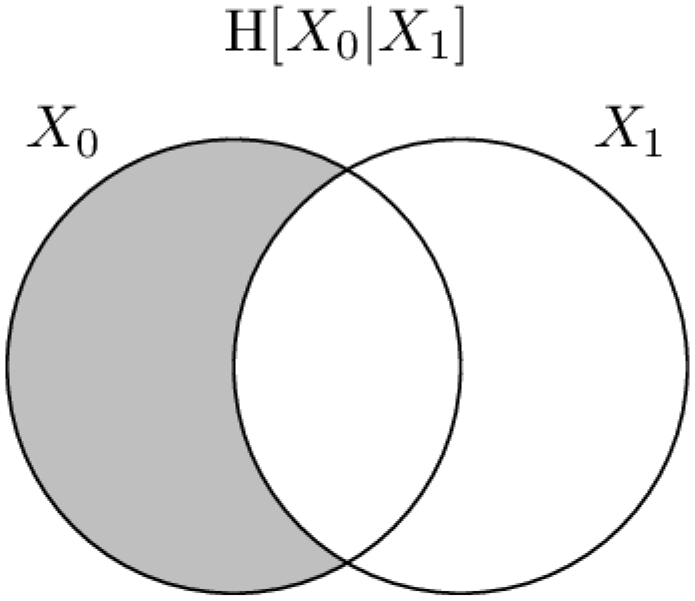
Similarly, this one shades the information corresponding to HX_1.



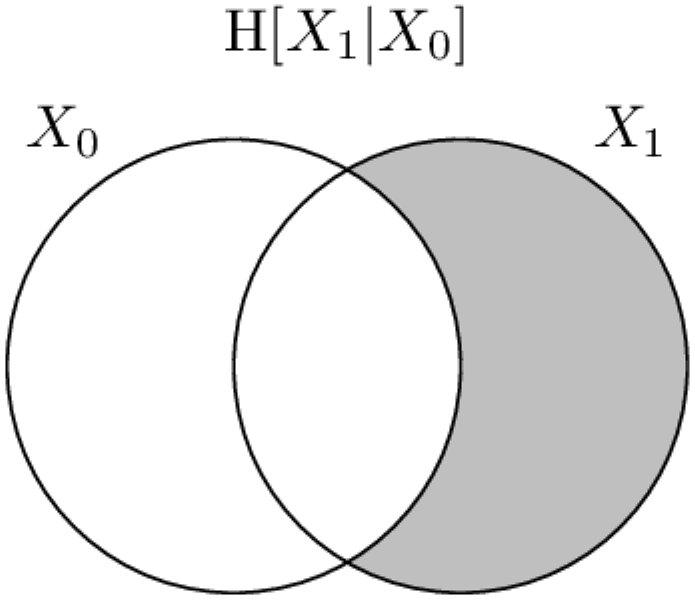
This image shades the information corresponding to HX_0, X_1. Notice that it is the union of the prior two, and not their sum (e.g. that overlap region is not double-counted).



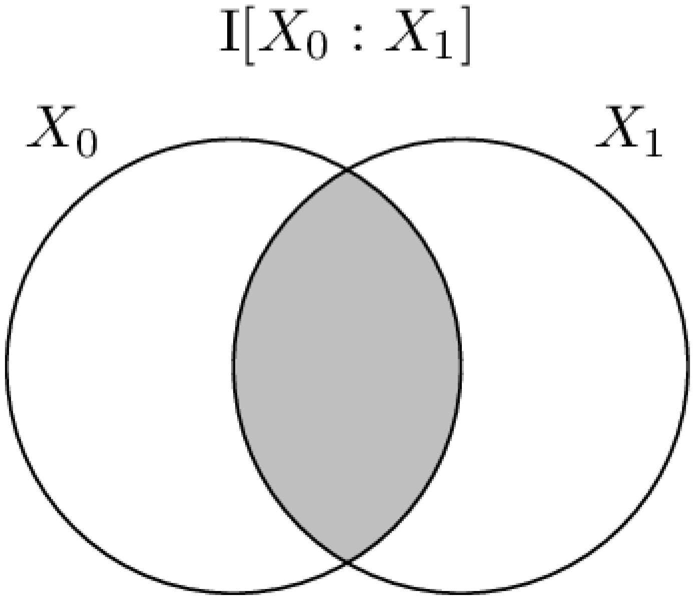
Next, the conditional entropy of X_0 conditioned on X_1, $HX_0|X_1$, is displayed. It consists of the area contained in the X_0 circle but not contained in X_1 circle.



In the same vein, here the conditional entropy $HX_1|X_0$ is shaded.



Finally, the mutual information between X_0 and X_1, $IX_0 : X_1$ is drawn. It is the region where the two circles overlap.



1.7.2 Multivariate

Multivariate measures of information generally attempt to capture some global property of a joint distribution. For example, they might attempt to quantify how much information is shared among the random variables, or quantify how "non-indpendent" in the joint distribution is.

Total Information

These quantities, currently just the Shannon entropy, measure the total amount of information contained in a set of joint variables.

Entropy

The entropy measures the total amount of information contained in a set of random variables, $X_{0:n}$, potentially excluding the information contain in others, $Y_{0:m}$.

$$HX_{0:n}|Y_{0:m} = -\sum_{\substack{x_{0:n}\in\mathcal{X}_{0:n}\\ y_{0:m}\in\mathcal{Y}_{0:m}}} p(x_{0:n}, y_{0:m}) \log_2 p(x_{0:n}|y_{0:m})$$

Let's consider two coins that are interdependent: the first coin fips fairly, and if the first comes up heads, the other is fair, but if the first comes up tails the other is certainly tails:

```
In [1]: d = dit.Distribution(['HH', 'HT', 'TT'], [1/4, 1/4, 1/2])
```

We would expect that entropy of the second coin conditioned on the first coin would be 0.5 bits, and sure enough that is what we find:

```
In [2]: from dit.multivariate import entropy

In [3]: entropy(d, [1], [0])
Out[3]: 0.49999999999999999
```

And since the first coin is fair, we would expect it to have an entropy of 1 bit:

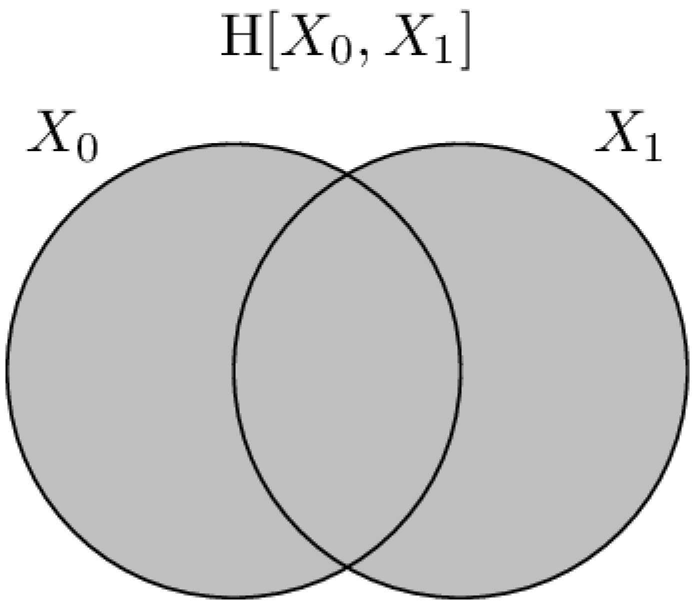
```
In [4]: entropy(d, [0])
Out[4]: 1.0
```

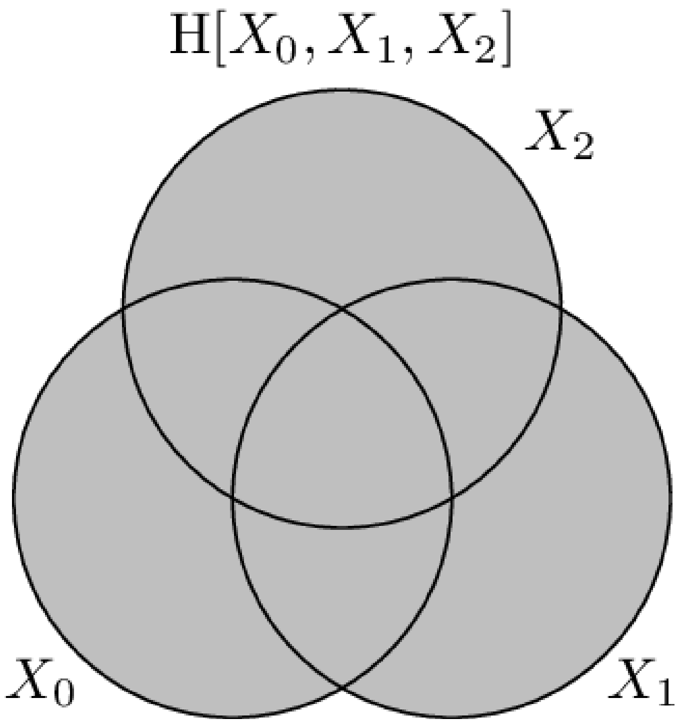
Taken together, we would then expect the joint entropy to be 1.5 bits:

```
In [5]: entropy(d)
Out[5]: 1.5
```

Visualization

Below we have a pictoral representation of the joint entropy for both 2 and 3 variable joint distributions.





API

entropy(**args*, ***kwargs*)

Calculates the conditional joint entropy.

Parameters

- **dist** (*Distribution*) – The distribution from which the entropy is calculated.
- **rvs** (*list, None*) – The indexes of the random variable used to calculate the entropy. If None, then the entropy is calculated over all random variables.
- **crvs** (*list, None*) – The indexes of the random variables to condition on. If None, then no variables are conditioned on.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **H** – The entropy.

Return type float

Raises `ditException` – Raised if *rvs* or *crvs* contain non-existant random variables.

Examples

Let's construct a 3-variable distribution for the XOR logic gate and name the random variables X, Y, and Z.

```
>>> d = dit.example_dists.Xor()
>>> d.set_rv_names(['X', 'Y', 'Z'])
```

The joint entropy of H[X,Y,Z] is:

```
>>> dit.multivariate.entropy(d, 'XYZ')
2.0
```

We can do this using random variables indexes too.

```
>>> dit.multivariate.entropy(d, [0,1,2], rv_mode='indexes')
2.0
```

The joint entropy H[X,Z] is given by:

```
>>> dit.multivariate.entropy(d, 'XZ')
1.0
```

Conditional entropy can be calculated by passing in the conditional random variables. The conditional entropy H[Y|X] is:

```
>>> dit.multivariate.entropy(d, 'Y', 'X')
1.0
```

Mutual Informations

These measures all reduce to the standard Shannon *Mutual Information* for bivariate distributions.

Co-Information

The co-information *[Bel03]* is one generalization of the *Mutual Information* to multiple variables. The co-information quantifies the amount of infomration that *all* variables participate in. It is defined via an inclusion/exclusion sum:

$$\begin{aligned} IX_{0:n} &= -\sum_{y \in \mathcal{P}(\{0..n\})} (-1)^{|y|} HX_y \\ &= \sum_{x_{0:n} \in X_{0:n}} p(x_{0:n}) \log_2 \prod_{y \in \mathcal{P}(\{0..n\})} p(y)^{(-1)^{|y|}} \end{aligned}$$

It is clear that the co-information measures the "center-most" atom of the diagram only, which is the only atom to which every variable contributes. To exemplifying this, consider "giant bit" distributions:

```
In [1]: from dit import Distribution as D

In [2]: from dit.multivariate import coinformation as I

In [3]: [ I(D(['0'*n, '1'*n], [1/2, 1/2])) for n in range(2, 6) ]
Out[3]: [1.0, 1.0, 1.0, 1.0]
```

This verifies intuition that the entire one bit of the distribution's entropy is condensed in a single atom. One notable property of the co-information is that for $n \geq 3$ it can be negative. For example:

```
In [4]: from dit.example_dists import Xor

In [5]: d = Xor()
In [6]: I(d)
Out[6]: -1.0
```

Based on these two examples one might get the impression that the co-information is positive for “redundant” distributions and negative for “synergistic” distributions. This however is not true — consider the four-variable parity distribution:

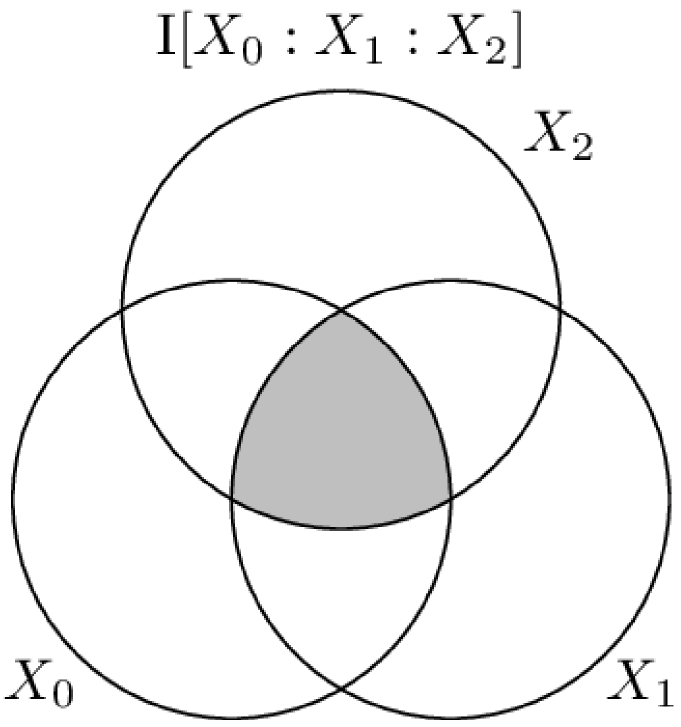
```
In [7]: from dit.example_dists import n_mod_m

In [8]: d = n_mod_m(4, 2)

In [9]: I(d)
Out[9]: 1.0
```

Meaning that the co-information is positive for both the most redundant distribution, the giant bit, and the most synergistic, the parity. Therefore the coinformation can not be used to measure redundancy or synergy.

Note: Correctly measuring redundancy and synergy is an ongoing problem. See [Griffith2013] and references therein for the current status of the problem.

Visualization

The co-information can be visuallized on an i-diagram as below, where only the centermost atom is shaded:



API

coinformation(**args*, ***kwargs*)
Calculates the coinformation.

Parameters

- **dist** (*Distribution*) – The distribution from which the coinformation is calculated.
- **rvs** (*list, None*) – The indexes of the random variable used to calculate the coinformation between. If None, then the coinformation is calculated over all random variables.
- **crvs** (*list, None*) – The indexes of the random variables to condition on. If None, then no variables are condition on.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **I** – The coinformation.

Return type float

Raises ditException – Raised if *dist* is not a joint distribution or if *rvs* or *crvs* contain non-existant random variables.

Examples

Let's construct a 3-variable distribution for the XOR logic gate and name the random variables X, Y, and Z.

```
>>> d = dit.example_dists.Xor()
>>> d.set_rv_names(['X', 'Y', 'Z'])
```

To calculate coinformations, recall that *rvs* specifies which groups of random variables are involved. For example, the 3-way mutual information I[X:Y:Z] is calculated as:

```
>>> dit.multivariate.coinformation(d, ['X', 'Y', 'Z'])
-1.0
```

It is a quirk of strings that each element of a string is also an iterable. So an equivalent way to calculate the 3-way mutual information I[X:Y:Z] is:

```
>>> dit.multivariate.coinformation(d, 'XYZ')
-1.0
```

The reason this works is that list('XYZ') == ['X', 'Y', 'Z']. If we want to use random variable indexes, we need to have explicit groupings:

```
>>> dit.multivariate.coinformation(d, [[0], [1], [2]], rv_mode='indexes')
-1.0
```

To calculate the mutual information I[X, Y : Z], we use explicit groups:

```
>>> dit.multivariate.coinformation(d, ['XY', 'Z'])
```

Using indexes, this looks like:

```
>>> dit.multivariate.coinformation(d, [[0, 1], [2]], rv_mode='indexes')
```

The mutual information I[X:Z] is given by:

```
>>> dit.multivariate.coinformation(d, 'XZ')
0.0
```

Equivalently,

```
>>> dit.multivariate.coinformation(d, ['X', 'Z'])
0.0
```

Using indexes, this becomes:

```
>>> dit.multivariate.coinformation(d, [[0], [2]])
0.0
```

Conditional mutual informations can be calculated by passing in the conditional random variables. The conditional entropy I[X:Y|Z] is:

```
>>> dit.multivariate.coinformation(d, 'XY', 'Z')
1.0
```

Using indexes, this becomes:

```
>>> rvs = [[0], [1]]
>>> crvs = [[2]] # broken
>>> dit.multivariate.coinformation(d, rvs, crvs, rv_mode='indexes')
1.0
```

For the conditional random variables, groupings have no effect, so you can also obtain this as:

```
>>> rvs = [[0], [1]]
>>> crvs = [2]
>>> dit.multivariate.coinformation(d, rvs, crvs, rv_mode='indexes')
1.0
```

Finally, note that entropy can also be calculated. The entropy H[Z|XY] is obtained as:

```
>>> rvs = [[2]]
>>> crvs = [[0], [1]] # broken
>>> dit.multivariate.coinformation(d, rvs, crvs, rv_mode='indexes')
0.0
```

```
>>> crvs = [[0, 1]] # broken
>>> dit.multivariate.coinformation(d, rvs, crvs, rv_mode='indexes')
0.0
```

```
>>> crvs = [0, 1]
>>> dit.multivariate.coinformation(d, rvs, crvs, rv_mode='indexes')
0.0
```

```
>>> rvs = 'Z'
>>> crvs = 'XY'
>>> dit.multivariate.coinformation(d, rvs, crvs, rv_mode='indexes')
0.0
```

Note that [[0], [1]] says to condition on two groups. But conditioning is a flat operation and doesn't respect the groups, so it is equal to a single group of 2 random variables: [[0, 1]]. With random variable names 'XY' is acceptable because list('XY') = ['X', 'Y'], which is species two singleton groups. By the previous argument, this is will be treated the same as ['XY'].

Total Correlation

The total correlation *[Wat60]*, denoted T, also known as the multi-information or integration, is one generalization of the *Mutual Information*. It is defined as the amount of information each individual variable carries above and beyond the joint entropy, e.g. the difference between the whole and the sum of its parts:

$$\begin{aligned} TX_{0:n} &= \sum HX_i - HX_{0:n} \\ &= \sum_{x_{0:n} \in X_{0:n}} p(x_{0:n}) \log_2 \frac{p(x_{0:n})}{\prod p(x_i)} \end{aligned}$$

Two nice features of the total correlation are that it is non-negative and that it is zero if and only if the random variables $X_{0:n}$ are all independent. Some baseline behavior is good to note also. First its behavior when applied to "giant bit" distributions:

```
In [1]: from dit import Distribution as D

In [2]: from dit.multivariate import total_correlation as T

In [3]: [ T(D(['0'*n, '1'*n], [0.5, 0.5])) for n in range(2, 6) ]
Out[3]: [1.0, 2.0, 3.0, 4.0]
```

So we see that for giant bit distributions, the total correlation is equal to one less than the number of variables. The second type of distribution to consider is general parity distributions:

```
In [4]: from dit.example_dists import n_mod_m

In [5]: [ T(n_mod_m(n, 2)) for n in range(3, 6) ]
Out[5]: [1.0, 1.0, 1.0]

In [6]: [ T(n_mod_m(3, m)) for m in range(2, 5) ]
Out[6]: [1.0, 1.584962500721156, 2.0]
```

Here we see that the total correlation is equal to $\log_2 m$ regardless of n.

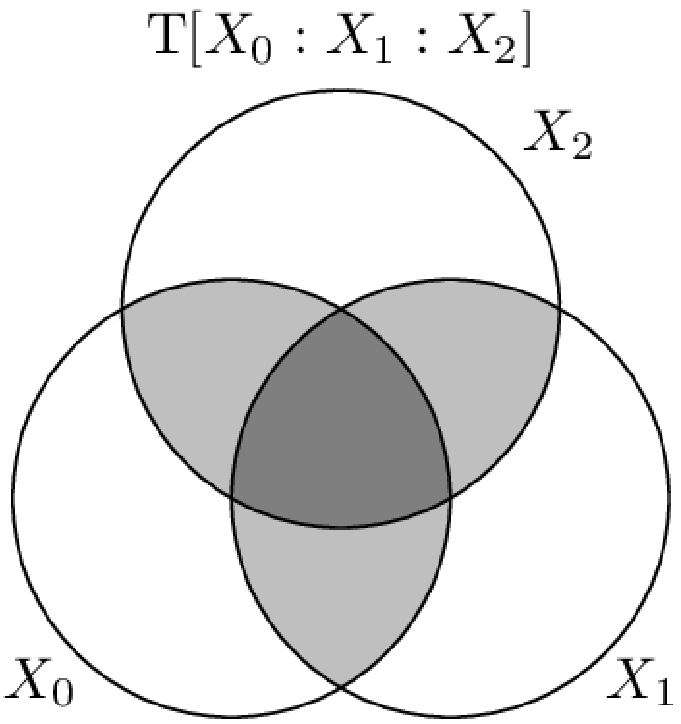
The total correlation follows a nice decomposition rule. Given two sets of (not necessarily independent) random variables, A and B, the total correaltion of $A \cup B$ is:

$$TA \cup B = TA + TB + IA : B$$

```
In [7]: from dit.multivariate import coinformation as I

In [8]: d = n_mod_m(4, 3)

In [9]: T(d) == T(d, [[0], [1]]) + T(d, [[2], [3]]) + I(d, [[0, 1], [2, 3]])
Out[9]: True
```

Visualization

The total correlation consists of all information that is shared among the variables, and weights each piece according to how many variables it is shared among.



API

total_correlation(**args*, ***kwargs*)

Computes the total correlation, also known as either the multi-information or the integration.

Parameters

- **dist** (*Distribution*) – The distribution from which the total correlation is calculated.
- **rvs** (*list, None*) – A list of lists. Each inner list specifies the indexes of the random variables used to calculate the total correlation. If None, then the total correlation is calculated over all random variables, which is equivalent to passing *rvs=dist.rvs*.
- **crvs** (*list, None*) – A single list of indexes specifying the random variables to condition on. If None, then no variables are conditioned on.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **T** – The total correlation.

Return type float

Examples

```
>>> d = dit.example_dists.Xor()
>>> dit.multivariate.total_correlation(d)
1.0
```

```
>>> dit.multivariate.total_correlation(d, rvs=[[0], [1]])
0.0
```

Raises `ditException` – Raised if *dist* is not a joint distribution or if *rvs* or *crvs* contain non-existant random variables.

Dual Total Correlation

The dual total correlation *[Han75]*, or binding information *[AP12]*, is yet another generalization of the *Mutual Information*. It is the amount of information that is shared among the variables. It is defined as:

$$\begin{aligned} BX_{0:n} &= HX_{0:n} - \sum HX_i | X_{\{0..n\}/i} \\ &= - \sum_{x_{0:n} \in X_{0:n}} p(x_{0:n}) \log_2 \frac{p(x_{0:n})}{\prod p(x_i | x_{\{0:n\}/i})} \end{aligned}$$

In a sense the binding information captures the same information that the *Total Correlation* does, in that both measures are zero or non-zero together. However, the two measures take on very different quantitative values for different distributions. By way of example, the type of distribution that maximizes the total correlation is a "giant bit":

```
In [1]: from dit.multivariate import binding_information, total_correlation

In [2]: d = dit.Distribution(['000', '111'], [1/2, 1/2])

In [3]: total_correlation(d)
Out[3]: 2.0

In [4]: binding_information(d)
Out[4]: 1.0
```

For the same distribution, the dual total correlation takes on a relatively low value. On the other hand, the type of distribution that maximizes the dual total correlation is a "parity" distribution:

```
In [5]: from dit.example_dists import n_mod_m

In [6]: d = n_mod_m(3, 2)

In [7]: total_correlation(d)
Out[7]: 1.0

In [8]: binding_information(d)
Out[8]: 2.0
```

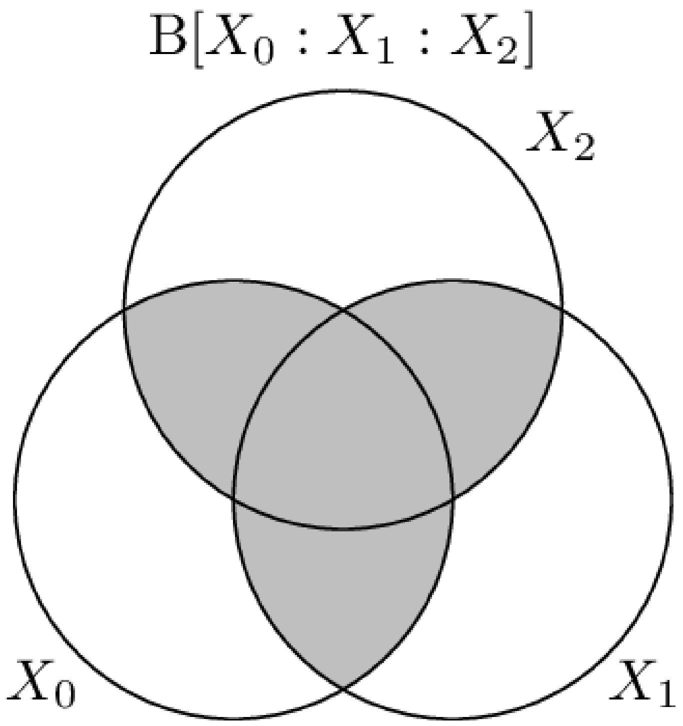
Relationship to Other Measures

The dual total correlation obeys particular bounds related to both the *Entropy* and the *Total Correlation*:

$$\begin{aligned} 0 &\leq BX_{0:n} \leq HX_{0:n} \\ \frac{TX_{0:n}}{n-1} &\leq BX_{0:n} \leq (n-1) TX_{0:n} \end{aligned}$$

Visualization

The binding information, as seen below, consists equally of the information shared among the variables.



API

dual_total_correlation(**args*, ***kwargs*)
Calculates the dual total correlation, also known as the binding information.

Parameters

- **dist** (`Distribution`) – The distribution from which the dual total correlation is calculated.
- **rvs** (`list, None`) – The indexes of the random variable used to calculate the binding information. If None, then the dual total correlation is calculated over all random variables.
- **crvs** (`list, None`) – The indexes of the random variables to condition on. If None, then no variables are condition on.
- **rv_mode** (`str, None`) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **B** – The dual total correlation.

Return type float

Raises `ditException` – Raised if *dist* is not a joint distribution or if *rvs* or *crvs* contain non-existant random variables.

CAEKL Mutual Information

The Chan-AlBashabsheh-Ebrahimi-Kaced-Liu mutual information *[CABE+15]* is one possible generalization of the *Mutual Information*.

$JX_{0:n}$ is the smallest γ such that:

$$HX_{0:n} - \gamma = \sum_{C \in \mathcal{P}} [HX_C - \gamma]$$

for some non-trivial partition $\mathcal{P}$ of $\{0 : n\}$. For example, the CAEKL mutual information for the `xor` distribution is $\frac{1}{2}$, because the joint entropy is 2 bits, each of the three marginals is 1 bit, and $2 - \frac{1}{2} = 3(1 - \frac{1}{2})$.

```
In [1]: from dit.multivariate import caekl_mutual_information as J

In [2]: d = dit.example_dists.Xor()

In [3]: J(d)
Out[3]: 0.5
```

A more concrete way of defining the CAEKL mutual information is:

$$JX_{0:n} = \min_{\mathcal{P} \in \Pi} \operatorname{I}_{\mathcal{P}} [X_{0:n}]$$

where $\operatorname{I}_{\mathcal{P}}$ is the total_correlation of the partition:

$$\operatorname{I}_{\mathcal{P}} [X_{0:n}] = \sum_{C \in \mathcal{P}} HX_C - HX_{0:n}$$

and Π is the set of all non-trivial partitions of $\{0 : n\}$.

API

`caekl_mutual_information`(**args*, ***kwargs*)

Calculates the Chan-AlBashabsheh-Ebrahimi-Kaced-Liu mutual information.

Parameters

- **`dist`** (`Distribution`) – The distribution from which the CAEKL mutual information is calculated.
- **`rvs`** (`list`, `None`) – A list of lists. Each inner list specifies the indexes of the random variables used to calculate the total correlation. If None, then the total correlation is calculated over all random variables, which is equivalent to passing *rvs=dist.rvs*.
- **`crvs`** (`list`, `None`) – A single list of indexes specifying the random variables to condition on. If None, then no variables are conditioned on.
- **`rv_mode`** (`str`, `None`) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **J** – The CAEKL mutual information.

Return type float

Examples

```
>>> d = dit.example_dists.Xor()
>>> dit.multivariate.caekl_mutual_information(d)
0.5
>>> dit.multivariate.caekl_mutual_information(d, rvs=[[0], [1]])
0.0
```

Raises `ditException` – Raised if *dist* is not a joint distribution or if *rvs* or *crvs* contain non-existant random variables.

Interaction Information

The interaction information is equal in magnitude to the *Co-Information*, but has the opposite sign when taken over an odd number of variables:

$$IIX_{0:n} = (-1)^n \cdot IX_{0:n}$$

Interaction information was first studied in the 3-variable case which, for $X_{0:3} = X_0X_1X_2$, takes the following form:

$$IIX_0 : X_1 : X_2 = IX_0 : X_1|X_2 - IX_0 : X_1$$

The extension to $n > 3$ proceeds recursively. For example,

$$\begin{aligned} IIX_0 : X_1 : X_2 : X_3 &= IIX_0 : X_1 : X_2|X_3 - IIX_0 : X_1 : X_2 \\ &= IX_0 : X_1|X_2, X_3 - IX_0 : X_1|X_3 \\ &\quad - IX_0 : X_1|X_2 + IX_0 : X_1 \end{aligned}$$

See also:

For more information, see *Co-Information*.

API

`interaction_information`(**args*, ***kwargs*)

Calculates the interaction information.

Parameters

- **dist** (*Distribution*) – The distribution from which the interaction information is calculated.
- **rvs** (*list, None*) – The indexes of the random variable used to calculate the interaction information between. If None, then the interaction information is calculated over all random variables.
- **crvs** (*list, None*) – The indexes of the random variables to condition on. If None, then no variables are condition on.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **II** – The interaction information.

Return type float

Raises `ditException` – Raised if *dist* is not a joint distribution or if *rvs* or *crvs* contain non-existant random variables.

DeWeese-like Measures

Mike DeWeese has introduced a family of multivariate information measures based on a multivariate extension of the data processing inequality. The general idea is the following: local modification of a single variable can not increase the amount of correlation or dependence it has with the other variables. Consider, however, the triadic distribution:

```
In [1]: from dit.example_dists import dyadic, triadic

In [2]: print(triadic)
Class:          Distribution
Alphabet:       ('0', '1', '2', '3') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 3
RV Names:       None

x     p(x)
000   1/8
022   1/8
111   1/8
133   1/8
202   1/8
220   1/8
313   1/8
331   1/8
```

This particular distribution has zero coinformation:

```
In [3]: from dit.multivariate import coinformation

In [4]: coinformation(triadic)
Out[4]: 0.0
```

Yet the distribution is a product of a giant bit (coinformation 1.0) and the xor (coinformation -1.0), and so there exists within it the capability of having a coinformation of 1.0 if the xor component were dropped. This is exactly what the DeWeese construction captures:

$$I_D X_0 : \ldots : X_n = \max_{p(x_i'|x_i)} I X_0' : \ldots : X_n'$$

```
In [5]: from dit.multivariate import deweese_coinformation

In [6]: deweese_coinformation(triadic)
Out[6]: 1.0000000000000004
```

DeWeese version of the total_correlation, dual_total_correlation, and caekl_mutual_information are also available, and operate on an arbitrary number of variables with optional conditional variables.

API

`deweese_coinformation`(**args*, ***kwargs*)

Compute the DeWeese coinformation.

Parameters

- **`dist`** (*`Distribution`*) – The distribution of interest.
- **`rvs`** (*`iter of iters, None`*) – The random variables of interest. If None, use all.
- **`crvs`** (*`iter, None`*) – The variables to condition on. If None, none.
- **`niter`** (*`int, None`*) – If specified, the number of optimization steps to perform.
- **`deterministic`** (*`bool`*) – Whether the functions to optimize over should be deterministic or not. Defaults to False.
- **`rv_mode`** (*`str, None`*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **val** – The value of the DeWeese coinformation.

Return type float

`deweese_total_correlation`(**args*, ***kwargs*)

Compute the DeWeese total correlation.

Parameters

- **`dist`** (*`Distribution`*) – The distribution of interest.
- **`rvs`** (*`iter of iters, None`*) – The random variables of interest. If None, use all.
- **`crvs`** (*`iter, None`*) – The variables to condition on. If None, none.
- **`niter`** (*`int, None`*) – If specified, the number of optimization steps to perform.
- **`deterministic`** (*`bool`*) – Whether the functions to optimize over should be deterministic or not. Defaults to False.
- **`rv_mode`** (*`str, None`*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **val** – The value of the DeWeese total correlation.

Return type float

`deweese_dual_total_correlation`(**args*, ***kwargs*)

Compute the DeWeese dual total correlation.

Parameters

- **`dist`** (*`Distribution`*) – The distribution of interest.
- **`rvs`** (*`iter of iters, None`*) – The random variables of interest. If None, use all.
- **`crvs`** (*`iter, None`*) – The variables to condition on. If None, none.
- **`niter`** (*`int, None`*) – If specified, the number of optimization steps to perform.

- **deterministic** (*bool*) – Whether the functions to optimize over should be deterministic or not. Defaults to False.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **val** – The value of the DeWeese dual total correlation.

Return type float

deweese_caekl_mutual_information(**args*, ***kwargs*)

Compute the DeWeese caekl mutual information.

Parameters

- **dist** (*Distribution*) – The distribution of interest.
- **rvs** (*iter of iters, None*) – The random variables of interest. If None, use all.
- **crvs** (*iter, None*) – The variables to condition on. If None, none.
- **niter** (*int, None*) – If specified, the number of optimization steps to perform.
- **deterministic** (*bool*) – Whether the functions to optimize over should be deterministic or not. Defaults to False.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **val** – The value of the DeWeese caekl mutual information.

Return type float

It is perhaps illustrative to consider how each of these measures behaves on two canonical distributions: the giant bit and parity.

	giant bit					parity				
size	I	II	T	B	J	I	II	T	B	J
2	1	1	1	1	1	1	1	1	1	1
3	1	-1	2	1	1	-1	1	1	2	$\frac{1}{2}$
4	1	1	3	1	1	1	1	1	3	$\frac{1}{3}$
5	1	-1	4	1	1	-1	1	1	4	$\frac{1}{4}$
n	1	$(-1)^n$	n	1	1	$(-1)^n$	1	1	n	$\frac{1}{n-1}$

Common Informations

These measures all somehow measure shared information, but do not equal the mutual information in the bivaraite case.

Gács-Körner Common Information

The Gács-Körner common information *[GacsKorner73]* take a very direct approach to the idea of common information. It extracts a random variable that is contained within each of the random variables under consideration.

The Common Information Game

Let's play a game. We have an n-variable joint distribution, and one player for each variable. Each player is given the probability mass function of the joint distribution then isolated from each other. Each round of the game the a joint outcome is generated from the distribution and each player is told the symbol that their particular variable took. The goal of the game is for the players to simultaneously write the same symbol on a piece of paper, and for the entropy of the players' symbols to be maximized. They must do this using only their knowledge of the joint random variable and the particular outcome of their marginal variable. The matching symbols produced by the players are called the *common random variable* and the entropy of that variable is the Gács-Körner common information, K.

Two Variables

Consider a joint distribution over X_0 and X_1. Given any particular outcome from that joint, we want a function $f(X_0)$ and a function $g(X_1)$ such that $\forall x_0 x_1 = X_0 X_1, f(x_0) = g(x_1) = v$. Of all possible pairs of functions $f(X_0) = g(X_1) = V$, there exists a "largest" one, and it is known as the common random variable. The entropy of that common random variable is the Gács-Körner common information:

$$\begin{aligned} K X_0 : X_1 &= \max_{f(X_0)=g(X_1)=V} H V \\ &= H X_0 \curlywedge X_1 \end{aligned}$$

As a canonical example, consider the following:

```
In [1]: from dit import Distribution as D

In [2]: from dit.multivariate import gk_common_information as K

In [3]: outcomes = ['00', '01', '10', '11', '22', '33']

In [4]: pmf = [1/8, 1/8, 1/8, 1/8, 1/4, 1/4]

In [5]: d = D(outcomes, pmf, sample_space=outcomes)

In [6]: K(d)
Out[6]: 1.5
```

Note: It is important that we set the *sample_space* argument. If it is *None* then the Cartesian product of each alphabet, and in such a case the meet will trivially be degenerate.

So, the Gács-Körner common information is 1.5 bits. But what is the common random variable?

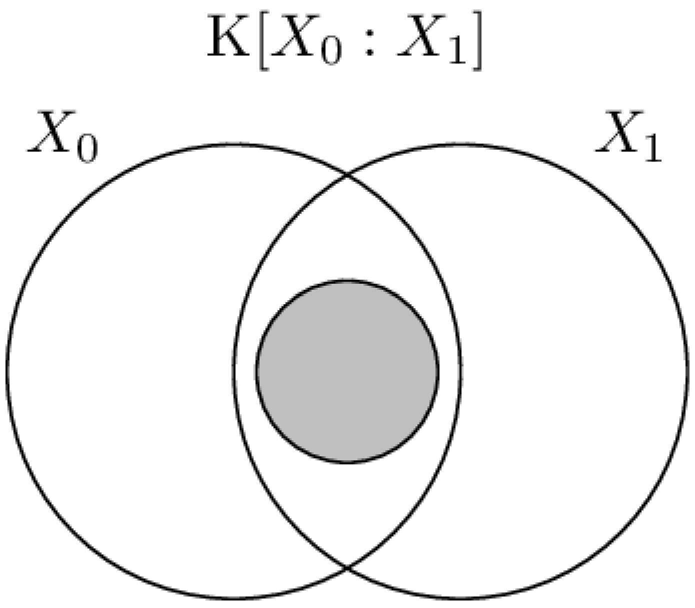
```
In [7]: from dit.algorithms import insert_meet

In [8]: crv = insert_meet(d, -1, [[0],[1]])

In [9]: print(crv)
Class:          Distribution
Alphabet:       (('0', '1', '2', '3'), ('0', '1', '2', '3'), ('0', '1', '2'))
Base:           linear
Outcome Class:  str
Outcome Length: 3
RV Names:       None

x     p(x)
```

```
002   1/8
012   1/8
102   1/8
112   1/8
220   1/4
331   1/4
```

Looking at the third index of the outcomes, we see that the common random variable maps 2 to 0 and 3 to 1, maintaining the information from those values. When X_0 or X_1 are either 0 or 1, however, it maps them to 2. This is because f and g must act independently: if x_0 is a 0 or a 1, there is no way to know if x_1 is a 0 or a 1 and vice versa. Therefore we aggregate 0s and 1s into 2.

Visualization

The Gács-Körner common information is the largest "circle" that entirely fits within the mutual information's "football":



Properties & Uses

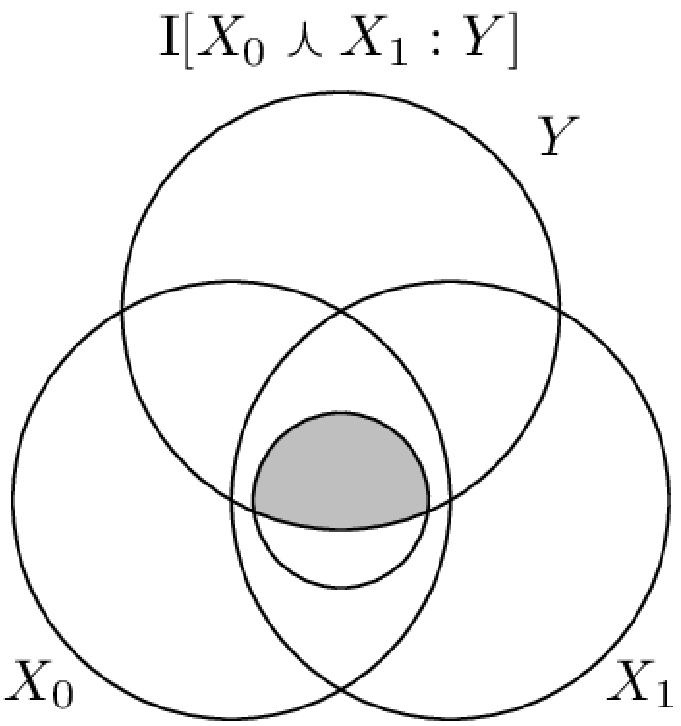
The Gács-Körner common information satisfies an important inequality:

$$0 \leq K X_0 : X_1 \leq I X_0 : X_1$$

One usage of the common information is as a measure of *redundancy* *[GCJ+14]*. Consider a function that takes two inputs, X_0 and X_1, and produces a single output Y. The output can be influenced redundantly by both inputs, uniquely from either one, or together they can synergistically influence the output. Determining how to compute the amount of redundancy is an open problem, but one proposal is:

$$I X_0 \curlywedge X_1 : Y$$

Which can be visualized as this:



This quantity can be computed easily using dit:

```
In [10]: from dit.example_dists import RdnXor

In [11]: from dit.shannon import mutual_information as I

In [12]: d = RdnXor()

In [13]: d = dit.pruned_samplespace(d)

In [14]: d = insert_meet(d, -1, [[0],[1]])

In [15]: I(d, [3], [2])
Out[15]: 1.0
```

n-Variables

With an arbitrary number of variables, the Gács-Körner common information *[TNG11]* is defined similarly:

$$\begin{aligned} K X_0 : \ldots : X_n &= \max_{\substack{V=f_0(X_0) \\ \vdots \\ V=f_n(X_n)}} H V \\ &= H X_0 \curlywedge \ldots \curlywedge X_n \end{aligned}$$

The common information is a monotonically decreasing function in the number of variables:

$$K X_0 : \ldots : X_{n-1} \geq K X_0 : \ldots : X_n$$

The multivariate common information follows a similar inequality as the two variable version:

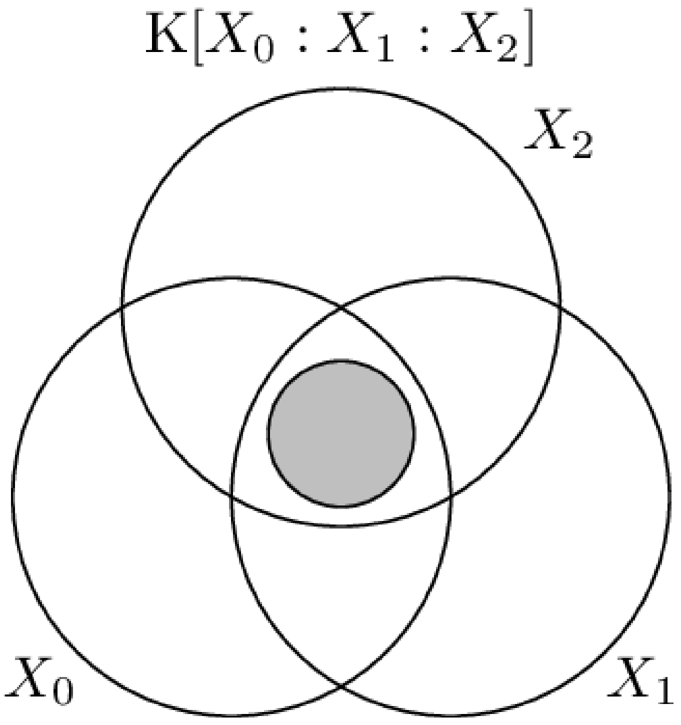
$$0 \leq K X_0 : \cdots : X_n \leq \min_{i,j \in \{0..n\}} I X_i : X_j$$

It is interesting to note that the Gács-Körner common information can be non-zero even when the coinformation is negative:

```
In [16]: from dit.example_dists.miscellaneous import gk_pos_i_neg

In [17]: from dit.multivariate import coinformation as I

In [18]: K(gk_pos_i_neg)
Out[18]: 0.5435644431995964

In [19]: I(gk_pos_i_neg)
Out[19]: -0.33143555680040304
```

Visualization

Here, as above, the Gács-Körner common information among three variables is the largest "circle" this time fiting in the vaguely triangular *Co-Information* region.



API

gk_common_information(**args*, ***kwargs*)
Calculates the Gacs-Korner common information K[X1:X2. . .] over the random variables in *rvs*.

Parameters

- **dist** (*Distribution*) – The distribution from which the common information is calculated.
- **rvs** (*list, None*) – The indexes of the random variables for which the Gacs-Korner common information is to be computed. If None, then the common information is calculated

over all random variables.

- **crvs** (*list, None*) – The indexes of the random variables to condition the common information by. If none, than there is no conditioning.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **K** – The Gacs-Korner common information of the distribution.

Return type float

Raises `ditException` – Raised if *rvs* or *crvs* contain non-existant random variables.

Wyner Common Information

The Wyner common information *[Wyn75][LXC10]* measures the minimum amount of information necessary needed to reconstruct a joint distribution from each marginal.

$$X_{0:n}|Y_{0:m} = \min_{\perp\!\!\!\perp X_{0:n}|Y_{0:m},V} I X_{0:n} : V|Y_{0:m}$$

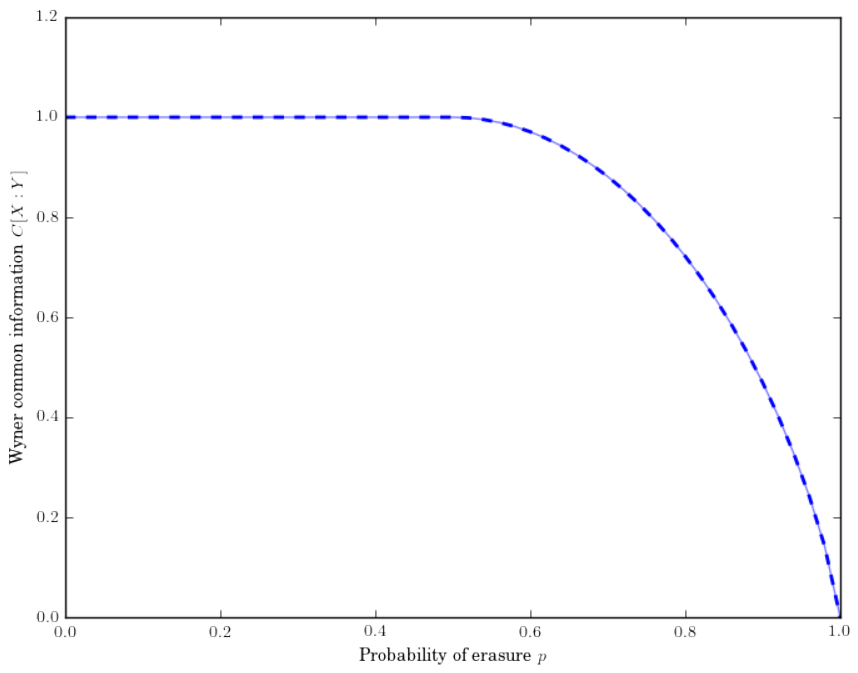
Binary Symmetric Erasure Channel

The Wyner common information of the binary symmetric erasure channel is known to be:

$$X : Y = \begin{cases} 1 & p < \frac{1}{2} \\ Hp & p \geq \frac{1}{2} \end{cases}.$$

We can verify this:

```
In [1]: from dit.multivariate import wyner_common_information as C

In [2]: ps = np.linspace(1e-6, 1-1e-6, 51)

In [3]: sbec = lambda p: dit.Distribution(['00', '0e', '1e', '11'], [(1-p)/2, p/2, p/
↪2, (1-p)/2])

In [4]: wci_true = [1 if p < 1/2 else dit.shannon.entropy(p) for p in ps]

In [5]: wci_opt = [C(sbec(p)) for p in ps]

In [6]: plt.plot(ps, wci_true, ls='-', alpha=0.5, c='b');

In [7]: plt.plot(ps, wci_opt, ls='--', lw=2, c='b');

In [8]: plt.xlabel(r'Probability of erasure $p$');

In [9]: plt.ylabel(r'Wyner common information $C[X:Y]$');

In [10]: plt.show()
```



API

`wyner_common_information`(**args*, ***kwargs*)

Computes the wyner common information, min I[X:V] such that V renders all X_i independent.

Parameters

- **`dist`** (*`Distribution`*) – The distribution for which the wyner common information will be computed.
- **`rvs`** (*`list, None`*) – A list of lists. Each inner list specifies the indexes of the random variables used to calculate the wyner common information. If None, then it calculated over all random variables, which is equivalent to passing *rvs=dist.rvs*.
- **`crvs`** (*`list, None`*) – A single list of indexes specifying the random variables to condition on. If None, then no variables are conditioned on.
- **`niter`** (*`int > 0`*) – Number of basin hoppings to perform during the optimization.
- **`maxiter`** (*`int > 0`*) – The number of iterations of the optimization subroutine to perform.
- **`polish`** (*`False, float`*) – Whether to polish the result or not. If a float, this will perform a second optimization seeded with the result of the first, but with smaller tolerances and probabilities below polish set to 0. If False, don't polish.
- **`bound`** (*`int`*) – Bound the size of the Markov variable.
- **`rv_mode`** (*`str, None`*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns ci – The wyner common information.

Return type float

Exact Common Information

The exact common information [KLEG14] is the entropy of the smallest variable V which renders all variables of interest independent:

$$X_{0:n}|Y_{0:m} = \min_{\perp\!\!\!\perp X_{0:n}|Y_{0:m}, V} HV|Y_{0:m}$$

Subadditivity of Independent Variables

Kumar **et. al.** [KLEG14] have shown that the exact common information of a pair of independent pairs of variables can be less than the sum of their individual exact common informations. Here we verify this claim:

```
In [1]: from dit.multivariate import exact_common_information as G

In [2]: d = dit.Distribution([(0,0), (0,1), (1,0)], [1/3]*3)

In [3]: d2 = d.__matmul__(d) # RTD doesn't use python 3

In [4]: print(d2)
Class:          Distribution
Alphabet:       (0, 1) for all rvs
Base:           linear
Outcome Class:  tuple
Outcome Length: 4
RV Names:       None

x              p(x)
(0, 0, 0, 0)   1/9
(0, 0, 0, 1)   1/9
(0, 0, 1, 0)   1/9
(0, 1, 0, 0)   1/9
(0, 1, 0, 1)   1/9
(0, 1, 1, 0)   1/9
(1, 0, 0, 0)   1/9
(1, 0, 0, 1)   1/9
(1, 0, 1, 0)   1/9

In [5]: 2*G(d)
Out[5]: 1.836582044884422

In [6]: G(d2, [[0, 2], [1, 3]])
Out[6]: 1.752667122516085
```

API

exact_common_information(**args*, ***kwargs*)

Computes the exact common information, min H[V] where V renders all *rvs* independent.

Parameters

- **dist** (*Distribution*) – The distribution for which the exact common information will be computed.
- **rvs** (*list, None*) – A list of lists. Each inner list specifies the indexes of the random variables used to calculate the exact common information. If None, then it calculated over all random variables, which is equivalent to passing *rvs=dist.rvs*.
- **crvs** (*list, None*) – A single list of indexes specifying the random variables to condition on. If None, then no variables are conditioned on.
- **niter** (*int > 0*) – Number of basin hoppings to perform during the optimization.
- **maxiter** (*int > 0*) – The number of iterations of the optimization subroutine to perform.
- **polish** (*False, float*) – Whether to polish the result or not. If a float, this will perform a second optimization seeded with the result of the first, but with smaller tolerances and probabilities below polish set to 0. If False, don't polish.
- **bound** (*int*) – Bound the size of the Markov variable.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **ci** – The exact common information.

Return type float

Functional Common Information

The functional common information captures the minimum amount of information neccessary to capture all of a distribution's share information using a function of that information. In other words:

$$FX_{0:n} \mid Y_{0:m} = \min_{\substack{\perp\!\!\!\perp X_{0:n}|Y_{0:m},W \\ W=f(X_{0:n},Y_{0:m})}} HW$$

Relationship To Other Measures of Common Information

Since this is an additional constraint on the Exact common information, it is generally larger than it, and since its constraint is weaker than that of the *MSS Common Information*, it is generally less than it:

$$X_{0:n} \leq FX_{0:n} \leq MX_{0:n}$$

API

functional_common_information(**args*, ***kwargs*)

Compute the functional common information, F, of *dist*. It is the entropy of the smallest random variable W such that all the variables in *rvs* are rendered independent conditioned on W, and W is a function of *rvs*.

Parameters

- **dist** (*Distribution*) – The distribution from which the functional common information is computed.

- **rvs** (*list, None*) – A list of lists. Each inner list specifies the indexes of the random variables used to calculate the total correlation. If None, then the total correlation is calculated over all random variables, which is equivalent to passing *rvs=dist.rvs*.
- **crvs** (*list, None*) – A single list of indexes specifying the random variables to condition on. If None, then no variables are conditioned on.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **F** – The functional common information.

Return type float

MSS Common Information

The Minimal Sufficient Statistic Common Information is the entropy of the join of the minimal sufficient statistic of each variable about the others:

$$MX_{0:n} = H\curlyvee_i \left(X_i \searrow X_{\overline{\{i\}}}\right)$$

The distribution that the MSS common information is the entroy of is also known "information trim" of the original distribution, and is accessable via `dit.algorithms.minimal_sufficient_statistic.info_trim()`.

API

`mss_common_information`(**args, **kwargs*)

Compute the minimal sufficient statistic common information, which is the entropy of the join of the minimal sufficent statistic of each variable about the others.

Parameters

- **dist** (*Distribution*) – The distribution for which the joint minimal sufficient statistic is computed.
- **rvs** (*list, None*) – The random variables to compute the joint minimal sufficient statistic of. If None, all random variables are used.
- **crvs** (*list, None*) – The random variables to condition the joint minimal sufficient statistic on. If None, then no random variables are conditioned on.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Ordering

The common information measures (together with the *Dual Total Correlation* and *CAEKL Mutual Information*) form an ordering:

$$KX_{0:n} \leq JX_{0:n} \leq BX_{0:n} \leq X_{0:n} \leq X_{0:n} \leq FX_{0:n} \leq MX_{0:n}$$

Others

These measures quantify other aspects of a joint distribution.

Residual Entropy

The residual entropy, or erasure entropy, is a dual to the *Dual Total Correlation*. It is dual in the sense that together they form the entropy of the distribution.

$$
\begin{aligned}
RX_{0:n} &= \sum HX_i | X_{\{0..n\}/i} \\
&= - \sum_{x_{0:n} \in X_{0:n}} p(x_{0:n}) \log_2 \prod p(x_i | x_{\{0:n\}/i})
\end{aligned}
$$

The residual entropy was originally proposed in *[VW08]* to quantify the information lost by sporatic erasures in a channel. The idea here is that only the information uncorrelated with other random variables is lost if that variable is erased.

If a joint distribution consists of independent random variables, the residual entropy is equal to the *Entropy*:

```
In [1]: from dit.multivariate import entropy, residual_entropy

In [2]: d = dit.uniform_distribution(3, 2)

In [3]: entropy(d) == residual_entropy(d)
Out[3]: True
```

Another simple example is a distribution where one random variable is independent of the others:

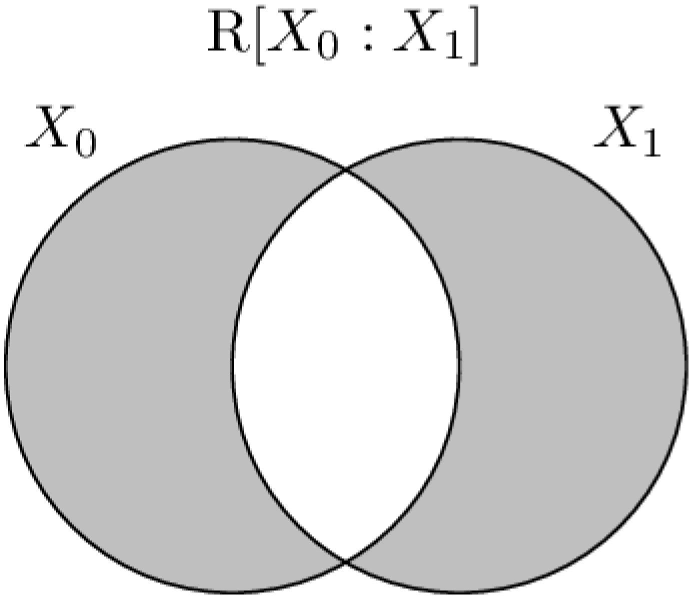
```
In [4]: d = dit.uniform(['000', '001', '110', '111'])

In [5]: residual_entropy(d)
Out[5]: 1.0
```

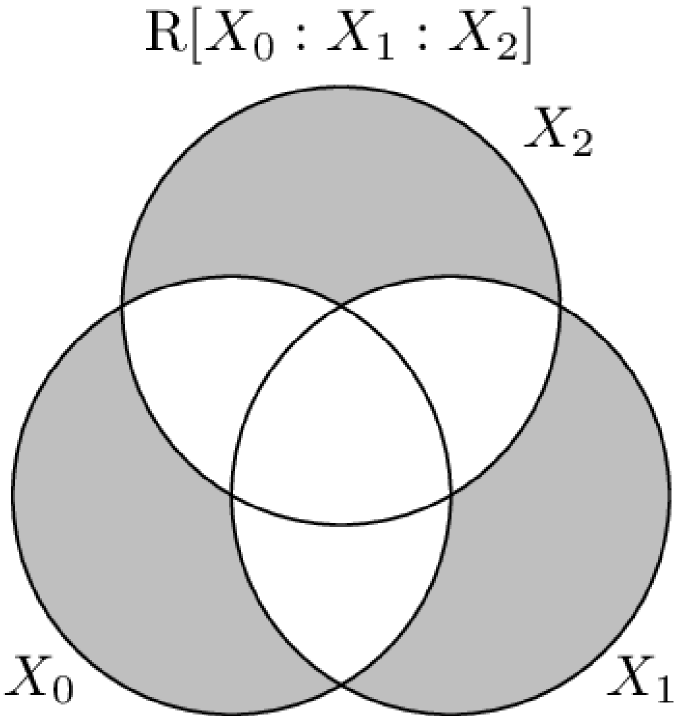
If we ask for the residual entropy of only the latter two random variables, the middle one is now independent of the others and so the residual entropy grows:

```
In [6]: residual_entropy(d, [[1], [2]])
Out[6]: 2.0
```

Visualization

The residual entropy consists of all the unshared information in the distribution. That is, it is the information in each variable not overlapping with any other.





API

residual_entropy(**args*, ***kwargs*)
Compute the residual entropy.

Parameters

- **dist** (`Distribution`) – The distribution from which the residual entropy is calculated.
- **rvs** (`list`, `None`) – The indexes of the random variable used to calculate the residual entropy. If None, then the total correlation is calculated over all random variables.

- **crvs** (*list, None*) – The indexes of the random variables to condition on. If None, then no variables are condition on.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **R** – The residual entropy.

Return type float

Raises `ditException` – Raised if *dist* is not a joint distribution or if *rvs* or *crvs* contain non-existant random variables.

TSE Complexity

The Tononi-Sporns-Edelmans (TSE) complexity *[TSE94]* is a complexity measure for distributions. It is designed so that it maximized by distributions where small subsets of random variables are loosely coupled but the overall distribution is tightly coupled.

$$TSEX|Z = \sum_{k=1}^{|X|} \left(\binom{N}{k}^{-1} \sum_{\substack{y \subseteq X \\ |y|=k}} (Hy|Z) - \frac{k}{|X|} HX|Z \right)$$

Two distributions which might be considered tightly coupled are the "giant bit" and the "parity" distributions:

```
In [1]: from dit.multivariate import tse_complexity

In [2]: from dit.example_dists import Xor

In [3]: d1 = Xor()

In [4]: tse_complexity(d1)
Out[4]: 1.0

In [5]: d2 = dit.Distribution(['000', '111'], [1/2, 1/2])

In [6]: tse_complexity(d2)
Out[6]: 1.0
```

The TSE Complexity assigns them both a value of 1.0 bits, which is the maximal value the TSE takes over trivariate, binary alphabet distributions.

API

tse_complexity(**args*, ***kwargs*)
Calculates the TSE complexity.

Parameters

- **dist** (*Distribution*) – The distribution from which the TSE complexity is calculated.
- **rvs** (*list, None*) – The indexes of the random variable used to calculate the TSE complexity between. If None, then the TSE complexity is calculated over all random variables.

- **crvs** (*list, None*) – The indexes of the random variables to condition on. If None, then no variables are condition on.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **TSE** – The TSE complexity.

Return type float

Raises `ditException` – Raised if *dist* is not a joint distribution or if *rvs* or *crvs* contain non-existant random variables.

Necessary Conditional Entropy

The necessary conditional entropy *[CPC10]* quantifies the amount of information that a random variable X necessarily must carry above and beyond the mutual information $IX : Y$ to actually contain that mutual information:

$$HX \dagger Y = HX \searrow Y|Y$$

API

necessary_conditional_entropy (**args*, ***kwargs*)

Calculates the necessary conditional entropy $H[X \dagger Y]$. This is the entropy of the minimal sufficient statistic of X about Y, given Y.

Parameters

- **dist** (*Distribution*) – The distribution from which the necessary conditional entropy is calculated.
- **rvs** (*list, None*) – The indexes of the random variable used to calculate the necessary conditional entropy. If None, then the entropy is calculated over all random variables.
- **crvs** (*list, None*) – The indexes of the random variables to condition on. If None, then no variables are conditioned on.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **H** – The necessary conditional entropy.

Return type float

Raises `ditException` – Raised if *rvs* or *crvs* contain non-existant random variables.

Example

This next group of measures can not be represented on information diagrams, and can not really be directly compared to the measures above:

1.7.3 Other Measures

Other measures of information. These are generally based around alternatives to the Shannon entropy proposed for a variety of reasons.

Cumulative Residual Entropy

The cumulative residual entropy *[RCVW04]* is an alternative to the differential Shannon entropy. The differential entropy has many issues, including that it can be negative even for simple distributions such as the uniform distribution; and that if one takes discrete estimates that limit to the continuous distribution, the discrete entropy does not limit to the differential (continuous) entropy. It also attempts to provide meaningful differences between numerically different random variables, such as a die labeled [1, 2, 3, 4, 5, 6] and one lebeled [1, 2, 3, 4, 5, 100].

Note: The Cumulative Residual Entropy is unrelated to *Residual Entropy*.

$$\mathcal{E}X = -\int_0^\infty p(|X| > x)\log_2 p(|X| > x)dx$$

```
In [1]: from dit.other import cumulative_residual_entropy

In [2]: d1 = dit.ScalarDistribution([1, 2, 3, 4, 5, 6], [1/6]*6)

In [3]: d2 = dit.ScalarDistribution([1, 2, 3, 4, 5, 100], [1/6]*6)

In [4]: cumulative_residual_entropy(d1)
Out[4]: 2.068318255702844

In [5]: cumulative_residual_entropy(d2)
Out[5]: 22.672680046016705
```

Generalized Cumulative Residual Entropy

The genearlized form of the cumulative residual entropy integrates over the intire set of reals rather than just the positive ones:

$$\mathcal{E}'X = -\int_{-\infty}^\infty p(X > x)\log_2 p(X > x)dx$$

```
In [6]: from dit.other import generalized_cumulative_residual_entropy

In [7]: generalized_cumulative_residual_entropy(d1)
Out[7]: 2.068318255702844

In [8]: d3 = dit.ScalarDistribution([-2, -1, 0, 1, 2], [1/5]*5)

In [9]: cumulative_residual_entropy(d3)
Out[9]: 0.9065649754771961

In [10]: generalized_cumulative_residual_entropy(d3)
Out[10]: 1.6928786893420307
```

Conditional Cumulative Residual Entropy

The conditional cumulative residual entropy $\mathcal{E}[X|Y]$ is a distribution with the same probability mass function as Y, and the outcome associated with $p(y)$ is equal to the cumulative residual entropy over probabilities conditioned on $Y = y$. In this sense the conditional cumulative residual entropy is more akin to a distribution over $H[X|Y = y]$ than the single scalar quantity $H[X|Y]$.

$$\mathcal{E}X|Y = -\int_{0}^{\infty} p(|X| > x|Y) \log_2 p(|X| > x|Y) dx$$

Conditional Generalized Cumulative Residual Entropy

Conceptually the conditional generalized cumulative residual entropy is the same as the non-generalized form, but integrated over the entire real line rather than just the positive:

$$\mathcal{E}'X|Y = -\int_{-\infty}^{\infty} p(X > x|Y) \log_2 p(X > x|Y) dx$$

API

`cumulative_residual_entropy`(*dist*, *extract=False*)

The cumulative residual entropy is an alternative to the Shannon differential entropy with several desirable properties including non-negativity.

Parameters

- **dist** (*Distribution*) – The distribution to compute the cumulative residual entropy of each index for.
- **extract** (*bool*) – If True and *dist.outcome_length()* is 1, return the single GCRE value rather than a length-1 array.

Returns **CREs** – The cumulative residual entropy for each index.

Return type ndarray

Examples

```
>>> d1 = ScalarDistribution([1, 2, 3, 4, 5, 6], [1/6]*6)
>>> d2 = ScalarDistribution([1, 2, 3, 4, 5, 100], [1/6]*6)
>>> cumulative_residual_entropy(d1)
2.0683182557028439
>>> cumulative_residual_entropy(d2)
22.672680046016705
```

`generalized_cumulative_residual_entropy`(*dist*, *extract=False*)

The generalized cumulative residual entropy is a generalized from of the cumulative residual entropy. Rather than integrating from 0 to infinity over the absolute value of the CDF.

Parameters

- **dist** (*Distribution*) – The distribution to compute the generalized cumulative residual entropy of each index for.
- **extract** (*bool*) – If True and *dist.outcome_length()* is 1, return the single GCRE value rather than a length-1 array.

Returns **GCREs** – The generalized cumulative residual entropy for each index.

Return type ndarray

Examples

```
>>> generalized_cumulative_residual_entropy(uniform(-2, 3))
1.6928786893420307
>>> generalized_cumulative_residual_entropy(uniform(0, 5))
1.6928786893420307
```

Conditional Forms

conditional_cumulative_residual_entropy(*dist*, *rv*, *crvs=None*, *rv_mode=None*)

Returns the conditional cumulative residual entropy.

Parameters

- **dist** (*Distribution*) – The distribution to compute the conditional cumulative residual entropy of.
- **rv** (*list, None*) – The possibly joint random variable to compute the conditional cumulative residual entropy of. If *None*, then all variables not in *crvs* are used.
- **crvs** (*list, None*) – The random variables to condition on. If *None*, nothing is conditioned on.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **CCRE** – The conditional cumulative residual entropy.

Return type ScalarDistribution

Examples

```
>>> from itertools import product
>>> events = [ (a, b) for a, b, in product(range(5), range(5)) if a <= b ]
>>> probs = [ 1/(5-a)/5 for a, b in events ]
>>> d = Distribution(events, probs)
>>> print(conditional_cumulative_residual_entropy(d, 1, [0]))
Class:    ScalarDistribution
Alphabet: (-0.0, 0.5, 0.91829583405448956, 1.3112781244591329, 1.6928786893420307)
Base:     linear
```

x p(x) -0.0 0.2 0.5 0.2 0.918295834054 0.2 1.31127812446 0.2 1.69287868934 0.2

conditional_generalized_cumulative_residual_entropy(*dist*, *rv*, *crvs=None*, *rv_mode=None*)

Returns the conditional cumulative residual entropy.

Parameters

- **dist** (*Distribution*) – The distribution to compute the conditional generalized cumulative residual entropy of.
- **rv** (*list, None*) – The possibly joint random variable to compute the conditional generalized cumulative residual entropy of. If *None*, then all variables not in *crvs* are used.
- **crvs** (*list, None*) – The random variables to condition on. If *None*, nothing is conditioned on.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **CCRE** – The conditional cumulative residual entropy.

Return type ScalarDistribution

Examples

```
>>> from itertools import product
>>> events = [ (a-2, b-2) for a, b, in product(range(5), range(5)) if a <= b ]
>>> probs = [ 1/(3-a)/5 for a, b in events ]
>>> d = Distribution(events, probs)
>>> print(conditional_generalized_cumulative_residual_entropy(d, 1, [0]))
Class:    ScalarDistribution
Alphabet: (-0.0, 0.5, 0.91829583405448956, 1.3112781244591329, 1.6928786893420307)
Base:     linear
```

x p(x) -0.0 0.2 0.5 0.2 0.918295834054 0.2 1.31127812446 0.2 1.69287868934 0.2

Disequilibrium and the LMPR Complexity

Lamberti, Martin, Plastino, and Rosso have proposed a complexity measure *[LMPR04]* disigned around the idea of being a measure of "distance from equilibrium", or disequilibrium, multiplied by a measure of "randomness". Here, they measure "randomness" by the (normalized) *Entropy*:

$$HX/\log_2|X|$$

and the disequilibrium as a (normalized) *Jensen-Shannon Divergence*:

$$D_{JS}X||P_e/Q_0$$

where P_e is a uniform distribution over the same outcome space as X, and Q_0 is the maximum possible value of the Jensen-Shannon divergence of a distribution with P_e.

The LMPR complexity does not necessarily behave as one might intuitively hope. For example, the LMPR complexity of the `xor` and "double bit" with independent bit are identical:

```
In [1]: from dit.other.disequilibrium import *

In [2]: d1 = dit.Distribution(['000', '001', '110', '111'], [1/4]*4)

In [3]: d2 = dit.Distribution(['000', '011', '101', '110'], [1/4]*4)

In [4]: LMPR_complexity(d1)
```

```
Out[4]: 0.2894598616025801

In [5]: LMPR_complexity(d2)
Out[5]: 0.2894598616025801
```

This is because they are both equally "far from equilibrium" with four equiprobable events over the space of three binary variables, and both have the same entropy of two bits.

This implies that the LMPR complexity is perhaps best applied to a `ScalarDistribution`, and is not suitable for measuring the complexity of dependencies between variables.

API

`disequilibrium`(*dist*, *rvs=None*, *rv_mode=None*)

Compute the (normalized) disequilibrium as measured the Jensen-Shannon divergence from an equilibrium distribution.

Parameters

- **dist** (*Distribution*) – Distribution to compute the disequilibrium of.
- **rvs** (*list, None*) – The indexes of the random variable used to calculate the diseqilibrium. If None, then the disequilibrium is calculated over all random variables. This should remain *None* for ScalarDistributions.
- **rv_mode** (*str, None*) – Specifies how to interpret the elements of *rvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted.

Returns **D** – The disequilibrium.

Return type float

`LMPR_complexity`(*dist*, *rvs=None*, *rv_mode=None*)

Compute the LMPR complexity.

Parameters

- **dist** (*Distribution*) – Distribution to compute the LMPR complexity of.
- **rvs** (*list, None*) – The indexes of the random variable used to calculate the LMPR complexity. If None, then the LMPR complexity is calculated over all random variables. This should remain *None* for ScalarDistributions.
- **rv_mode** (*str, None*) – Specifies how to interpret the elements of *rvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted.

Returns **C** – The LMPR complexity.

Return type float

Extropy

The extropy *[LSAgro11]* is a dual to the *Entropy*. It is defined by:

$$XX = -\sum_{x \in X} (1 - p(x)) \log_2 (1 - p(x))$$

The entropy and the extropy satisify the following relationship:

$$HX + XX = \sum_{x \in \mathcal{X}} Hp(x), 1 - p(x) = \sum_{x \in \mathcal{X}} Xp(x), 1 - p(x)$$

Unfortunately, the extropy does not yet have any intuitive interpretation.

```
In [1]: from dit.other import extropy

In [2]: from dit.example_dists import Xor

In [3]: extropy(Xor())
Out[3]: 1.2451124978365313

In [4]: extropy(Xor(), [0])
Out[4]: 1.0
```

API

extropy (*dist*, *rvs=None*, *rv_mode=None*)

Returns the extropy J[X] over the random variables in *rvs*.

If the distribution represents linear probabilities, then the extropy is calculated with units of 'bits' (base-2).

Parameters

- **dist** (`Distribution or float`) – The distribution from which the extropy is calculated. If a float, then we calculate the binary extropy.
- **rvs** (`list, None`) – The indexes of the random variable used to calculate the extropy. If None, then the extropy is calculated over all random variables. This should remain *None* for ScalarDistributions.
- **rv_mode** (`str, None`) – Specifies how to interpret the elements of *rvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted.

Returns **J** – The extropy of the distribution.

Return type float

Lautum Information

The lautum information *[PVerdu08]* is, in a sense, the mutual information in reverse (*lautum* is *mutual* backwards):

$$X_{0:n} = D_{KL} X_0 \cdot X_1 \cdot \ldots \cdot X_n || X_{0:n}$$

API

lautum_information (*dist*, *rvs=None*, *crvs=None*, *rv_mode=None*)

Computes the lautum information.

Parameters

- **dist** (`Distribution`) – The distribution from which the lautum information is calculated.

- **rvs** (*list, None*) – A list of lists. Each inner list specifies the indexes of the random variables used to calculate the lautum information. If None, then the lautum information is calculated over all random variables, which is equivalent to passing *rvs=dist.rvs*.
- **crvs** (*list, None*) – A single list of indexes specifying the random variables to condition on. If None, then no variables are conditioned on.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **L** – The lautum information.

Return type float

Examples

```
>>> outcomes = ['000', '001', '010', '011', '100', '101', '110', '111']
>>> pmf = [3/16, 1/16, 1/16, 3/16, 1/16, 3/16, 3/16, 1/16]
>>> d = dit.Distribution(outcomes, pmf)
>>> dit.other.lautum_information(d)
0.20751874963942196
>>> dit.other.lautum_information(d, rvs=[[0], [1]])
0.0
```

Raises `ditException` – Raised if *dist* is not a joint distribution or if *rvs* or *crvs* contain non-existent random variables.

Perplexity

The perplexity is a trivial measure to make the *Entropy* more intuitive:

$$PX = 2^{HX}$$

The perplexity of a random variable is the size of a uniform distribution that would have the same entropy. For example, a distribution with 2 bits of entropy has a perplexity of 4, and so could be said to be "as random" as a four-sided die.

The conditional perplexity is defined in the natural way:

$$PX|Y = 2^{HX|Y}$$

We can see that the *xor* distribution is "4-way" perplexed:

```
In [1]: from dit.other import perplexity

In [2]: from dit.example_dists import Xor

In [3]: perplexity(Xor())
Out[3]: 4.0
```

API

perplexity(*dist*, *rvs=None*, *crvs=None*, *rv_mode=None*)

Parameters

- **dist** (*Distribution*) – The distribution from which the perplexity is calculated.
- **rvs** (*list, None*) – The indexes of the random variable used to calculate the perplexity. If None, then the perpelxity is calculated over all random variables.
- **crvs** (*list, None*) – The indexes of the random variables to condition on. If None, then no variables are condition on.
- **rv_mode** (*str, None*) – Specifies how to interpret the elements of *rvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted.

Returns **P** – The perplexity.

Return type float

Rényi Entropy

The Rényi entropy is a spectrum of generalizations to the Shannon *Entropy*:

$$H_\alpha X = \frac{1}{1-\alpha} \log_2 \left(\sum_{x \in \mathcal{X}} p(x)^\alpha \right)$$

```
In [1]: from dit.other import renyi_entropy

In [2]: from dit.example_dists import binomial

In [3]: d = binomial(15, 0.4)

In [4]: renyi_entropy(d, 3)
Out[4]: 2.6611840717104625
```

Special Cases

For several values of α, the Rényi entropy takes on particular values.

$\alpha = 0$

When $\alpha = 0$ the Rényi entropy becomes what is known as the Hartley entropy:

$$H_0 X = \log_2 |X|$$

```
In [5]: renyi_entropy(d, 0)
Out[5]: 4.0
```

$\alpha = 1$

When $\alpha = 1$ the Rényi entropy becomes the standard Shannon entropy:

$$H_1 X = H X$$

```
In [6]: renyi_entropy(d, 1)
Out[6]: 2.9688513169509623
```

$\alpha = 2$

When $\alpha = 2$, the Rényi entropy becomes what is known as the collision entropy:

$$H_2 X = -\log_2 p(X = Y)$$

where Y is an IID copy of X. This is basically the surprisal of "rolling doubles"

```
In [7]: renyi_entropy(d, 2)
Out[7]: 2.7607270851693615
```

$\alpha = \infty$

Finally, when $\alpha = \infty$ the Rényi entropy picks out the probability of the most-probable event:

$$H_\infty X = -\log_2 \max_{x \in \mathcal{X}} p(x)$$

```
In [8]: renyi_entropy(d, np.inf)
Out[8]: 2.275104563096674
```

General Properies

In general, the Rényi entropy is a monotonically decreasing function in α:

$$H_\alpha X \geq H_\beta X, \quad \beta > \alpha$$

Further, the following inequality holds in the other direction:

$$H_2 X \leq 2 \cdot H_\infty X$$

API

`renyi_entropy`(*dist*, *order*, *rvs=None*, *rv_mode=None*)

Compute the Renyi entropy of order *order*.

Parameters

- **`dist`** (*`Distribution`*) – The distribution to take the Renyi entropy of.
- **`order`** (*`float >= 0`*) – The order of the Renyi entropy.

- **rvs** (*list, None*) – The indexes of the random variable used to calculate the Renyi entropy of. If None, then the Renyi entropy is calculated over all random variables.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **H_a** – The Renyi entropy.

Return type float

Raises

- `ditException` – Raised if *rvs* or *crvs* contain non-existant random variables.
- `ValueError` – Raised if *order* is not a non-negative float.

Tsallis Entropy

The Tsallis entropy is a generalization of the Shannon (or Boltzmann-Gibbs) entropy to the case where entropy is nonextensive. It is given by:

$$S_q X = \frac{1}{q-1}\left(1 - \sum_{x \in \mathcal{X}} p(x)^q\right)$$

```
In [1]: from dit.other import tsallis_entropy

In [2]: from dit.example_dists import n_mod_m

In [3]: d = n_mod_m(4, 3)

In [4]: tsallis_entropy(d, 4)
Out[4]: 0.3333163982455249
```

Non-additivity

One interesting property of the Tsallis entropy is the relationship between the joint Tsallis entropy of two indpendent systems, and the Tsallis entropy of those subsystems:

$$S_q X, Y = S_q X + S_q Y + (1-q) S_q X S_q Y$$

API

tsallis_entropy(*dist*, *order*, *rvs=None*, *rv_mode=None*)

Compute the Tsallis entropy of order *order*.

Parameters

- **dist** (*Distribution*) – The distribution to take the Tsallis entropy of.
- **order** (*float >= 0*) – The order of the Tsallis entropy.
- **rvs** (*list, None*) – The indexes of the random variable used to calculate the Tsallis entropy of. If None, then the Tsallis entropy is calculated over all random variables.

- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns S_q – The Tsallis entropy.

Return type float

Raises

- ditException – Raised if *rvs* or *crvs* contain non-existant random variables.
- ValueError – Raised if *order* is not a non-negative float.

There are also measures of "distance" or divergence between two (and im some cases, more) distribution:

1.7.4 Divergences

Divergences are measures of comparison between distributions:

Cross Entropy

The cross entropy between two distributions $p(x)$ and $q(x)$ is given by:

$$xHp||q = -\sum_{x\in\mathcal{X}} p(x)\log_2 q(x)$$

This quantifies the average cost of representing a distribution defined by the probabilities $p(x)$ using the probabilities $q(x)$. For example, the cross entropy of a distribution with itself is the entropy of that distribion because the entropy quantifies the average cost of representing a distribution:

```
In [1]: from dit.divergences import cross_entropy

In [2]: p = dit.Distribution(['0', '1'], [1/2, 1/2])

In [3]: cross_entropy(p, p)
Out[3]: 1.0
```

If, however, we attempted to model a fair coin with a biased on, we could compute this mis-match with the cross entropy:

```
In [4]: q = dit.Distribution(['0', '1'], [3/4, 1/4])

In [5]: cross_entropy(p, q)
Out[5]: 1.207518749639422
```

Meaning, we will on average use about 1.2 bits to represent the flips of a fair coin. Turning things around, what if we had a biased coin that we attempted to represent with a fair coin:

```
In [6]: cross_entropy(q, p)
Out[6]: 1.0
```

So although the entropy of q is less than 1, we will use a full bit to represent its outcomes. Both of these results can easily be seen by considering the following identity:

$$xHp||q = Hp + D_{KL}p||q$$

So in representing p using q, we of course must at least use Hp bits – the minimum required to represent p – plus the Kullback-Leibler divergence of q from p.

API

cross_entropy(**args*, ***kwargs*)
The cross entropy between *dist1* and *dist2*.

Parameters

- **dist1** (*Distribution*) – The first distribution in the cross entropy.
- **dist2** (*Distribution*) – The second distribution in the cross entropy.
- **rvs** (*list, None*) – The indexes of the random variable used to calculate the cross entropy between. If None, then the cross entropy is calculated over all random variables.
- **rv_mode** (*str, None*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **xh** – The cross entropy between *dist1* and *dist2*.

Return type float

Raises `ditException` – Raised if either *dist1* or *dist2* doesn't have *rvs* or, if *rvs* is None, if *dist2* has an outcome length different than *dist1*.

Kullback-Leibler Divergence

The Kullback-Leibler divergence, sometimes also called the *relative entropy*, of a distribution p from a distribution q is defined as:

$$D_{KL}p||q = \sum_{x \in \mathcal{X}} p(x) \log_2 \frac{p(x)}{q(x)}$$

The Kullback-Leibler divergence quantifies the average number of *extra* bits required to represent a distribution p when using an arbitrary distribution q. This can be seen through the following identity:

$$D_{KL}p||q = xHp||q - Hp$$

Where the *Cross Entropy* quantifies the total cost of encoding p using q, and the *Entropy* quantifies the true, minimum cost of encoding p. For example, let's consider the cost of representing a biased coin by a fair one:

```
In [1]: from dit.divergences import kullback_leibler_divergence

In [2]: p = dit.Distribution(['0', '1'], [3/4, 1/4])

In [3]: q = dit.Distribution(['0', '1'], [1/2, 1/2])

In [4]: kullback_leibler_divergence(p, q)
Out[4]: 0.18872187554086717
```

That is, it costs us 0.1887 bits of wasted overhead by using a mismatched distribution.

Not a Metric

Although the Kullback-Leibler divergence is often used to see how "different" two distributions are, it is not a metric. Importantly, it is neither symmetric nor does it obey the triangle inequality. It does, however, have the following property:

$$D_{KL}p||q \geq 0$$

with equality if and only if $p = q$. This makes it a premetric.

API

`kullback_leibler_divergence`(*dist1*, *dist2*, *rvs=None*, *crvs=None*, *rv_mode=None*)
The Kullback-Liebler divergence between *dist1* and *dist2*.

Parameters

- **`dist1`** (*`Distribution`*) – The first distribution in the Kullback-Leibler divergence.
- **`dist2`** (*`Distribution`*) – The second distribution in the Kullback-Leibler divergence.
- **`rvs`** (*`list, None`*) – The indexes of the random variable used to calculate the Kullback-Leibler divergence between. If None, then the Kullback-Leibler divergence is calculated over all random variables.
- **`rv_mode`** (*`str, None`*) – Specifies how to interpret *rvs* and *crvs*. Valid options are: {'indices', 'names'}. If equal to 'indices', then the elements of *crvs* and *rvs* are interpreted as random variable indices. If equal to 'names', the the elements are interpreted as random variable names. If *None*, then the value of *dist._rv_mode* is consulted, which defaults to 'indices'.

Returns **dkl** – The Kullback-Leibler divergence between *dist1* and *dist2*.

Return type float

Raises `ditException` – Raised if either *dist1* or *dist2* doesn't have *rvs* or, if *rvs* is None, if *dist2* has an outcome length different than *dist1*.

Jensen-Shannon Divergence

The Jensen-Shannon divergence is a principled divergence measure which is always finite for finite random variables. It quantifies how "distinguishable" two or more distributions are from each other. In its basic form it is:

$$D_{JS}X||Y = H\frac{X+Y}{2} - \frac{HX + HY}{2}$$

That is, it is the entropy of the mixture minus the mixture of the entropy. This can be generalized to an arbitrary number of random variables with arbitrary weights:

$$D_{JS}X_{0:n} = H\sum w_i X_i - \sum (w_i H X_i)$$

```
In [1]: from dit.divergences import jensen_shannon_divergence

In [2]: X = dit.ScalarDistribution(['red', 'blue'], [1/2, 1/2])

In [3]: Y = dit.ScalarDistribution(['blue', 'green'], [1/2, 1/2])
In [4]: jensen_shannon_divergence([X, Y])
Out[4]: 0.5

In [5]: jensen_shannon_divergence([X, Y], [3/4, 1/4])
Out[5]: 0.40563906222956647

In [6]: Z = dit.ScalarDistribution(['blue', 'yellow'], [1/2, 1/2])

In [7]: jensen_shannon_divergence([X, Y, Z])
Out[7]: 0.7924812503605778

In [8]: jensen_shannon_divergence([X, Y, Z], [1/2, 1/4, 1/4])
Out[8]: 0.75
```

Derivation

Where does this equation come from? Consider Jensen's inequality:

$$\Psi\left(\mathbb{E}(x)\right) \geq \mathbb{E}\left(\Psi(x)\right)$$

where Ψ is a concave function. If we consider the *divergence* of the left and right side we find:

$$\Psi\left(\mathbb{E}(x)\right) - \mathbb{E}\left(\Psi(x)\right) \geq 0$$

If we make that concave function Ψ the Shannon entropy H, we get the Jensen-Shannon divergence. Jensen from Jensen's inequality, and Shannon from the use of the Shannon entropy.

Note: Some people look at the Jensen-Rényi divergence (where Ψ is the *Rényi Entropy*) and the Jensen-Tsallis divergence (where Ψ is the *Tsallis Entropy*).

Metric

The square root of the Jensen-Shannon divergence, $\sqrt{D_{JS}}$, is a true metric between distributions.

Relationship to the Other Measures

The Jensen-Shannon divergence can be derived from other, more well known information measures; notably the *Kullback-Leibler Divergence* and the *Mutual Information*.

Kullback-Leibler divergence

The Jensen-Shannon divergence is the average Kullback-Leibler divergence of X and Y from their mixture distribution, M:

$$D_{JS}X||Y = \frac{1}{2}\left(D_{KL}X||M + D_{KL}Y||M\right)$$
$$M = \frac{X+Y}{2}$$

Mutual Information

$$D_{JS} X || Y = I Z : M$$

where M is the mixture distribution as before, and Z is an indicator variable over X and Y. In essence, if X and Y are each an urn containing colored balls, and I randomly selected one of the urns and draw a ball from it, then the Jensen-Shannon divergence is the mutual information between which urn I drew the ball from, and the color of the ball drawn.

API

jensen_shannon_divergence(**args*, ***kwargs*)

The Jensen-Shannon Divergence: H(sum(w_i*P_i)) - sum(w_i*H(P_i)).

The square root of the Jensen-Shannon divergence is a distance metric.

Parameters

- **dists** (*[Distribution]*) – The distributions, P_i, to take the Jensen-Shannon Divergence of.
- **weights** (*[float], None*) – The weights, w_i, to give the distributions. If None, the weights are assumed to be uniform.

Returns **jsd** – The Jensen-Shannon Divergence

Return type float

Raises

- `ditException` – Raised if there *dists* and *weights* have unequal lengths.
- `InvalidNormalization` – Raised if the weights do not sum to unity.
- `InvalidProbability` – Raised if the weights are not valid probabilities.

Earth Mover's Distance

The Earth mover's distance is a distance measure between probability distributions. If we consider each probability mass function as a histogram of dirt, it is equal to the amount of work needed to optimally move the dirt of one histogram into the shape of the other.

For categorical data, the "distance" between unequal symbols is unitary. In this case, $1/6$ of the probability in symbol '0' needs to be moved to '1', and $1/6$ needs to be moved to '2', for a total of $1/3$:

```
In [1]: from dit.divergences import earth_movers_distance

In [2]: d1 = dit.ScalarDistribution([0, 1, 2], [2/3, 1/6, 1/6])

In [3]: d2 = dit.ScalarDistribution([0, 1, 2], [1/3, 1/3, 1/3])

In [4]: earth_movers_distance(d1, d2)
Out[4]: 0.5
```

API

earth_movers_distance(*dist1*, *dist2*, *distances=None*)

Compute the Earth Mover's Distance (EMD) between *dist1* and *dist2*. The EMD is the least amount of "probability mass flow" that must occur to transform *dist1* to *dist2*.

Parameters

- **dist1** (*Distribution*) – The first distribution.
- **dist2** (*Distribution*) – The second distribution.
- **distances** (*np.ndarray, None*) – A matrix of distances between outcomes of the distributions. If None, a distance matrix is constructed; if the distributions are categorical each non-equal event is considered at unit distance, and if numerical abs(x, y) is used as the distance.

Returns **emd** – The Earth Mover's Distance.

Return type float

While the cross entropy and the Kullback-Leibler divergence are not true metrics (they are not symmetric), the square root of the Jensen-Shannon divergence is.

Several measures of shared information are related to the ability of two (or more) agents to agree upon a secret key in the face of an eavesdropper:

1.7.5 Secret Key Agreement

One of the only methods of encrypting a message from Alice to Bomb such that no third party (Eve) can possibly decrypt it is a one-time pad. This technique requires that Alice and Bob have a secret sequence of bits, S, which Alice then encrypts by computing the exclusive-or of it with the plaintext, P, to produce the cyphertext, C: $C = S \oplus P$. Bob can then decrypt by xoring again: $P = S \oplus C$.

In order to pull this off, Alice and Bob need to construct S out of some sort joint randomness, $p(x, y, z)$, and public communication, V, which is assumed to have perfect fidelity. The maximum rate at which S can be constructed in the *secret key agreement rate*.

Background

Given N IID copies of a joint distribution governed by $p(x, y, z)$, let X^N denote the random variables observed by Alice, Y^N denote the random variables observed by Bob, and Z^N denote the random variables observed by Even. Furthermore, let $S[X : Y||Z]$ be the maximum rate R such that, for $N > 0$, $\epsilon > 0$, some public communication V, and functions f and g:

$$\begin{aligned} S_X &= f(X^N, V) \\ S_Y &= g(Y^N, V) \\ p(S_X \neq S_Y \neq S) &\leq \epsilon \\ IS : VZ^N &\leq \epsilon \\ \frac{1}{N} HS &\geq R - \epsilon \end{aligned}$$

Intuitively, this means there exists some procedure such that, for every N observations, Alice and Bob can publicly converse and then construct S bits which agree almost surely, and are almost surely independent of everything Eve has access to. S is then known as a *secret key*.

Lower Bounds

Lower Intrinsic Mutual Information

The first lower bound on the secret key agreement rate is known in `dit` as the `lower_intrinsic_mutual_information()`, and is given by:

$$IX : Y \uparrow Z = \max\{IX : Y - IX : Z, IX : Y - IY : Z, 0\}$$

Secrecy Capacity

Next is the secrecy capacity:

$$IX : Y \uparrow\uparrow Z = \max \begin{cases} \max\limits_{U-X-YZ} IU : Y - IU : Z \\ \max\limits_{U-Y-XZ} IU : X - IU : Z \end{cases}$$

This gives the secret key agreement rate when communication is not allowed.

Necessary Intrinsic Mutual Information

A tighter bound is given by the `necessary_intrinsic_mutual_information()` *[GGunluK17]*:

$$IX : Y \uparrow\uparrow\uparrow Z = \max \begin{cases} \max\limits_{V-U-X-YZ} IU : Y|V - IU : Z|V \\ \max\limits_{V-U-Y-XZ} IU : X|V - IU : Z|V \end{cases}$$

This quantity is actually equal to the secret key agreement rate when communication is limited to being unidirectional.

Upper Bounds

Upper Intrinsic Mutual Information

The secret key agreement rate is trivially upper bounded by:

$$\min\{IX : Y, IX : Y|Z\}$$

Intrinsic Mutual Information

The `intrinsic_mutual_information()` *[MW97]* is defined as:

$$IX : Y \downarrow Z = \min_{p(\overline{z}|z)} IX : Y|\overline{Z}$$

It is straightforward to see that $p(\overline{z}|z)$ being a constant achieves $IX : Y$, and $p(\overline{z}|z)$ being the identity achieves $IX : Y|Z$.

Reduced Intrinsic Mutual Information

This bound can be improved, producing the `reduced_intrinsic_mutual_information()` *[RSW03]*:

$$IX : Y \downarrow\downarrow Z = \min_U IX : Y \downarrow ZU + HU$$

This bound improves upon the Intrinsic Mutual Information when a small amount of information, U, can result in a larger decrease in the amount of information shared between X and Y given Z and U.

Minimal Intrinsic Mutual Information

The Reduced Intrinsic Mutual Information can be further reduced into the `minimal_intrinsic_total_correlation()` *[GA17]*:

$$IX : Y \downarrow\downarrow\downarrow Z = \min_U IX : Y|U + IXY : U|Z$$

All Together Now

Taken together, we see the following structure:

$$\begin{aligned}
&\min\{IX : Y, IX : Y|Z\} && (1.1)\\
&\geq IX : Y \downarrow Z && (1.2)\\
&\geq IX : Y \downarrow\downarrow Z && (1.3)\\
&\geq IX : Y \downarrow\downarrow\downarrow Z && (1.4)\\
&\geq S[X : Y||Z] && (1.5)\\
&\geq IX : Y \uparrow\uparrow\uparrow Z && (1.6)\\
&\geq IX : Y \uparrow\uparrow Z && (1.7)\\
&\geq IX : Y \uparrow Z && (1.8)\\
&\geq 0 && (1.9)
\end{aligned}$$

Generalizations

Most of the above bounds have straightforward multivariate generalizations. These are not necessarily bounds on the multiparty secret key agreement rate. For example, one could compute the `minimal_intrinsic_dual_total_correlation()`:

$$BX_0 : \ldots : X_n \downarrow\downarrow\downarrow Z = \min_U BX_0 : \ldots : X_n|U + IX_0, \ldots, X_n : U|Z$$

Examples

Let us consider a few examples:

```
In [1]: from dit.multivariate.secret_key_agreement import *

In [2]: from dit.example_dists.intrinsic import intrinsic_1, intrinsic_2, intrinsic_3
```

First, we consider the distribution `intrinsic_1`:

```
In [3]: print(intrinsic_1)
Class:          Distribution
Alphabet:       ('0', '1', '2', '3') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 3
RV Names:       None

x     p(x)
000   1/8
011   1/8
101   1/8
110   1/8
222   1/4
333   1/4
```

With upper bounds:

```
In [4]: upper_intrinsic_mutual_information(intrinsic_1, [[0], [1]], [2])
Out[4]: 0.5
```

We see that the trivial upper bound is 0.5, because without conditioning on Z, X and Y can agree when the observe either a 2 or a 3, which results in $IX : Y = 0.5$. Given Z, however, that information is no longer private. But, given Z, a conditional dependence is induced between X and Y: Z knows that if she is a 0 that X and Y agree, and if she is a 1 they disagree. This results $IX : Y|Z = 0.5$. In either case, however, X and Y can not agree upon a secret key: in the first case the eavesdropper knows their correlation, while in the second they are actually independent.

The `intrinsic_mutual_information()`, however can detect this:

```
In [5]: intrinsic_mutual_information(intrinsic_1, [[0], [1]], [2])
Out[5]: -4.440892098500626e-16
```

Next, let's consider the distribution `intrinsic_2`:

```
In [6]: print(intrinsic_2)
Class:          Distribution
Alphabet:       (('0', '1', '2', '3'), ('0', '1', '2', '3'), ('0', '1'))
Base:           linear
Outcome Class:  str
Outcome Length: 3
RV Names:       None

x     p(x)
000   1/8
011   1/8
101   1/8
110   1/8
220   1/4
331   1/4
```

In this case, Z no longer can distinguish between the case where X and Y can agree on a secret bit, and when they can not, because she can not determine when they are in the 01 regime or in the 23 regime:

```
In [7]: intrinsic_mutual_information(intrinsic_2, [[0], [1]], [2])
Out[7]: 1.5
```

This seems to imply that X and Y can adopt a scheme such as: if they observe either a 0 or a 1, write down 0, and if they observe either a 2 or a 3, write that down. This has a weakness, however: what if Z were able to distinguish the two regimes? This costs her 1 bit, but reduces the secrecy of X and Y to nil. Thus, the secret key agreement rate is actually only 1 bit:

```
In [8]: minimal_intrinsic_mutual_information(intrinsic_2, [[0], [1]], [2], bounds=(3,
↪))
Out[8]: 1.0
```

1.8 Information Profiles

There are several ways to decompose the information contained in a joint distribution. Here, we will demonstrate their behavior using four examples drawn from *[ASBY14]*:

```
In [1]: from dit.profiles import *

In [2]: ex1 = dit.Distribution(['000', '001', '010', '011', '100', '101', '110', '111
↪'], [1/8]*8)

In [3]: ex2 = dit.Distribution(['000', '111'], [1/2]*2)

In [4]: ex3 = dit.Distribution(['000', '001', '110', '111'], [1/4]*4)

In [5]: ex4 = dit.Distribution(['000', '011', '101', '110'], [1/4]*4)
```

1.8.1 Shannon Partition and Extropy Partition

The I-diagrams, or `ShannonPartition`, for these four examples can be computed thusly:

```
In [6]: ShannonPartition(ex1)
Out[6]:
+----------+--------+
| measure  |  bits  |
+----------+--------+
| H[0|1,2] |  1.000 |
| H[1|0,2] |  1.000 |
| H[2|0,1] |  1.000 |
| I[0:1|2] |  0.000 |
| I[0:2|1] |  0.000 |
| I[1:2|0] |  0.000 |
| I[0:1:2] |  0.000 |
+----------+--------+

In [7]: ShannonPartition(ex2)
Out[7]:
+----------+--------+
| measure  |  bits  |
+----------+--------+
| H[0|1,2] |  0.000 |
| H[1|0,2] |  0.000 |
| H[2|0,1] |  0.000 |
| I[0:1|2] |  0.000 |
| I[0:2|1] |  0.000 |
| I[1:2|0] |  0.000 |
| I[0:1:2] |  1.000 |
+----------+--------+
In [8]: ShannonPartition(ex3)
Out[8]:
+----------+--------+
| measure  |  bits  |
+----------+--------+
| H[0|1,2] |  0.000 |
| H[1|0,2] |  0.000 |
| H[2|0,1] |  1.000 |
| I[0:1|2] |  1.000 |
| I[0:2|1] |  0.000 |
| I[1:2|0] |  0.000 |
| I[0:1:2] |  0.000 |
+----------+--------+

In [9]: ShannonPartition(ex4)
Out[9]:
+----------+--------+
| measure  |  bits  |
+----------+--------+
| H[0|1,2] |  0.000 |
| H[1|0,2] |  0.000 |
| H[2|0,1] |  0.000 |
| I[0:1|2] |  1.000 |
| I[0:2|1] |  1.000 |
| I[1:2|0] |  1.000 |
| I[0:1:2] | -1.000 |
+----------+--------+
```

And their X-diagrams, or `ExtropyDiagram`, can be computed like so:

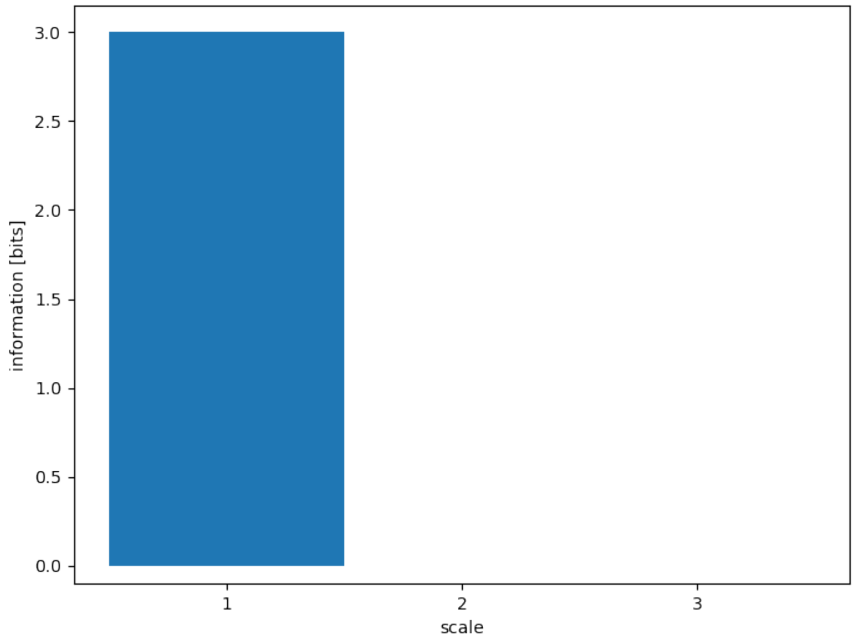
```
In [10]: ExtropyPartition(ex1)
Out[10]:
+----------+--------+
| measure  | exits  |
+----------+--------+
| X[0|1,2] |  0.103 |
| X[1|0,2] |  0.103 |
| X[2|0,1] |  0.103 |
| X[0:1|2] |  0.142 |
| X[0:2|1] |  0.142 |
| X[1:2|0] |  0.142 |
| X[0:1:2] |  0.613 |
+----------+--------+

In [11]: ExtropyPartition(ex2)
Out[11]:
+----------+--------+
| measure  | exits  |
+----------+--------+
| X[0|1,2] |  0.000 |
| X[1|0,2] |  0.000 |
| X[2|0,1] |  0.000 |
| X[0:1|2] |  0.000 |
| X[0:2|1] |  0.000 |
| X[1:2|0] |  0.000 |
| X[0:1:2] |  1.000 |
+----------+--------+
In [12]: ExtropyPartition(ex3)
Out[12]:
+----------+--------+
| measure  | exits  |
+----------+--------+
| X[0|1,2] |  0.000 |
| X[1|0,2] |  0.000 |
| X[2|0,1] |  0.245 |
| X[0:1|2] |  0.245 |
| X[0:2|1] |  0.000 |
| X[1:2|0] |  0.000 |
| X[0:1:2] |  0.755 |
+----------+--------+

In [13]: ExtropyPartition(ex4)
Out[13]:
+----------+--------+
| measure  | exits  |
+----------+--------+
| X[0|1,2] |  0.000 |
| X[1|0,2] |  0.000 |
| X[2|0,1] |  0.000 |
| X[0:1|2] |  0.245 |
| X[0:2|1] |  0.245 |
| X[1:2|0] |  0.245 |
| X[0:1:2] |  0.510 |
+----------+--------+
```

1.8.2 Complexity Profile

The complexity profile, implimented by `ComplexityProfile` is simply the amount of information at scale $\geq k$ of each "layer" of the I-diagram *[BY04]*.

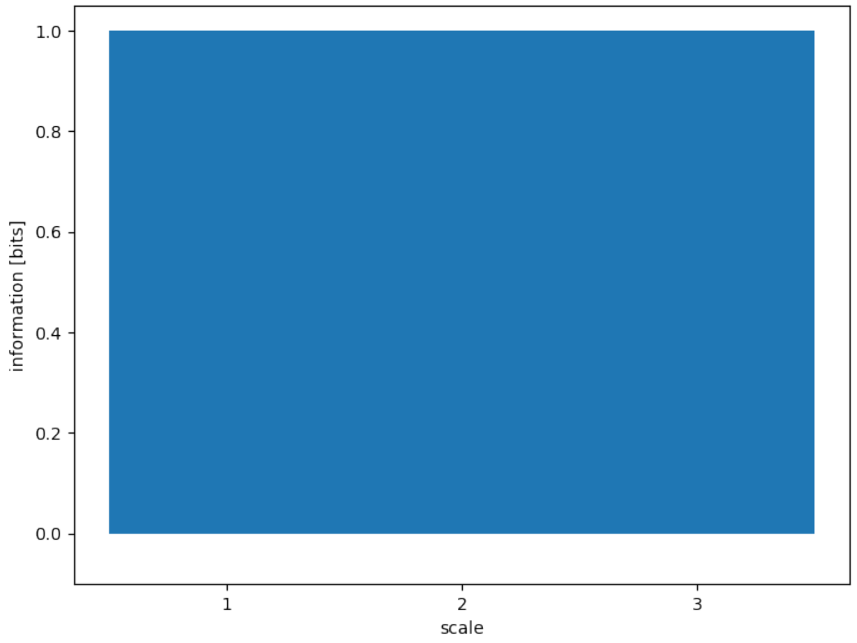
Consider example 1, which contains three independent bits. Each of these bits are in the outermost "layer" of the i-diagram, and so the information in the complexity profile is all at layer 1:

```
In [14]: ComplexityProfile(ex1).draw();
```



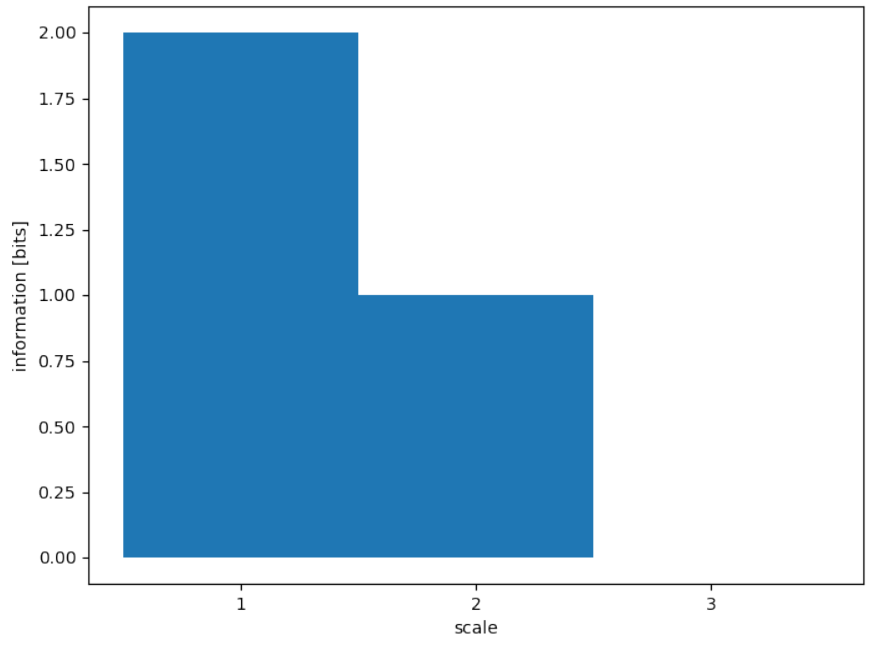
Whereas in example 2, all the information is in the center, and so each scale of the complexity profile picks up that one bit:

```
In [15]: ComplexityProfile(ex2).draw();
```



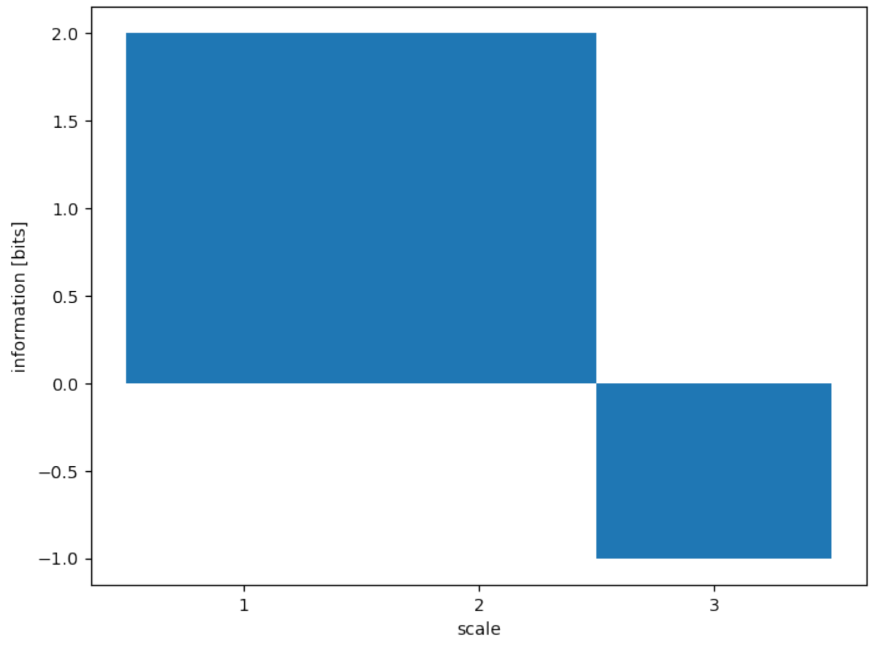
Both bits in example 3 are at a scale of at least 1, but only the shared bit persists to scale 2:

```
In [16]: ComplexityProfile(ex3).draw();
```



Finally, example 4 (where each variable is the `exclusive or` of the other two):

```
In [17]: ComplexityProfile(ex4).draw();
```

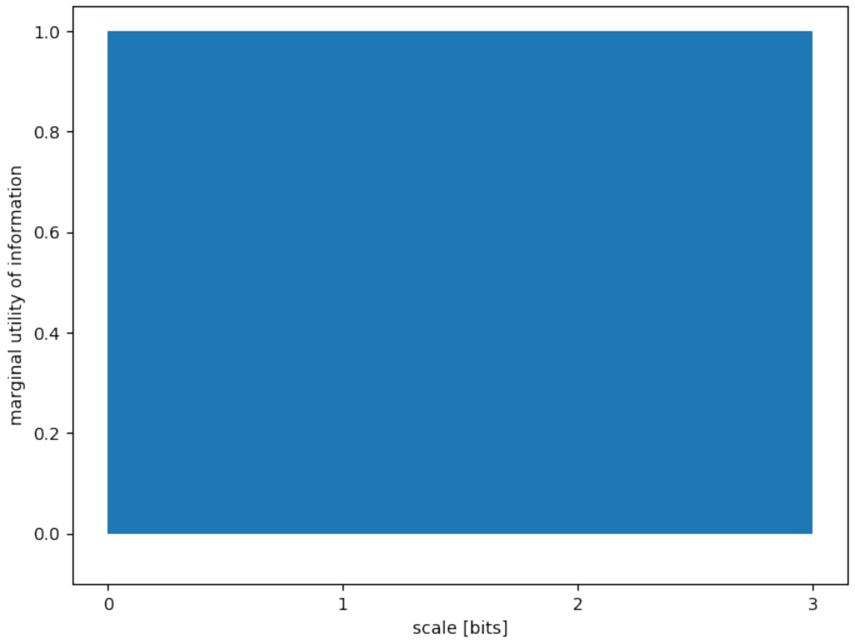


1.8.3 Marginal Utility of Information

The marginal utility of information (MUI) [ASBY14], implimented by `MUIProfile` takes a different approach. It asks, given an amount of information $Id : \{X\} = y$, what is the maximum amount of information one can extract using an auxilliary variable d as measured by the sum of the pairwise mutual informations, $\sum Id : X_i$. The MUI is then the rate of this maximum as a function of y.

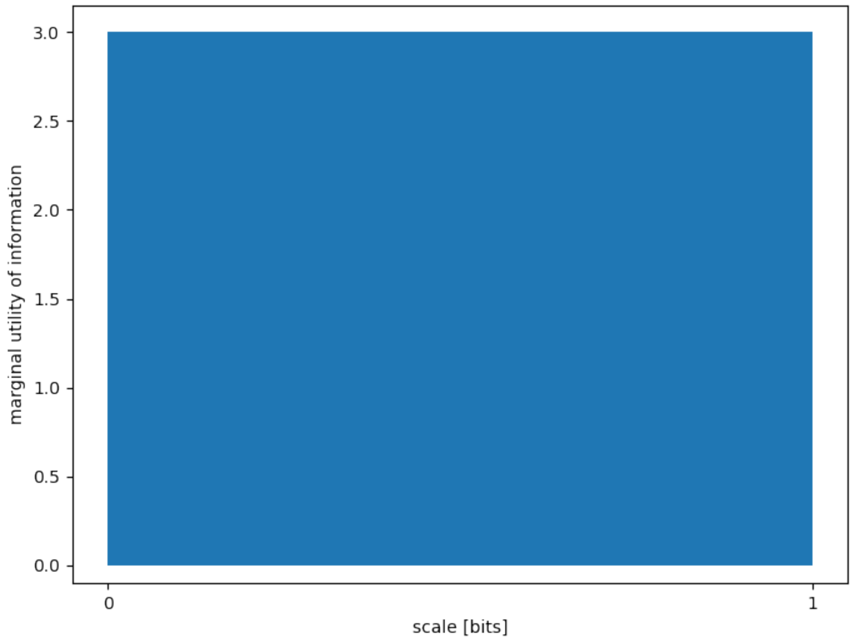
For the first example, each bit is independent and so basically must be extracted independently. Thus, as one increases y the maximum amount extracted grows equally:

```
In [18]: MUIProfile(ex1).draw();
```



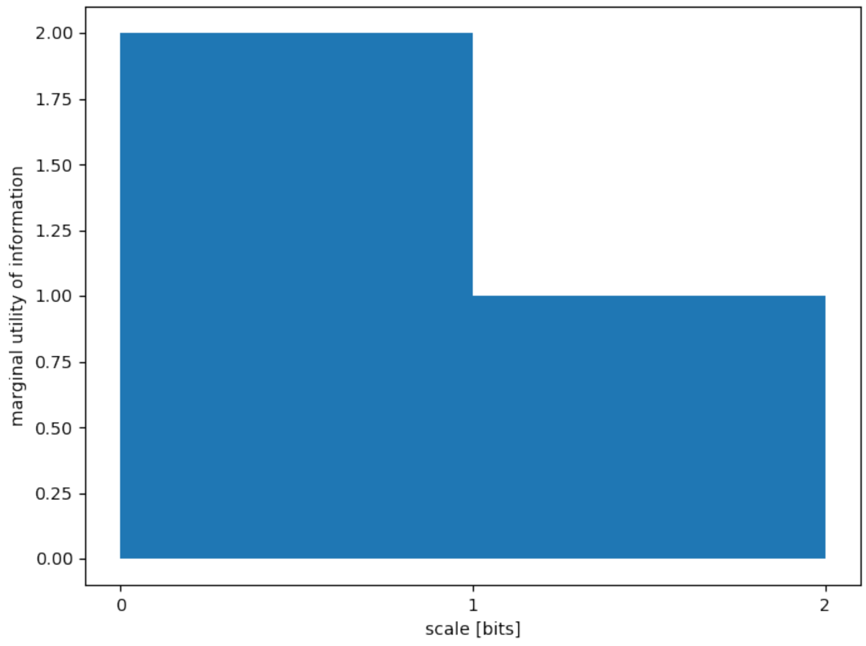
In the second example, there is only one bit total to be extracted, but it is shared by each pairwise mutual information. Therefore, for each increase in y we get a threefold increase in the amount extracted:

```
In [19]: MUIProfile(ex2).draw();
```



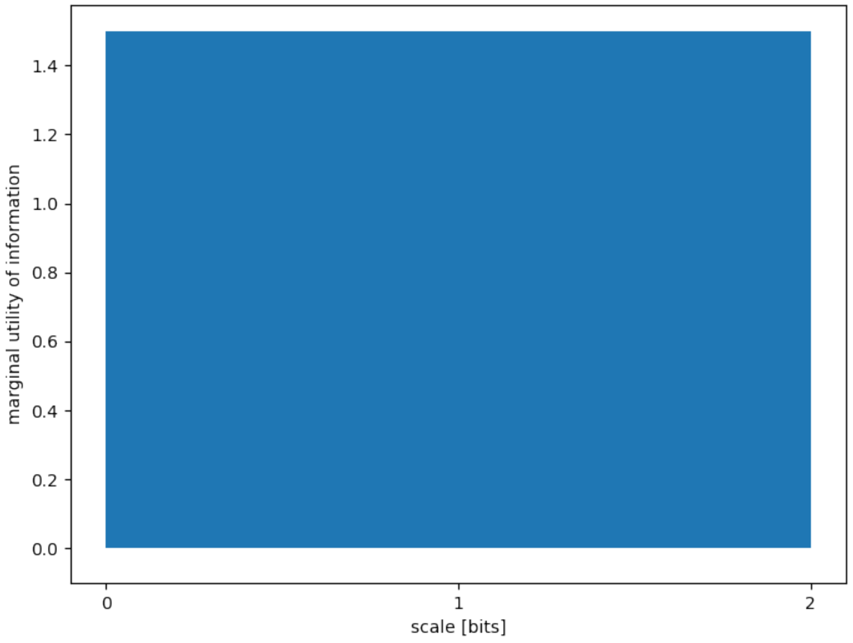
For the third example, for the first one bit of y we can pull from the shared bit, but after that one must pull from the independent bit, so we see a step in the MUI profile:

```
In [20]: MUIProfile(ex3).draw();
```



Lastly, the `xor` example:

```
In [21]: MUIProfile(ex4).draw();
```

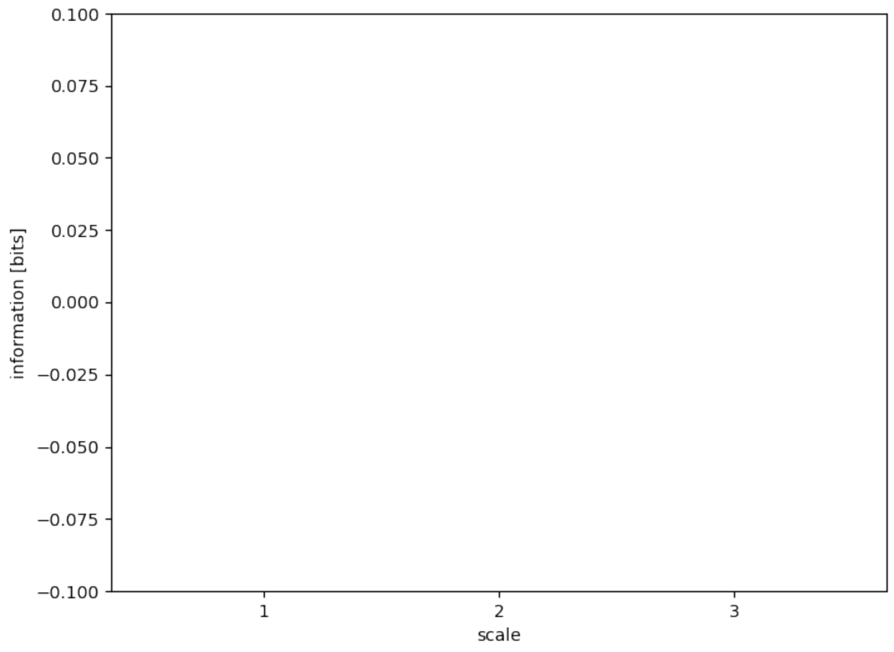


1.8.4 Schneidman Profile

Also known as the *connected information* or *network informations*, the Schneidman profile (`SchneidmanProfile`) exposes how much information is learned about the distribution when considering k-way dependencies *[Ama01][SSB+03]*. In all the following examples, each individual marginal is already uniformly distributed, and so the connected information at scale 1 is 0.

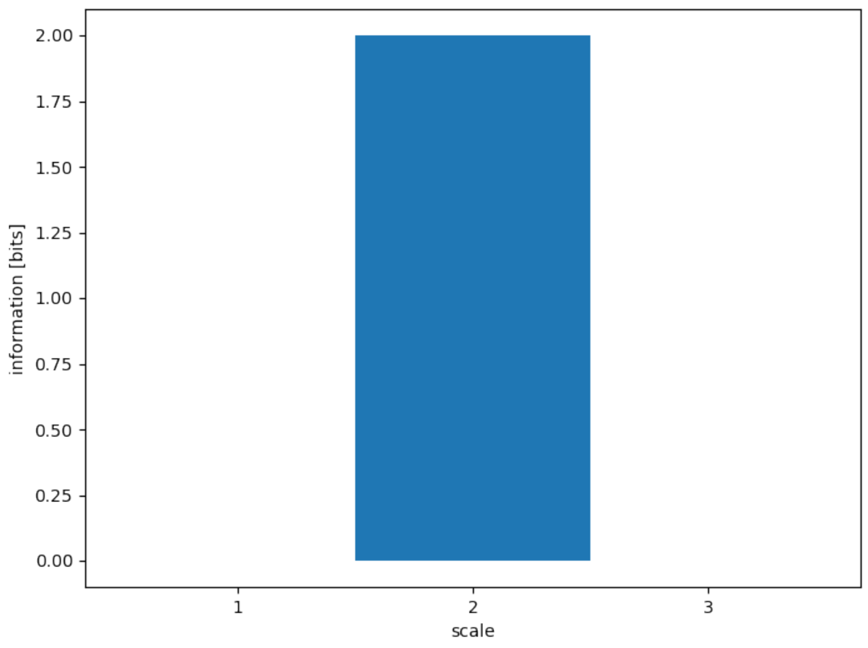
In the first example, all the random variables are independent already, so fixing marginals above $k = 1$ does not result in any change to the inferred distribution:

```
In [22]: SchneidmanProfile(ex1).draw();
```



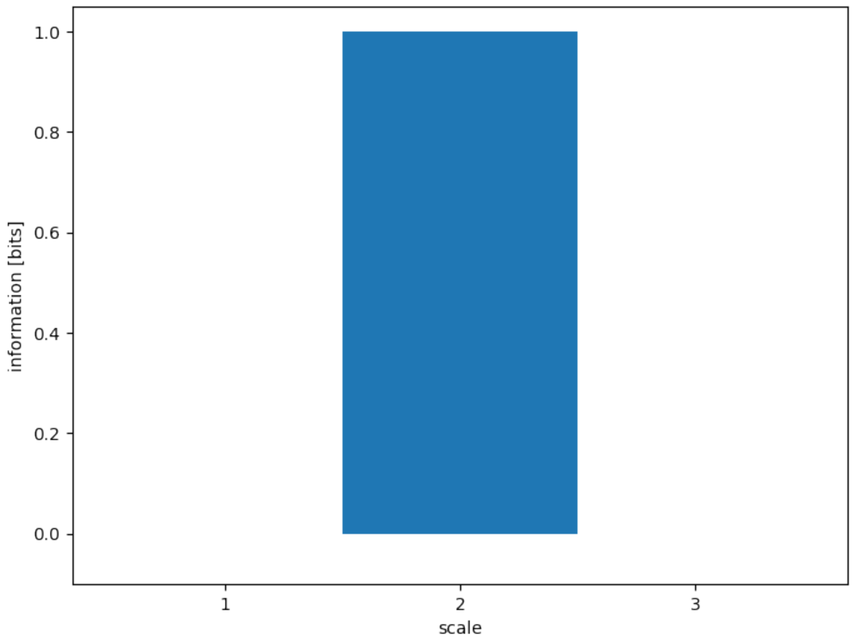
In the second example, by learning the pairwise marginals, we reduce the entropy of the distribution by two bits (from three independent bits, to one giant bit):

```
In [23]: SchneidmanProfile(ex2).draw();
```



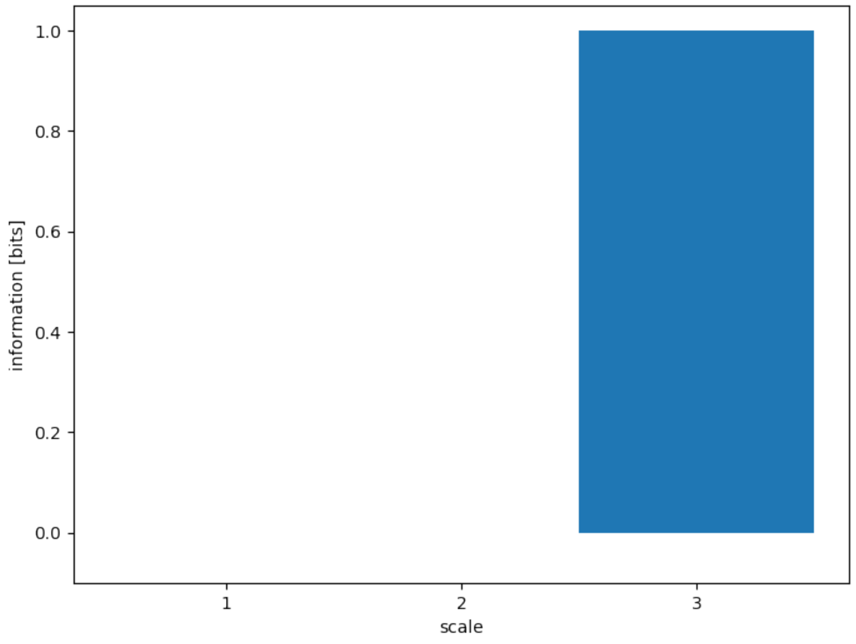
For the third example, learning pairwise marginals only reduces the entropy by one bit:

```
In [24]: SchneidmanProfile(ex3).draw();
```



And for the `xor`, all bits appear independent until fixing the three-way marginals at which point one bit about the distribution is learned:

```
In [25]: SchneidmanProfile(ex4).draw();
```



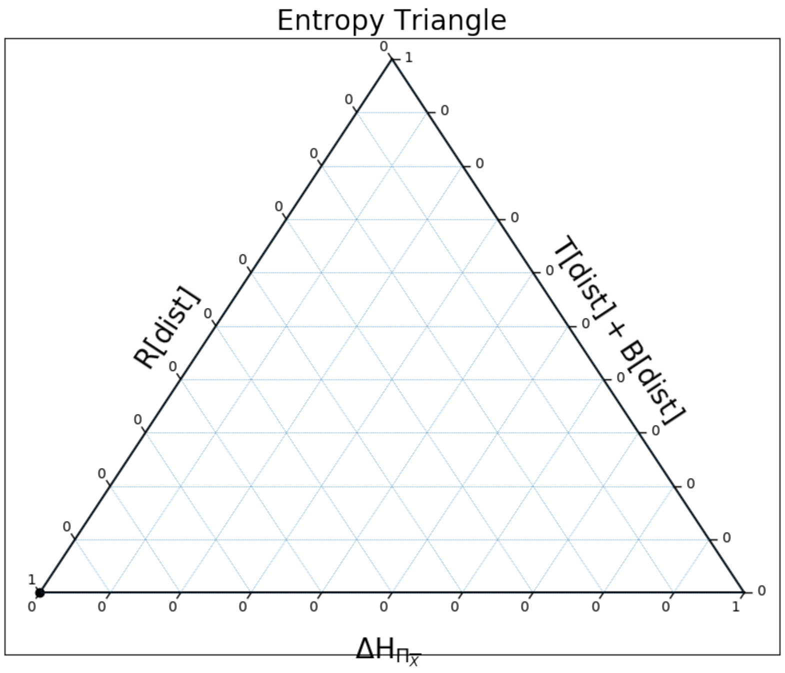
1.8.5 Entropy Triangle and Entropy Triangle2

The entropy triangle, `EntropyTriangle`, *[VAPelaezM16]* is a method of visualizing how the information in the distribution is distributed among deviation from uniformity, independence, and dependence. The deviation from independence is measured by considering the difference in entropy between a independent variables with uniform distributions, and independent variables with the same marginal distributions as the distribution in question. Independence is measured via the *Residual Entropy*, and dependence is measured by the sum of the *Total Correlation* and *Dual Total Correlation*.

All four examples lay along the left axis because their distributions are uniform over the events that have non-zero probability.

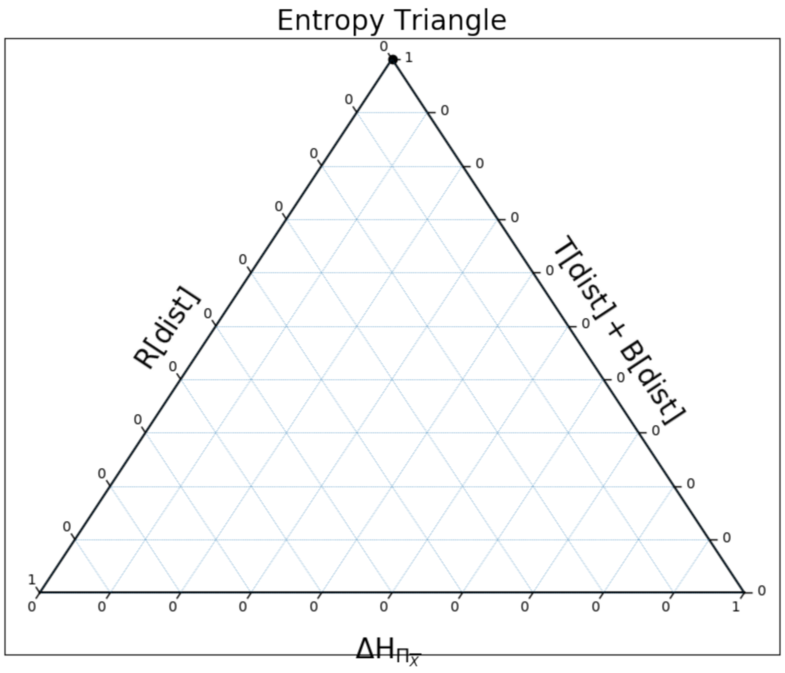
In the first example, the distribution is all independence because the three variables are, in fact, independent:

```
In [26]: EntropyTriangle(ex1).draw();
```



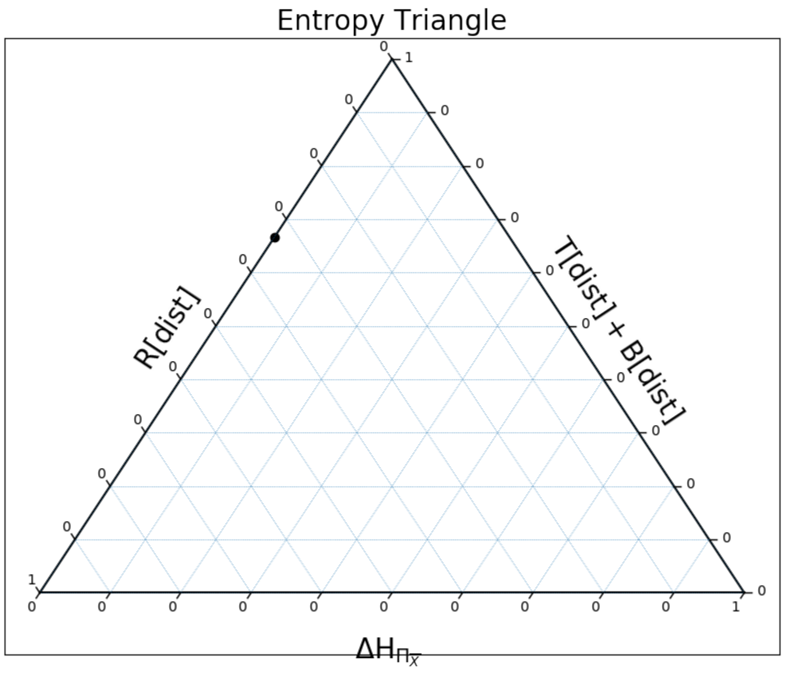
In the second example, the distribution is all dependence, because the three variables are perfectly entwined:

```
In [27]: EntropyTriangle(ex2).draw();
```



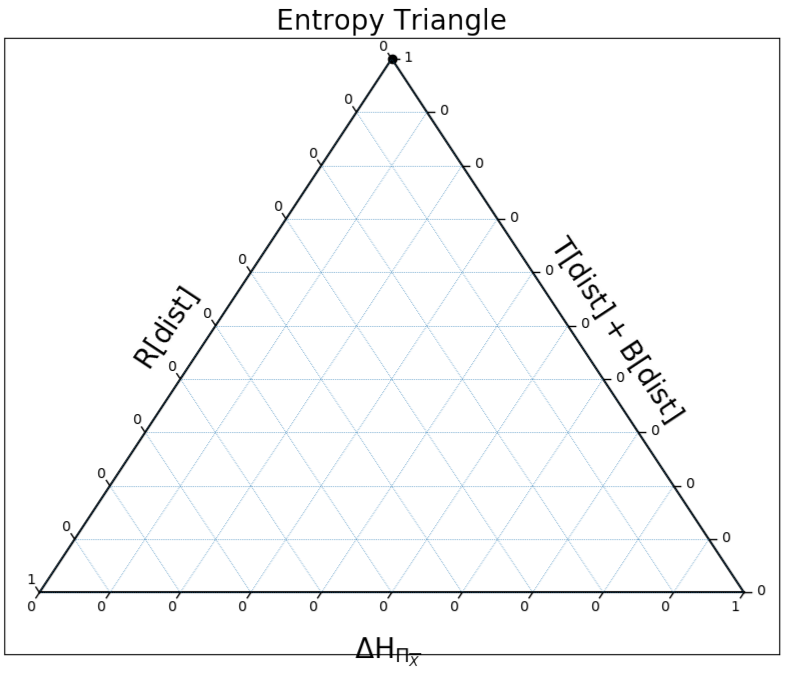
Here, there is a mix of independence and dependence:

```
In [28]: EntropyTriangle(ex3).draw();
```



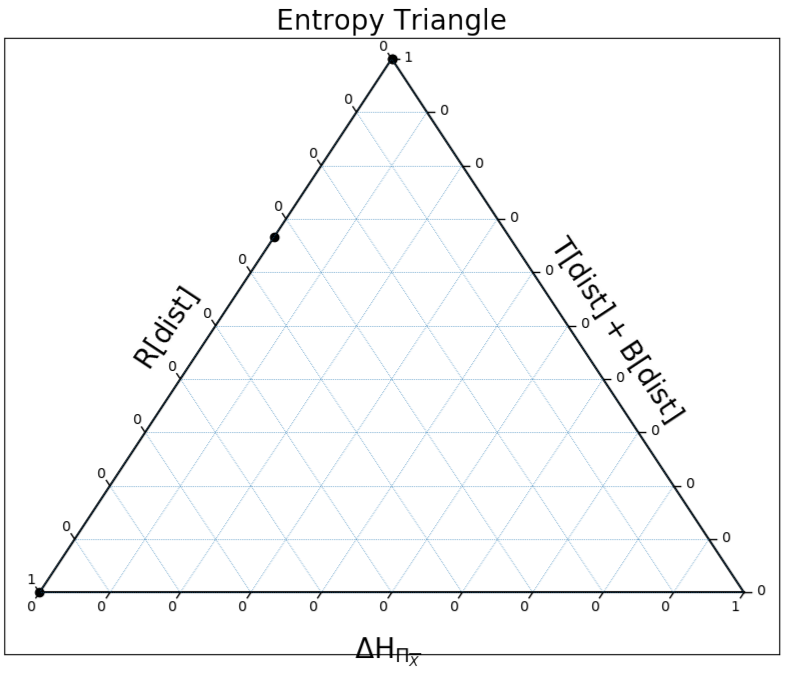
And finally, in the case of `xor`, the variables are completely dependent again:

```
In [29]: EntropyTriangle(ex4).draw();
```

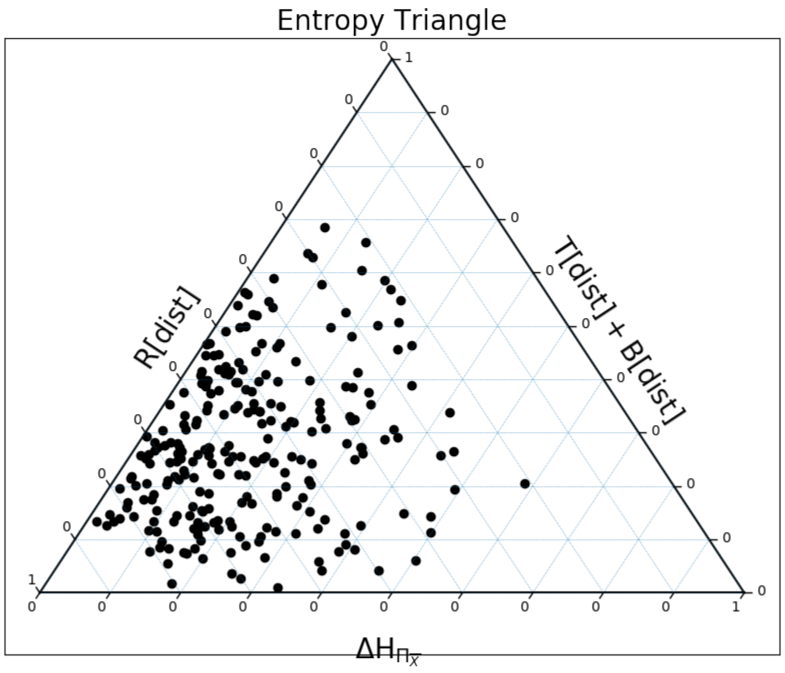


We can also plot all four on the same entropy triangle:

```
In [30]: EntropyTriangle([ex1, ex2, ex3, ex4]).draw();
```

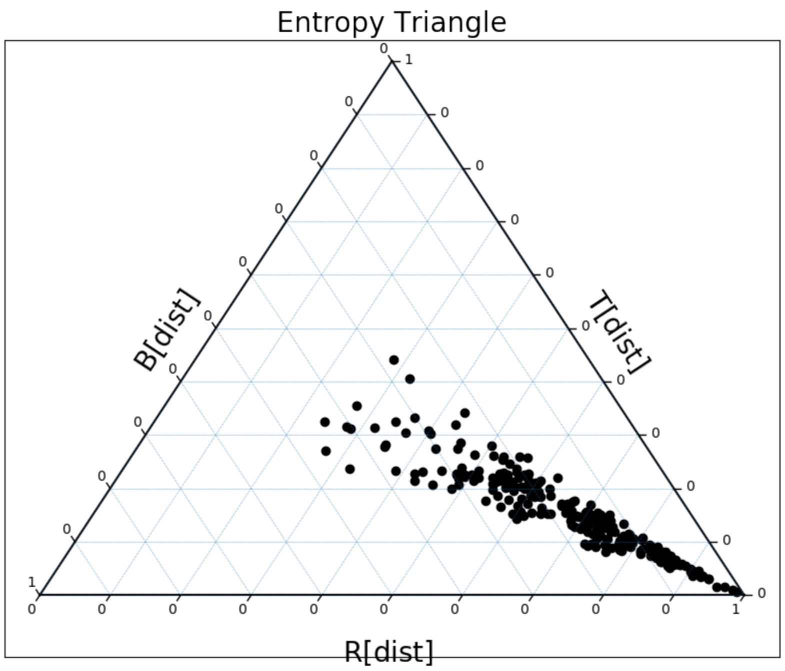


```
In [31]: dists = [ dit.random_distribution(3, 2, alpha=(0.5,)*8) for _ in range(250) ]

In [32]: EntropyTriangle(dists).draw();
```



We can plot these same distributions on a slightly different entropy triangle as well, `EntropyTriangle2`, one comparing the *Residual Entropy*, *Total Correlation*, and *Dual Total Correlation*:

```
In [33]: EntropyTriangle2(dists).draw();
```



1.8.6 Dependency Decomposition

Using `DependencyDecomposition`, one can discover how an arbitrary information measure varies as marginals of the distribution are fixed. In our first example, each variable is independent of the others, and so constraining marginals makes no difference:

```
In [34]: DependencyDecomposition(ex1)
Out[34]:
+------------+--------+
| dependency |   H    |
+------------+--------+
|    012     | 3.000 |
|  01:02:12  | 3.000 |
|   01:02    | 3.000 |
|   01:12    | 3.000 |
|   02:12    | 3.000 |
|    01:2    | 3.000 |
|    02:1    | 3.000 |
|    12:0    | 3.000 |
|   0:1:2    | 3.000 |
+------------+--------+
```

In the second example, we see that fixing any one of the pairwise marginals reduces the entropy by one bit, and by fixing a second we reduce the entropy down to one bit:

```
In [35]: DependencyDecomposition(ex2)
Out[35]:
+------------+--------+
| dependency |   H    |
+------------+--------+
|    012     | 1.000 |
|  01:02:12  | 1.000 |
|   01:02    | 1.000 |
|   01:12    | 1.000 |
|   02:12    | 1.000 |
|    01:2    | 2.000 |
|    02:1    | 2.000 |
|    12:0    | 2.000 |
|   0:1:2    | 3.000 |
+------------+--------+
```

In the third example, only constraining the 01 marginal reduces the entropy, and it reduces it by one bit:

```
In [36]: DependencyDecomposition(ex3)
Out[36]:
+------------+--------+
| dependency |   H    |
+------------+--------+
|    012     | 2.000 |
|  01:02:12  | 2.000 |
|   01:02    | 2.000 |
|   01:12    | 2.000 |
|   02:12    | 3.000 |
|    01:2    | 2.000 |
|    02:1    | 3.000 |
|    12:0    | 3.000 |
|   0:1:2    | 3.000 |
+------------+--------+
```

And finally in the case of the exclusive or, only constraining the 012 marginal reduces the entropy.

```
In [37]: DependencyDecomposition(ex4)
Out[37]:
+------------+--------+
| dependency |   H    |
+------------+--------+
|    012     | 2.000 |
|  01:02:12  | 3.000 |
|   01:02    | 3.000 |
|   01:12    | 3.000 |
|   02:12    | 3.000 |
|    01:2    | 3.000 |
|    02:1    | 3.000 |
|    12:0    | 3.000 |
|   0:1:2    | 3.000 |
+------------+--------+
```

1.9 Rate Distortion Theory

Note: We use p to denote fixed probability distributions, and q to denote probability distributions that are optimized.

Rate-distortion theory *[CT06]* is a framework for studying optimal lossy compression. Given a distribution $p(x)$, we wish to find $q(\hat{x}|x)$ which compresses X as much as possible while limiting the amount of user-defined distortion,

$d(x, \hat{x})$. The minimum rate (effectively, code book size) at which X can be compressed while maintaining a fixed distortion is known as the rate-distortion curve:

$$R(D) = \min_{q(\hat{x}|x), \langle d(x,\hat{x}) \rangle = D} IX : \hat{X}$$

By introducing a Lagrange multiplier, we can transform this constrained optimization into an unconstrained one:

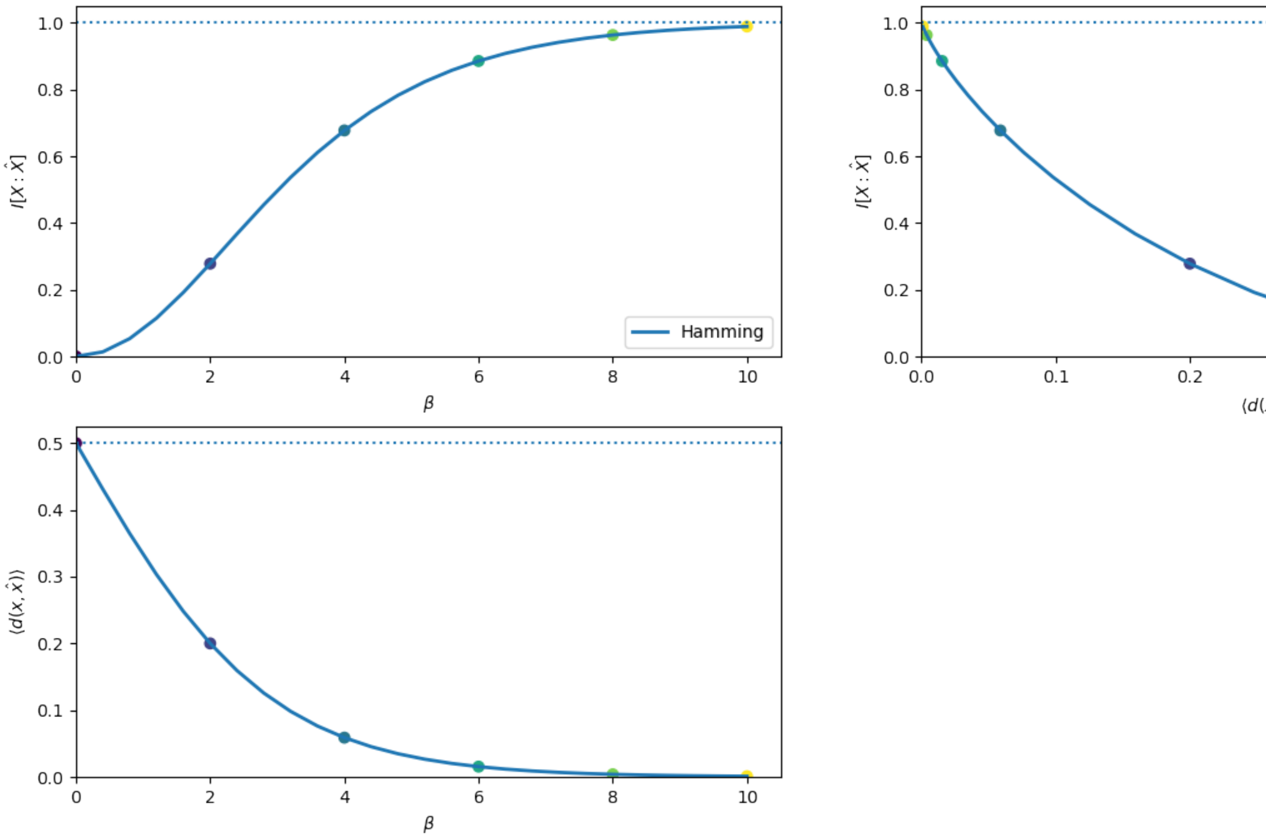
$$\mathcal{L} = IX : \hat{X} + \beta \langle d(x, \hat{x}) \rangle$$

where minimizing at each β produces a point on the curve.

1.9.1 Example

It is known that under the Hamming distortion ($d(x, \hat{x}) = [x \neq \hat{x}]$) the rate-distortion function for a biased coin has the following solution: $R(D) = Hp - HD$:

```
In [1]: from dit.rate_distortion import RDCurve

In [2]: d = dit.Distribution(['0', '1'], [1/2, 1/2])

In [3]: RDCurve(d, beta_num=26).plot();
```



1.10 Information Bottleneck

The information bottleneck *[TPB00]* is a form of rate-distortion where the distortion measure is given by:

$$d(x, \hat{x}) = D\,[\, p(Y|x) \;\;||\;\; q(Y|\hat{x})\,]$$

where D is an arbitrary divergence measure, and $\hat{X} - X - Y$ form a Markov chain. Traditionally, D is the *Kullback-Leibler Divergence*, in which case the average distortion takes a particular form:

$$\begin{aligned}
\langle d(x, \hat{x}) \rangle &= \sum_{x,\hat{x}} q(x, \hat{x}) D_{KL} p(Y|x) || q(Y|\hat{x}) \\
&= \sum_{x,\hat{x}} q(x, \hat{x}) \sum_{y} p(y|x) \log_2 \frac{p(y|x)}{q(y|\hat{x})} \\
&= \sum_{x,\hat{x},y} q(x, \hat{x}, y) \log_2 \frac{p(y|x)p(x)p(y)q(\hat{x})}{q(y|\hat{x})p(x)p(y)q(\hat{x})} \\
&= \sum_{x,\hat{x},y} q(x, \hat{x}, y) \log_2 \frac{p(y|x)p(x)}{p(x)p(y)} \frac{p(y)q(\hat{x})}{q(y|\hat{x})q(\hat{x})} \\
&= IX : Y - I\hat{X} : Y
\end{aligned}$$

Since $IX : Y$ is constant over $q(\hat{x}|x)$, it can be removed from the optimization. Furthermore,

$$\begin{aligned}
IX : Y - I\hat{X} : Y &= (IX : Y|\hat{X} + IX : Y : \hat{X}) - (IY : \hat{X}|X + IX : Y : \hat{X}) \\
&= IX : Y|\hat{X} - IY : \hat{X}|X \\
&= IX : Y|\hat{X}
\end{aligned}$$

where the final equality is due to the Markov chain. Due to all this, Information Bottleneck utilizes a “relevance” term, $I\hat{X} : Y$, which replaces the average distortion in the Lagrangian:

$$\mathcal{L} = IX : \hat{X} - \beta I\hat{X} : Y \;.$$

Though $IX : Y|\hat{X}$ is the most simplified form of the average distortion, it is faster to compute $I\hat{X} : Y$ during optimization.

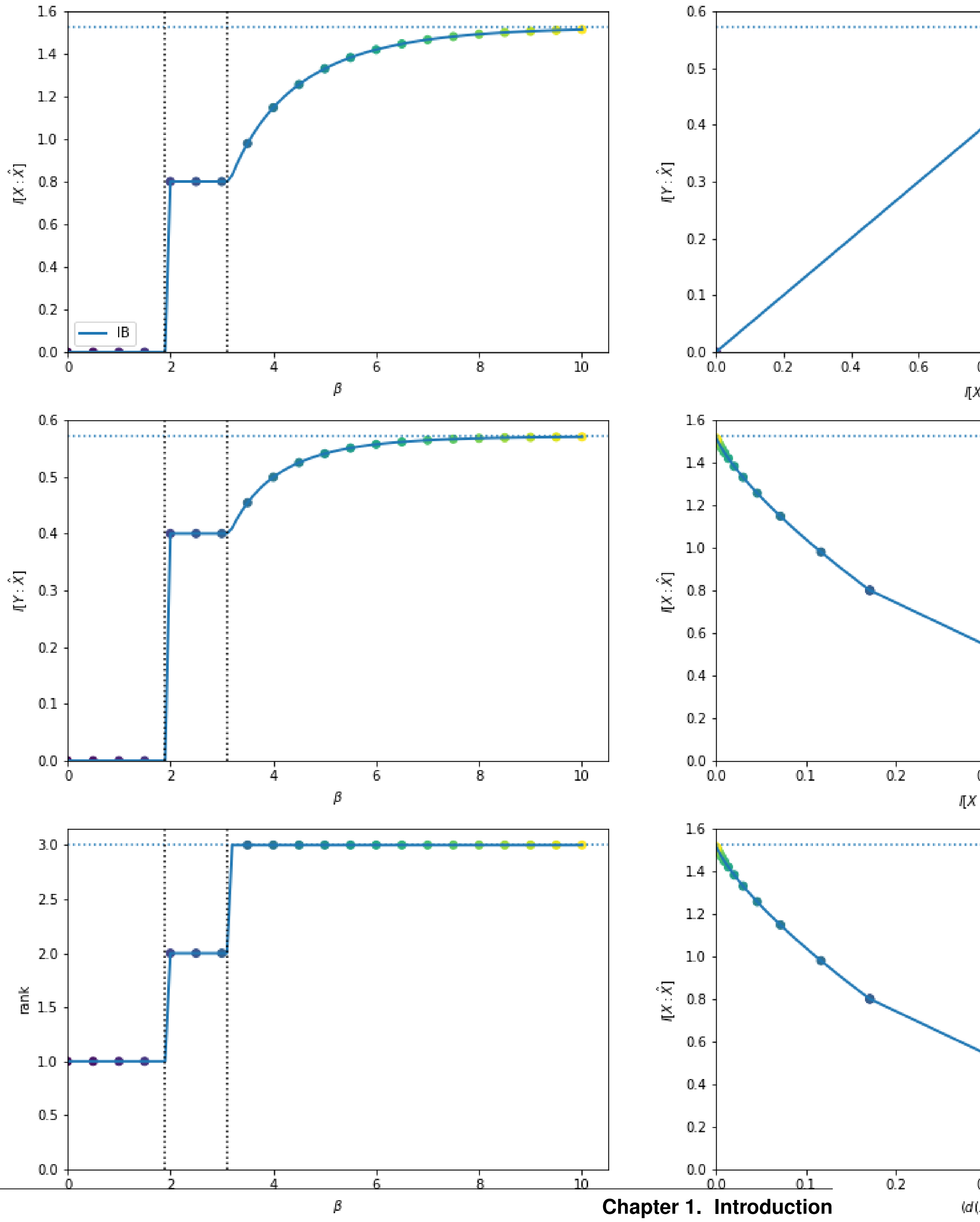
1.10.1 Example

Consider this distribution:

```
In [4]: d = dit.Distribution(['00', '02', '12', '21', '22'], [1/5]*5)
```

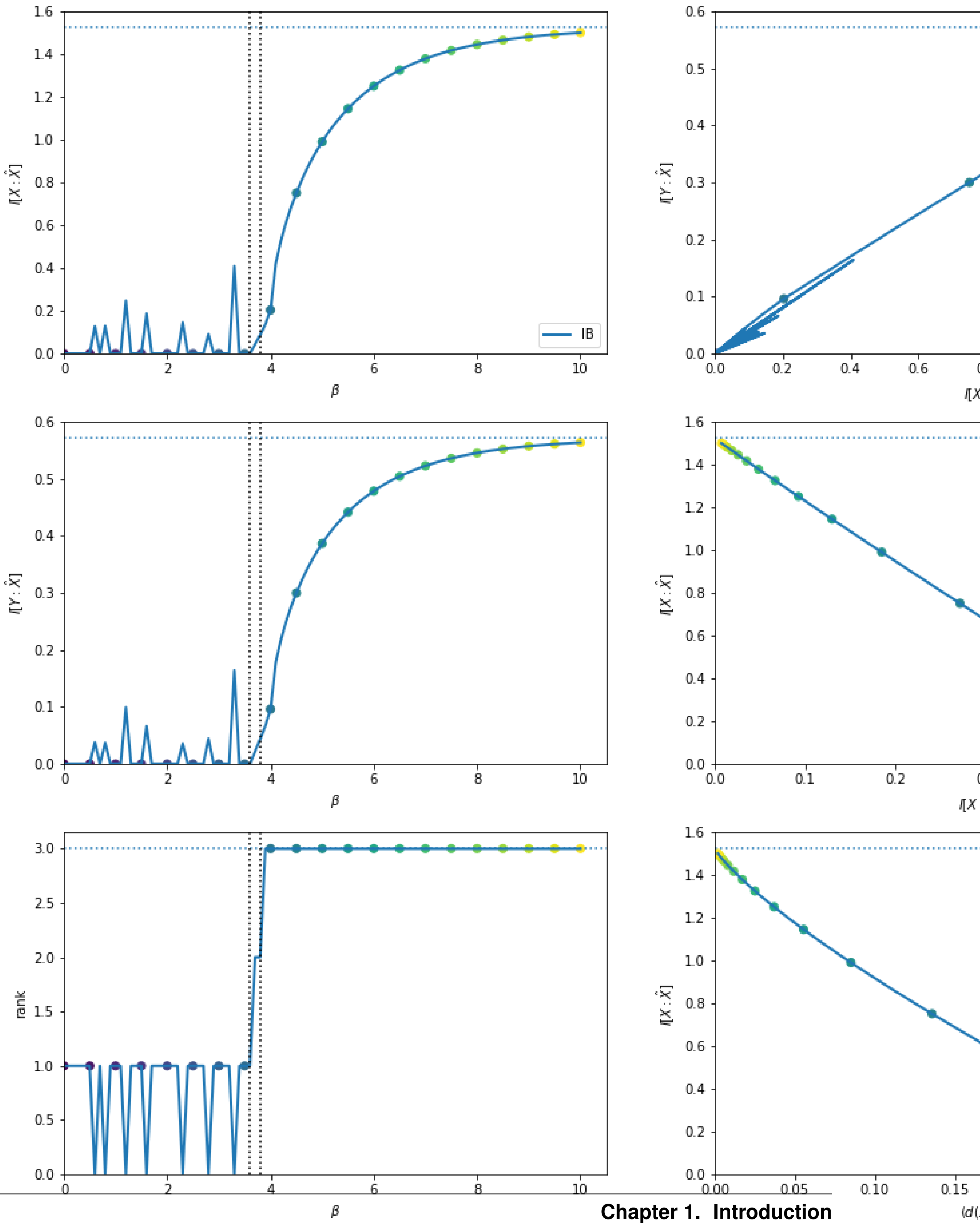
There are effectively three features that the fist index, X, has regarding the second index, Y. We can find them using the standard information bottleneck:

```
In [5]: from dit.rate_distortion import IBCurve

In [6]: IBCurve(d, beta_num=26).plot();
```


IB
β
$I[X:\hat{X}]$
$I[Y:\hat{X}]$
rank

We can also find them utilizing the total variation:

```
In [7]: from dit.divergences.pmf import variational_distance

In [8]: IBCurve(d, divergence=variational_distance).plot();
```


IB
β
$I[X:\hat{X}]$
$I[Y:\hat{X}]$
rank

Note: The spiky behavior at low β values is due to numerical imprecision.

1.11 APIs

class RDCurve(*dist*, *rv=None*, *crvs=None*, *beta_min=0*, *beta_max=10*, *beta_num=101*, *alpha=1.0*, *distortion=Distortion(name='Hamming'*, *matrix=<function hamming_distortion>*, *optimizer=<class 'dit.rate_distortion.rate_distortion.RateDistortionHamming'>)*, *method=None*)
Compute a rate-distortion curve.

class IBCurve(*dist*, *rvs=None*, *crvs=None*, *rv_mode=None*, *beta_min=0.0*, *beta_max=15.0*, *beta_num=101*, *alpha=1.0*, *method='sp'*, *divergence=None*)
Compute an information bottleneck curve.

1.12 Partial Information Decomposition

The *partial information decomposition* (PID), put forth by Williams & Beer *[WB10]*, is a framework for decomposing the information shared between a set of variables we will refer to as *inputs*, $X_0, X_1, \ldots$, and another random variable we will refer to as the *output*, Y. This decomposition seeks to partition the information $IX_0, X_1, \ldots : Y$ among the antichains of the inputs.

1.12.1 Background

It is often desirable to determine how a set of inputs influence the behavior of an output. Consider the exclusive or logic gates, for example:

```
In [1]: from dit.pid.distributions import bivariates, trivariates

In [2]: xor = bivariates['synergy']

In [3]: print(xor)
Class:          Distribution
Alphabet:       ('0', '1') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 3
RV Names:       None

x     p(x)
000   1/4
011   1/4
101   1/4
110   1/4
```

We can see from inspection that either input (the first two indexes) is independent of the output (the final index), yet the two inputs together determine the output. One could call this “synergistic” information. Next, consider the giant bit distribution:

```
In [4]: gb = bivariates['redundant']
In [5]: print(gb)
Class:          Distribution
Alphabet:       ('0', '1') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 3
RV Names:       None

x     p(x)
000   1/2
111   1/2
```

Here, we see that either input informs us of exactly what the output is. One could call this "redundant" information. Furthermore, consider the coinformation of these distributions:

```
In [6]: from dit.multivariate import coinformation as I

In [7]: I(xor)
Out[7]: -1.0

In [8]: I(gb)
Out[8]: 1.0
```

This could lead one to intuit that negative values of the coinformation correspond to synergistic effects in a distribution, while positive values correspond to redundant effects. This intuition, however, is at best misleading: the coinformation of a 4-variable giant bit and 4-variable parity distribution are both positive:

```
In [9]: I(dit.example_dists.giant_bit(4, 2))
Out[9]: 1.0

In [10]: I(dit.example_dists.n_mod_m(4, 2))
Out[10]: 1.0
```

This, as well as other issues, lead Williams & Beer *[WB10]* to propose the *partial information decomposition*.

1.12.2 Framework

The goal of the partial information is to assign to each some non-negative portion of $I\{X_i\} : Y$ to each antichain over the inputs. An antichain over the inputs is a set of sets, where each of those sets is not a subset of any of the others. For example, $\{\{X_0, X_1\}, \{X_1, X_2\}\}$ is an antichain, but $\{\{X_0, X_1\}, \{X_0 X_1, X_2\}\}$ is not.

The antichains for a lattice based on this partial order:

$$\alpha \leq \beta \iff \forall \mathbf{b} \in \beta, \exists \mathbf{a} \in \alpha, \mathbf{a} \subseteq \mathbf{b}$$

From here, we wish to find a redundancy measure, $I_\cap \bullet$ which would assign a fraction of $I\{X_i\} : Y$ to each antichain intuitively quantifying what portion of the information in the output could be learned by observing any of the sets of variables within the antichain. In order to be a viable measure of redundancy, there are several axioms a redundancy measure must satisfy.

Bivariate Lattice

Let us consider the special case of two inputs. The lattice consists of four elements: $\{\{X_0\}, \{X_1\}\}$, $\{\{X_0\}\}$, $\{\{X_1\}\}$, and $\{\{X_0, X_1\}\}$. We can interpret these elements as the *redundancy* provided by both inputs, the information *uniquely* provided by X_0, the information *uniquely* provided by X_1, and the information *synergistically* provided only by both inputs together. Together these for elements decompose the input-output mutual information:

$$IX_0, X_1 : Y = I_\cap\{X_0\}, \{X_1\} : Y + I_\cap\{X_0\} : Y + I_\cap\{X_1\} : Y + I_\cap\{X_0, X_1\} : Y$$

Furthermore, due to the self-redundancy axiom (described ahead), the single-input mutual informations decomposed in the following way:

$$IX_0 : Y = I_\cap\{X_0\}, \{X_1\} : Y + I_\cap\{X_0\} : Y$$
$$IX_1 : Y = I_\cap\{X_0\}, \{X_1\} : Y + I_\cap\{X_1\} : Y$$

Colloquially, from input X_0 one can learn what is redundantly provided by either input, plus what is uniquely provided by X_0, but not what is uniquely provided by X_1 or what can only be learned synergistically from both inputs.

Axioms

The following three axioms were provided by Williams & Beer.

Symmetry

The redundancy $I_\cap X_{0:n} : Y$ is invariant under reorderings of X_i.

Self-Redundancy

The redundancy of a single input is its mutual information with the output:

$$I_\cap X_i : Y = IX_i : Y$$

Monotonicity

The redundancy should only decrease with in inclusion of more inputs:

$$I_\cap \mathcal{A}_1, \ldots, \mathcal{A}_{k-1}, \mathcal{A}_k : Y \leq I_\cap \mathcal{A}_1, \ldots, \mathcal{A}_{k-1} : Y$$

with equality if $\mathcal{A}_{k-1} \subseteq \mathcal{A}_k$.

There have been other axioms proposed following from those of Williams & Beer.

Identity

The identity axiom *[HSP13]* states that if the output is identical to the inputs, then the redundancy is the mutual information between the inputs:

$$I_\cap X_0, X_1 : (X_0, X_1) = IX_0 : X_1$$

Target (output) Monotonicity

This axiom states that redundancy can not increase when replacing the output by a function of itself.

$$I_\cap X_{0:n} : Y \geq I_\cap X_{0:n} : f(Y)$$

It first appeared in *[BROJ13]* and was expanded upon in *[RBO+17]*.

1.12.3 Measures

We now turn our attention a variety of methods proposed to flesh out this partial information decomposition.

```
In [11]: from dit.pid import *
```

$I_{min}\bullet$

$I_{min}\bullet$*[WB10]* was Williams & Beer's initial proposal for a redundancy measure. It is given by:

$$I_{min}\mathcal{A}_1, \mathcal{A}_2, \ldots : Y = \sum_{y \in Y} p(y) \min_{\mathcal{A}_i} I\mathcal{A}_i : Y = y$$

However, this measure has been criticized for acting in an unintuitive manner *[GK14]*:

```
In [12]: d = dit.Distribution(['000', '011', '102', '113'], [1/4]*4)

In [13]: PID_WB(d)
Out[13]:
+--------+--------+--------+
| I_min  |  I_r   |   pi   |
+--------+--------+--------+
| {0:1}  | 2.0000 | 1.0000 |
|  {0}   | 1.0000 | 0.0000 |
|  {1}   | 1.0000 | 0.0000 |
| {0}{1} | 1.0000 | 1.0000 |
+--------+--------+--------+
```

We have constructed a distribution whose inputs are independent random bits, and whose output is the concatenation of those inputs. Intuitively, the output should then be informed by one bit of unique information from X_0 and one bit of unique information from X_1. However, $I_{min}\bullet$ assesses that there is one bit of redundant information, and one bit of synergistic information. This is because $I_{min}\bullet$ quantifies redundancy as the least amount of information one can learn about an output given any single input. Here, however, the one bit we learn from X_0 is, in a sense, orthogonal from the one bit we learn from X_1. This observation has lead to much of the follow-on work.

$I_{MMI}\bullet$

One potential measure of redundancy is the *minimum mutual information [BROJ13]*:

$$I_{MMI}X_{0:n} : Y = \min_i IX_i : Y$$

This measure, though crude, is known to be correct for multivariate gaussian variables *[OBR15]*.

$I_{\downarrow}\bullet$

Drawing inspiration from information-theoretic cryptography, this PID quantifies unique information using the Intrinsic Mutual Information:

$$I_{\downarrow}X_{0:n} : Y = IX_i : Y \downarrow X_{\overline{\{i\}}}$$

While this seems intuitively plausible, it turns out that this leads to an inconsistent decomposition *[BROJ13]*; namely, in the bivariate case, if one were to compute redundancy using either unique information subtracted from that inputs mutual information with the output the value should be the same. There are examples where this is not the case:

```
In [14]: d = bivariates['prob 2']

In [15]: PID_downarrow(d)
Out[15]:
+--------+--------+--------+
|  [31mI_da[0m  |  I_r   |   pi   |
+--------+--------+--------+
| {0:1}  | 1.0000 | 0.1887 |
|  {0}   | 0.3113 | 0.1887 |
|  {1}   | 0.5000 | 0.5000 |
| {0}{1} | 0.1226 | 0.1226 |
+--------+--------+--------+
```

Interestingly, compared to other measures the intrinsic mutual information seems to *overestimate* unique information. Since $I X_0 : Y \downarrow X_1 \leq \min\{I X_0 : Y | X_1, I X_0 : Y\} = \min\{U_0 + S, U_0 + R\}$, where R is redundancy, U_0 is unique information from input X_0, and S is synergy, this implies that the optimization performed in computing the intrinsic mutual information is unable to completely remove either redundancy, synergy, or both.

$I_\wedge \bullet$

Redundancy seems to intuitively be related to common information Common Informations. This intuition lead to the development of $I_\wedge \bullet$ *[GCJ+14]*:

$$I_\wedge X_{0:n} : Y = I \curlywedge X_i : Y$$

That is, redundancy is the information the Gács-Körner Common Information of the inputs shares with the output. This measure is known to produce negative partial information values in some instances.

$I_{proj} \bullet$

Utilizing information geometry, Harder et al *[HSP13]* have developed a strictly bivariate measure of redundancy, $I_{proj} \bullet$:

$$I_{proj}\{X_0\}\{X_1\} : Y = \min\{I_Y^\pi[X_0 \searrow X_1], I_Y^\pi[X_1 \searrow X_0]\}$$

where

$$\begin{aligned} I_Y^\pi[X_0 \searrow X_1] &= \sum_{x_0, y} p(x_0, y) \log \frac{p_{(x_0 \searrow X_1)}(y)}{p(y)} \\ p_{(x_0 \searrow X_1)}(Y) &= \pi_{C_{cl}(\langle X_1 \rangle_Y)}(p(Y|x_0) \\ \pi_B(p) &= \arg\min_{r \in B} D_{KL} p || r \\ C_{cl}(\langle X_1 \rangle_Y) &= C_{cl}(\{p(Y|x_1) : x_1 \in X_1\}) \end{aligned}$$

where $C_{cl}(\bullet)$ denotes closure. Intuitively, this measures seeks to quantify redundancy as the minimum of how much $p(Y|X_0)$ can be expressed when X_0 is projected on to X_1, and vice versa.

$I_{BROJA} \bullet$

In a very intuitive effort, Bertschinger et al (henceforth BROJA) *[BRO+14][GK14]* defined unique information as the minimum conditional mutual informations obtainable while holding the input-output marginals fixed:

$$\begin{aligned} \Delta &= \{Q : \forall i : p(x_i, y) = q(x_i, y)\} \\ I_{BROJA} X_{0:n} : Y &= \min_{Q \in \Delta} I X_i : Y | X_{\overline{\{i\}}} \end{aligned}$$

Note: In the bivariate case, Griffith independently suggested the same decomposition but from the viewpoint of synergy *[GK14]*.

The BROJA measure has recently been criticized for behaving in an unintuitive manner on some examples. Consider the *reduced or* distribution:

```
In [16]: bivariates['reduced or']
Out[16]:
Class:          Distribution
Alphabet:       ('0', '1') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 3
RV Names:       None

x     p(x)
000   1/2
011   1/4
101   1/4

In [17]: print(PID_BROJA(bivariates['reduced or']))
+---------+--------+--------+
| I_broja |  I_r   |   pi   |
+---------+--------+--------+
|  {0:1}  | 1.0000 | 0.6887 |
|   {0}   | 0.3113 | 0.0000 |
|   {1}   | 0.3113 | 0.0000 |
|  {0}{1} | 0.3113 | 0.3113 |
+---------+--------+--------+
```

We see that in this instance BROJA assigns no partial information to either unique information. However, it is not difficult to argue that in the case that either input is a 1, that input then has unique information regarding the output.

$I_{proj}\bullet$ and $I_{BROJA}\bullet$ are Distinct

In the BROJA paper *[BRO+14]* the only example given where their decomposition differs from that of Harder et al. is the `dit.example_dists.summed_dice()`. We can find a simpler example where they differ using hypothesis:

```
In [18]: from hypothesis import find

In [19]: from dit.utils.testing import distribution_structures

In [20]: find(distribution_structures(3, 2, True), lambda d: PID_Proj(d) != PID_
↪BROJA(d))
Out[20]:
Class:          Distribution
Alphabet:       (0, 1) for all rvs
Base:           linear
Outcome Class:  tuple
Outcome Length: 3
RV Names:       None

x           p(x)
(0, 0, 0)   1/4
(0, 0, 1)   1/4
```

```
(0, 1, 0)   1/4
(1, 0, 1)   1/4
```

I_{ccs}•

Taking a pointwise point of view, Ince has proposed a measure of redundancy based on the coinformation [Inc17a]:

$$I_{ccs}X_{0:n} : Y = \sum p(x_0, \ldots, x_n, y)Ix_0 : \ldots : x_n : y \text{ if } \operatorname{sign}(Ix_i : y) = \operatorname{sign}(Ix_0 : \ldots : x_n : y)$$

While this measure behaves intuitively in many examples, it also assigns negative values to some partial information atoms in some instances.

This decomposition also displays an interesting phenomena, that of *subadditive redundancy*. The **gband** distribution is an independent mix of a giant bit (redundancy of 1 bit) and the **and** distribution (redundancy of 0.1038 bits), and yet **gband** has 0.8113 bits of redundancy:

```
In [21]: PID_CCS(bivariates['gband'])
Out[21]:
+--------+--------+--------+
| I_ccs  |  I_r   |   pi   |
+--------+--------+--------+
| {0:1}  | 1.8113 | 0.0000 |
|  {0}   | 1.3113 | 0.5000 |
|  {1}   | 1.3113 | 0.5000 |
| {0}{1} | 0.8113 | 0.8113 |
+--------+--------+--------+
```

I_{dep}•

James et al [JEC17] have developed a method of quantifying unique information based on the Dependency Decomposition. Unique information from variable X_i is evaluated as the least change in sources-target mutual information when adding the constraint X_iY.

```
In [22]: PID_dep(bivariates['not two'])
Out[22]:
+--------+--------+--------+
| I_dep  |  I_r   |   pi   |
+--------+--------+--------+
| {0:1}  | 0.5710 | 0.5364 |
|  {0}   | 0.0200 | 0.0146 |
|  {1}   | 0.0200 | 0.0146 |
| {0}{1} | 0.0054 | 0.0054 |
+--------+--------+--------+
```

•

Also taking a pointwise view, Finn & Lizier's • [FL17] instead splits the pointwise mutual information into two components:

$$i(s, t) = h(s) - h(s|t)$$

They then define two partial information lattices, one quantified locally by $h(s)$ and the other by $h(s|t)$. By averaging these local lattices and then recombining them, we arrive at a standard Williams & Beer redundancy lattice.

```
In [23]: PID_PM(bivariates['pnt. unq'])
Out[23]:
+--------+--------+--------+
|  I_pm  |  I_r   |   pi   |
+--------+--------+--------+
| {0:1}  | 1.0000 | 0.0000 |
|  {0}   | 0.5000 | 0.5000 |
|  {1}   | 0.5000 | 0.5000 |
| {0}{1} | 0.0000 | 0.0000 |
+--------+--------+--------+
```

I_{RAV}•

Taking a functional perspective as in $I_\wedge$, I_{RAV} defines bivariate redundancy as the maximum coinformation between the two sources $X_0, X_1', atarget:math:`Y$, and a deterministic function of the inputs $f(X_0, X_1)$.

$$I_{RAV} X_{0:2} : Y = \max_f \left(I X_0 \colon X_1 \colon Y \colon f(X_0, X_1) \right)$$

This measure is designed to exploit the conflation of synergy and redundancy in the three variable coinformation: $I X_0 \colon X_1 \colon Y = R - S$.

```
In [24]: PID_RAV(bivariates['pnt. unq'])
Out[24]:
+--------+--------+--------+
| I_RAV  |  I_r   |   pi   |
+--------+--------+--------+
| {0:1}  | 1.0000 | 0.0000 |
|  {0}   | 0.5000 | 0.5000 |
|  {1}   | 0.5000 | 0.5000 |
| {0}{1} | 0.0000 | 0.0000 |
+--------+--------+--------+
```

I_{RR}•

In order to combine I_{MMI}• with the coinformation, Goodwell and Kumar [GK17] have introduced their *rescaled redundancy*:

$$\begin{aligned} I_{RR} X_0 : X_1 &= R_{\text{min}} + I_S (I_{MMI} X_{0:2} : Y - R_{\text{min}} \\ R_{\text{min}} &= \max\{0, I X_0 : X_1 : Y\} \\ I_S &= \frac{I X_0 : X_1}{\min\{H X_0, H X_1\}} \end{aligned}$$

```
In [25]: PID_RR(bivariates['pnt. unq'])
Out[25]:
+--------+--------+--------+
|  I_rr  |  I_r   |   pi   |
+--------+--------+--------+
| {0:1}  | 1.0000 | 0.3333 |
|  {0}   | 0.5000 | 0.1667 |
|  {1}   | 0.5000 | 0.1667 |
| {0}{1} | 0.3333 | 0.3333 |
+--------+--------+--------+
```

1.12.4 Partial Entropy Decomposition

Ince *[Inc17b]* proposed applying the PID framework to decompose multivariate entropy (without considering information about a separate target variable). This *partial entropy decomposition* (PED), seeks to partition a mutlivariate entropy $HX_0, X_1, \ldots$ among the antichains of the variables. The PED perspective shows that bivariate mutual information is equal to the difference between redundant entropy and synergistic entropy.

$$IX_0 : X_1 = H\partial\{X_0\}, \{X_1\} - H\partial\{X_0, X_1\}$$

H_{cs}●

Taking a pointwise point of view, following I_{ccs}●, Ince has proposed a measure of redundant entropy based on the coinformation *[Inc17b]*:

$$H_{cs}X_{0:n} = \sum p(x_0, \ldots, x_n) Ix_0 : \ldots : x_n \text{ if } (Ix_0 : \ldots : x_n > 0)$$

While this measure behaves intuitively in many examples, it also assigns negative values to some partial entropy atoms in some instances. However, Ince *[Inc17b]* argues that concepts such as mechanistic information redundnacy (non-zero information redundancy between independent predictors, c.f. AND) necessitate negative partial entropy terms.

Like I_{ccs}●, H_{cs}● is also subadditive.

```
In [26]: PED_CS(dit.Distribution(['00','01','10','11'],[0.25]*4))
Out[26]:
+--------+--------+--------+
|  H_cs  |  H_r   |  H_d   |
+--------+--------+--------+
| {0:1}  | 2.0000 | 0.0000 |
|  {0}   | 1.0000 | 1.0000 |
|  {1}   | 1.0000 | 1.0000 |
| {0}{1} | 0.0000 | 0.0000 |
+--------+--------+--------+
```

1.13 References

1.14 Changelog

- : Basic functionality.
- #26: Add the Jensen-Shannon Divergence, a measure of distribution distance.
- #5: Add the oft-used Total Correlation.
- #10: Add the Co-Information.
- #30: Add the Gács-Körner Common Information.
- #7: Add the Residual Entropy.
- #6: Add the Binding Information.
- #2: Add the Extropy.

- #33: Add the Perplexity.
- #45: Add the Interaction Information.
- #47: Add the TSE Complexity.
- #16: Add the Channel Capacity.
- #1: Add the Cumulative Residual Entropy.
- #14: Add the creation of Minimal Sufficient Statistics.
- #35: Add the cross entropy.
- #34: Add the Kullback-Leibler Divergence.
- #3: Add the Renyi entropy.
- #4: Add the Tsallis entropy.
- #87: Add Renyi, Tsallis, Hellinger, and alpha divergences.
- #13: Add the Joint Minimal Sufficient Statistic.
- #95: Add the complexity profile.
- #96: Add the marginal utility of information.
- #63: Add scalar operations for ScalarDistributions.
- #108: Add the multivariate entropy triangle.
- #109: Add the CAEKL mutual information.
- #111: Add the functional common information.
- #9: Add the Wyner common information.
- #11: Add the exact common information.
- #15: Add the intrinsic mutual information.
- #32: Use `six` for python 2/3 compatibility.
- #40: Use an Enum for rv_mode.
- #75: Enable coveralls to display detailed coverage information.
- #77: Enable landscape.io to do static code analysis.
- #58: Alias Dual Total Correlation as Binding Information.

1.15 Indices and tables

- genindex
- modindex
- search

Bibliography

[AP12] S. A. Abdallah and M. D. Plumbley. A measure of statistical complexity based on predictive information with application to finite spin systems. *Physics Letters A*, 376(4):275–281, 2012.

[ASBY14] B. Allen, B. C. Stacey, and Y. Bar-Yam. An information-theoretic formalism for multiscale structure in complex systems. *arXiv preprint arXiv:1409.4708*, 2014.

[Ama01] Shun-ichi Amari. Information geometry on hierarchy of probability distributions. *Information Theory, IEEE Transactions on*, 47(5):1701–1711, 2001.

[BG15] Pradeep Kr Banerjee and Virgil Griffith. Synergy, redundancy and common information. *arXiv preprint arXiv:1509.03706*, 2015.

[BRMontufar17] Pradeep Kr. Banerjee, Johannes Rauh, and Guido Montúfar. Computing the unique information. *arXive preprint arXiv:1709.07487*, 2017.

[BY04] Y. Bar-Yam. Multiscale complexity/entropy. *Advances in Complex Systems*, 7(01):47–63, 2004.

[BG16] Salman Beigi and Amin Gohari. Phi-entropic measures of correlation. *arXiv preprint arXiv:1611.01335*, 2016.

[Bel03] A. J. Bell. The co-information lattice. In S. Makino S. Amari, A. Cichocki and N. Murata, editors, *Proc. Fifth Intl. Workshop on Independent Component Analysis and Blind Signal Separation*, volume ICA 2003, 921–926. New York, 2003. Springer.

[BROJ13] Nils Bertschinger, Johannes Rauh, Eckehard Olbrich, and Jürgen Jost. Shared information—new insights and problems in decomposing information in complex systems. In *Proceedings of the European Conference on Complex Systems 2012*, 251–269. Springer, 2013.

[BRO+14] Nils Bertschinger, Johannes Rauh, Eckehard Olbrich, Jürgen Jost, and Nihat Ay. Quantifying unique information. *Entropy*, 16(4):2161–2183, 2014.

[Cal02] Cristian Calude. *Information and Randomness: An Algorithmic Perspective*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 2002. ISBN 3540434666.

[CABE+15] Chung Chan, Ali Al-Bashabsheh, Javad B Ebrahimi, Tarik Kaced, and Tie Liu. Multivariate mutual information inspired by secret-key agreement. *Proceedings of the IEEE*, 103(10):1883–1913, 2015.

[CP16] Daniel Chicharro and Stefano Panzeri. Redundancy and synergy in dual decompositions of mutual information gain and information loss. *arXiv preprint arXiv:1612.09522*, 2016.

[CRW03] Matthias Christandl, Renato Renner, and Stefan Wolf. A property of the intrinsic mutual information. In *IEEE international symposium on information theory*, 258–258. 2003.

[CT06] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, New York, second edition, 2006. ISBN 0471241954.

[CPC10] Paul Warner Cuff, Haim H Permuter, and Thomas M Cover. Coordination capacity. *IEEE Transactions on Information Theory*, 56(9):4181–4206, 2010.

[FL17] Conor Finn and Joseph Lizier. 2017.

[FWA16] Seth Frey, Paul L Williams, and Dominic K Albino. Information encryption in the expert management of strategic uncertainty. *arXiv preprint arXiv:1605.04233*, 2016.

[GA17] Amin Gohari and Venkat Anantharam. Comments on "information-theoretic key agreement of multiple terminals—part i". *IEEE Transactions on Information Theory*, 63(8):5440–5442, 2017.

[GGunluK17] Amin Gohari, Onur Günlü, and Gerhard Kramer. On achieving a positive rate in the source model key agreement problem. *arXiv preprint arXiv:1709.05174*, 2017.

[GK17] Allison E Goodwell and Praveen Kumar. Temporal information partitioning: characterizing synergy, uniqueness, and redundancy in interacting environmental variables. *Water Resources Research*, 53(7):5920–5942, 2017.

[GCJ+14] Virgil Griffith, Edwin KP Chong, Ryan G James, Christopher J Ellison, and James P Crutchfield. Intersection information based on common randomness. *Entropy*, 16(4):1985–2000, 2014.

[GK14] Virgil Griffith and Christof Koch. Quantifying synergistic mutual information. In *Guided Self-Organization: Inception*, pages 159–190. Springer, 2014.

[GacsKorner73] Peter Gács and János Körner. Common information is far less than mutual information. *Problems of Control and Information Theory*, 2(2):149–162, 1973.

[Han75] T. S. Han. Linear dependence structure of the entropy space. *Information and Control*, 29:337–368, 1975.

[HSP13] Malte Harder, Christoph Salge, and Daniel Polani. Bivariate measure of redundant information. *Physical Review E*, 87(1):012130, 2013.

[Inc17a] Robin A. A. Ince. Measuring multivariate redundant information with pointwise common change in surprisal. *Entropy*, 19(7):318, 2017.

[Inc17b] Robin A. A. Ince. The Partial Entropy Decompostion: decomposing multivariate entropy and mutual information via pointwise common surprisal. *arXive preprint arXiv:1702.01591*, 2017.

[JEC17] Ryan G. James, Jeffrey Emenheiser, and James P. Crutchfield. Unique information via dependency constraints. *arXiv preprint arXiv:1709.06653*, 2017.

[Kri09] Klaus Krippendorff. Ross ashby's information theory: a bit of history, some solutions to problems, and what we face today. *International Journal of General Systems*, 38(2):189–212, 2009.

[KLEG14] G. R. Kumar, C. T. Li, and A. El Gamal. Exact common information. In *Information Theory (ISIT), 2014 IEEE International Symposium on*, 161–165. IEEE, 2014.

[LSAgro11] Frank Lad, Giuseppe Sanfilippo, and Gianna Agrò. Extropy: a complementary dual of entropy. *arXiv preprint arXiv:1109.6440*, 2011.

[LMPR04] P. W. Lamberti, M. T. Martin, A. Plastino, and O. A. Rosso. Intensive entropic non-triviality measure. *Physica A: Statistical Mechanics and its Applications*, 334(1):119–131, 2004.

[LXC10] Wei Liu, Ge Xu, and Biao Chen. The common information of n dependent random variables. In *Communication, Control, and Computing (Allerton), 2010 48th Annual Allerton Conference on*, 836–843. IEEE, 2010.

[LFW13] Joseph T Lizier, Benjamin Flecker, and Paul L Williams. Towards a synergy-based approach to measuring information modification. In *Artificial Life (ALIFE), 2013 IEEE Symposium on*, 43–51. IEEE, 2013.

[Mac03] David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[MW97] Ueli Maurer and Stefan Wolf. The intrinsic conditional mutual information and perfect secrecy. In *IEEE international symposium on information theory*, 88–88. Citeseer, 1997.

[McG54] W. J. McGill. Multivariate information transmission. *Psychometrika*, 19(2):97–116, 1954.

[OBR15] Eckehard Olbrich, Nils Bertschinger, and Johannes Rauh. Information decomposition and synergy. *Entropy*, 17(5):3501–3517, 2015.

[PVerdu08] Daniel P Palomar and Sergio Verdú. Lautum information. *IEEE transactions on information theory*, 54(3):964–975, 2008.

[PPCP17] Giuseppe Pica, Eugenio Piasini, Daniel Chicharro, and Stefano Panzeri. Invariant components of synergy, redundancy, and unique information among three variables. *arXiv preprint arXiv:1706.08921*, 2017.

[RCVW04] Murali Rao, Yunmei Chen, Baba C Vemuri, and Fei Wang. Cumulative residual entropy: a new measure of information. *Information Theory, IEEE Transactions on*, 50(6):1220–1228, 2004.

[Rau17] Johannes Rauh. Secret sharing and shared information. *arXiv preprint arXiv:1706.06998*, 2017.

[RBO+17] Johannes Rauh, Pradeep Kr Banerjee, Eckehard Olbrich, Jürgen Jost, and Nils Bertschinger. On extractable shared information. *arXiv preprint arXiv:1701.07805*, 2017.

[RBOJ14] Johannes Rauh, Nils Bertschinger, Eckehard Olbrich, and Jurgen Jost. Reconsidering unique information: towards a multivariate information decomposition. In *Information Theory (ISIT), 2014 IEEE International Symposium on*, 2232–2236. IEEE, 2014.

[RSW03] Renato Renner, Juraj Skripsky, and Stefan Wolf. A new measure for conditional mutual information and its properties. In *IEEE INTERNATIONAL SYMPOSIUM ON INFORMATION THEORY*, 259–259. 2003.

[RNE+16] Fernando Rosas, Vasilis Ntranos, Christopher J Ellison, Sofie Pollin, and Marian Verhelst. Understanding interdependency through complex information sharing. *Entropy*, 18(2):38, 2016.

[SSB+03] E. Schneidman, S. Still, M. J. Berry, W. Bialek, and others. Network information and connected correlations. *Phys. Rev. Lett.*, 91(23):238701, 2003.

[TS80] H. Te Sun. Multiple mutual informations and multiple interactions in frequency data. *Information and Control*, 46:26–45, 1980.

[TPB00] Naftali Tishby, Fernando C Pereira, and William Bialek. The information bottleneck method. *arXiv preprint physics/0004057*, 2000.

[TSE94] Giulio Tononi, Olaf Sporns, and Gerald M Edelman. A measure for brain complexity: relating functional segregation and integration in the nervous system. *Proceedings of the National Academy of Sciences*, 91(11):5033–5037, 1994.

[TNG11] H. Tyagi, P. Narayan, and P. Gupta. When is a function securely computable? *Information Theory, IEEE Transactions on*, 57(10):6337–6350, 2011.

[VAPelaezM16] Francisco José Valverde-Albacete and Carmen Peláez-Moreno. The multivariate entropy triangle and applications. In *Hybrid Artificial Intelligent Systems*, pages 647–658. Springer, 2016.

[VW08] Sergio Verdu and Tsachy Weissman. The information lost in erasures. *Information Theory, IEEE Transactions on*, 54(11):5030–5058, 2008.

[VerduW06] S. Verdú and T. Weissman. Erasure entropy. In *Information Theory, 2006 IEEE International Symposium on*, 98–102. IEEE, 2006.

[Wat60] S. Watanabe. Information theoretical analysis of multivariate correlation. *IBM Journal of research and development*, 4(1):66–82, 1960.

[WB10] Paul L Williams and Randall D Beer. Nonnegative decomposition of multivariate information. *arXiv preprint arXiv:1004.2515*, 2010.

[WB11] Paul L Williams and Randall D Beer. Generalized measures of information transfer. *arXiv preprint arXiv:1102.1507*, 2011.

[Wyn75] A. D. Wyner. The common information of two dependent random variables. *Information Theory, IEEE Transactions on*, 21(2):163–179, 1975.

[Yeu08] Raymond W Yeung. *Information theory and network coding*. Springer, 2008.

Python Module Index

Index