Sage Reference Manual: Statistics

Release 8.4

The Sage Development Team

Oct 18, 2018

CONTENTS

CHAPTER

ONE

BASIC STATISTICS

This file contains basic descriptive functions. Included are the mean, median, mode, moving average, standard deviation, and the variance. When calling a function on data, there are checks for functions already defined for that data type.

The `mean` function returns the arithmetic mean (the sum of all the members of a list, divided by the number of members). Further revisions may include the geometric and harmonic mean. The `median` function returns the number separating the higher half of a sample from the lower half. The `mode` returns the most common occuring member of a sample, plus the number of times it occurs. If entries occur equally common, the smallest of a list of the most common entries is returned. The `moving_average` is a finite impulse response filter, creating a series of averages using a user-defined number of subsets of the full data set. The `std` and the `variance` return a measurement of how far data points tend to be from the arithmetic mean.

Functions are available in the namespace `stats`, i.e. you can use them by typing `stats.mean`, `stats.median`, etc.

REMARK: If all the data you are working with are floating point numbers, you may find `finance.TimeSeries` helpful, since it is extremely fast and offers many of the same descriptive statistics as in the module.

AUTHOR:

- Andrew Hou (11/06/2009)

`sage.stats.basic_stats.`**`mean`**`(v)`

Return the mean of the elements of v.

We define the mean of the empty list to be the (symbolic) NaN, following the convention of MATLAB, Scipy, and R.

INPUT:

- v – a list of numbers

OUTPUT:

- a number

EXAMPLES:

```
sage: mean([pi, e])
1/2*pi + 1/2*e
sage: mean([])
NaN
sage: mean([I, sqrt(2), 3/5])
1/3*sqrt(2) + 1/3*I + 1/5
sage: mean([RIF(1.0103,1.0103), RIF(2)])
1.5051500000000000?
sage: mean(range(4))
```

(continues on next page)

(continued from previous page)

```
3/2
sage: v = finance.TimeSeries([1..100])
sage: mean(v)
50.5
```

`sage.stats.basic_stats.median(v)`

Return the median (middle value) of the elements of v

If v is empty, we define the median to be NaN, which is consistent with NumPy (note that R returns NULL). If v is comprised of strings, TypeError occurs. For elements other than numbers, the median is a result of `sorted()`.

INPUT:

- v – a list

OUTPUT:

- median element of v

EXAMPLES:

```
sage: median([1,2,3,4,5])
3
sage: median([e, pi])
1/2*pi + 1/2*e
sage: median(['sage', 'linux', 'python'])
'python'
sage: median([])
NaN
sage: class MyClass:
....:     def median(self):
....:         return 1
sage: stats.median(MyClass())
1
```

`sage.stats.basic_stats.mode(v)`

Return the mode of v.

The mode is the list of the most frequently occuring elements in v. If n is the most times that any element occurs in v, then the mode is the list of elements of v that occur n times. The list is sorted if possible.

Note: The elements of v must be hashable.

INPUT:

- v – a list

OUTPUT:

- a list (sorted if possible)

EXAMPLES:

```
sage: v = [1,2,4,1,6,2,6,7,1]
sage: mode(v)
[1]
sage: v.count(1)
3
```

(continues on next page)

(continued from previous page)

```
sage: mode([])
[]

sage: mode([1,2,3,4,5])
[1, 2, 3, 4, 5]
sage: mode([3,1,2,1,2,3])
[1, 2, 3]
sage: mode([0, 2, 7, 7, 13, 20, 2, 13])
[2, 7, 13]

sage: mode(['sage', 'four', 'I', 'three', 'sage', 'pi'])
['sage']

sage: class MyClass:
....:   def mode(self):
....:       return [1]
sage: stats.mode(MyClass())
[1]
```

`sage.stats.basic_stats.`**`moving_average`**`(`*v*, *n*`)`

Provides the moving average of a list v

The moving average of a list is often used to smooth out noisy data.

If v is empty, we define the entries of the moving average to be NaN.

INPUT:

- v – a list
- n – the number of values used in computing each average.

OUTPUT:

- a list of length `len(v)-n+1`, since we do not fabric any values

EXAMPLES:

```
sage: moving_average([1..10], 1)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: moving_average([1..10], 4)
[5/2, 7/2, 9/2, 11/2, 13/2, 15/2, 17/2]
sage: moving_average([], 1)
[]
sage: moving_average([pi, e, I, sqrt(2), 3/5], 2)
[1/2*pi + 1/2*e, 1/2*e + 1/2*I, 1/2*sqrt(2) + 1/2*I, 1/2*sqrt(2) + 3/10]
```

We check if the input is a time series, and if so use the optimized `simple_moving_average` method, but with (slightly different) meaning as defined above (the point is that the `simple_moving_average` on time series returns n values:

```
sage: a = finance.TimeSeries([1..10])
sage: stats.moving_average(a, 3)
[2.0000, 3.0000, 4.0000, 5.0000, 6.0000, 7.0000, 8.0000, 9.0000]
sage: stats.moving_average(list(a), 3)
[2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

`sage.stats.basic_stats.`**`std`**`(`*v*, *bias=False*`)`

Returns the standard deviation of the elements of v

We define the standard deviation of the empty list to be NaN, following the convention of MATLAB, Scipy, and R.

INPUT:

- v – a list of numbers
- **bias – bool (default: False); if False, divide by** len(v) - 1 instead of len(v) to give a less biased estimator (sample) for the standard deviation.

OUTPUT:

- a number

EXAMPLES:

```
sage: std([1..6], bias=True)
1/2*sqrt(35/3)
sage: std([1..6], bias=False)
sqrt(7/2)
sage: std([e, pi])
sqrt(1/2)*abs(pi - e)
sage: std([])
NaN
sage: std([I, sqrt(2), 3/5])
1/15*sqrt(1/2)*sqrt((10*sqrt(2) - 5*I - 3)^2 + (5*sqrt(2) - 10*I + 3)^2 +
↪(5*sqrt(2) + 5*I - 6)^2)
sage: std([RIF(1.0103, 1.0103), RIF(2)])
0.6998235813403261?
sage: import numpy
sage: x = numpy.array([1,2,3,4,5])
sage: std(x, bias=False)
1.5811388300841898
sage: x = finance.TimeSeries([1..100])
sage: std(x)
29.011491975882016
```

sage.stats.basic_stats.**variance**(*v*, *bias=False*)

Returns the variance of the elements of v

We define the variance of the empty list to be NaN, following the convention of MATLAB, Scipy, and R.

INPUT:

- v – a list of numbers
- **bias – bool (default: False); if False, divide by** len(v) - 1 instead of len(v) to give a less biased estimator (sample) for the standard deviation.

OUTPUT:

- a number

EXAMPLES:

```
sage: variance([1..6])
7/2
sage: variance([1..6], bias=True)
35/12
sage: variance([e, pi])
1/2*(pi - e)^2
sage: variance([])
```

(continues on next page)

(continued from previous page)

```
NaN
sage: variance([I, sqrt(2), 3/5])
1/450*(10*sqrt(2) - 5*I - 3)^2 + 1/450*(5*sqrt(2) - 10*I + 3)^2 + 1/
↪450*(5*sqrt(2) + 5*I - 6)^2
sage: variance([RIF(1.0103, 1.0103), RIF(2)])
0.4897530450000000?
sage: import numpy
sage: x = numpy.array([1,2,3,4,5])
sage: variance(x, bias=False)
2.5
sage: x = finance.TimeSeries([1..100])
sage: variance(x)
841.6666666666666
sage: variance(x, bias=True)
833.25
sage: class MyClass:
....:   def variance(self, bias = False):
....:      return 1
sage: stats.variance(MyClass())
1
sage: class SillyPythonList:
....:   def __init__(self):
....:       self.__list = [2, 4]
....:   def __len__(self):
....:       return len(self.__list)
....:   def __iter__(self):
....:       return self.__list.__iter__()
....:   def mean(self):
....:       return 3
sage: R = SillyPythonList()
sage: variance(R)
2
sage: variance(R, bias=True)
1
```

CHAPTER

TWO

C INT LISTS

This is a class for fast basic operations with lists of C ints. It is similar to the double precision TimeSeries class. It has all the standard C int semantics, of course, including overflow. It is also similar to the Python list class, except all elements are C ints, which makes some operations much, much faster. For example, concatenating two IntLists can be over 10 times faster than concatenating the corresponding Python lists of ints, and taking slices is also much faster.

AUTHOR:

- William Stein, 2010-03

class sage.stats.intlist.**IntList**

Bases: object

A list of C int's.

list()

Return Python list version of self with Python ints as entries.

EXAMPLES:

```
sage: a = stats.IntList([1..15]); a
[1, 2, 3, 4, 5 ... 11, 12, 13, 14, 15]
sage: a.list()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
sage: list(a) == a.list()
True
sage: type(a.list()[0])
<... 'int'>
```

max(*index=False*)

Return the largest value in this time series. If this series has length 0 we raise a ValueError

INPUT:

- index – bool (default: False); if True, also return index of maximum entry.

OUTPUT:

- int – largest value
- int – index of largest value; only returned if index=True

EXAMPLES:

```
sage: v = stats.IntList([1,-4,3,-2,-4,3])
sage: v.max()
3
sage: v.max(index=True)
(3, 2)
```

min (*index=False*)

Return the smallest value in this integer list. If this series has length 0 we raise a ValueError.

INPUT:

- index – bool (default: False); if True, also return index of minimal entry.

OUTPUT:

- float – smallest value
- integer – index of smallest value; only returned if index=True

EXAMPLES:

```
sage: v = stats.IntList([1,-4,3,-2,-4])
sage: v.min()
-4
sage: v.min(index=True)
(-4, 1)
```

plot (**args*, ***kwds*)

Return a plot of this IntList. This just constructs the corresponding double-precision floating point TimeSeries object, passing on all arguments.

EXAMPLES:

```
sage: stats.IntList([3,7,19,-2]).plot()
Graphics object consisting of 1 graphics primitive
sage: stats.IntList([3,7,19,-2]).plot(color='red',pointsize=50,points=True)
Graphics object consisting of 1 graphics primitive
```

plot_histogram (**args*, ***kwds*)

Return a histogram plot of this IntList. This just constructs the corresponding double-precision floating point TimeSeries object, and plots it, passing on all arguments.

EXAMPLES:

```
sage: stats.IntList([1..15]).plot_histogram()
Graphics object consisting of 50 graphics primitives
```

prod ()

Return the product of the entries of self.

EXAMPLES:

```
sage: a = stats.IntList([1..10]); a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: a.prod()
3628800
sage: factorial(10)
3628800
```

Note that there can be overflow:

```
sage: a = stats.IntList([2^30, 2]); a
[1073741824, 2]
sage: a.prod()
-2147483648
```

sum()

Return the sum of the entries of self.

EXAMPLES:

```
sage: stats.IntList([1..100]).sum()
5050
```

Note that there can be overflow, since the entries are C ints:

```
sage: a = stats.IntList([2^30,2^30]); a
[1073741824, 1073741824]
sage: a.sum()
-2147483648
```

time_series()

Return TimeSeries version of self, which involves changing each entry to a double.

EXAMPLES:

```
sage: T = stats.IntList([-2,3,5]).time_series(); T
[-2.0000, 3.0000, 5.0000]
sage: type(T)
<... 'sage.finance.time_series.TimeSeries'>
```

sage.stats.intlist.**unpickle_intlist_v1**

Version 1 unpickle method.

INPUT:

- v – a raw char buffer

EXAMPLES:

```
sage: v = stats.IntList([1,2,3])
sage: s = v.__reduce__()[1][0]
sage: type(s) == type(b'')
True
sage: sage.stats.intlist.unpickle_intlist_v1(s, 3)
[1, 2, 3]
sage: sage.stats.intlist.unpickle_intlist_v1(s+s,6)
[1, 2, 3, 1, 2, 3]
sage: sage.stats.intlist.unpickle_intlist_v1(b'',0)
[]
```

CHAPTER

THREE

HIDDEN MARKOV MODELS

This is a complete pure-Cython optimized implementation of Hidden Markov Models. It fully supports Discrete, Gaussian, and Mixed Gaussian emissions.

The best references for the basic HMM algorithms implemented here are:

- Tapas Kanungo's "Hidden Markov Models"
- **Jackson's HMM tutorial:** http://personal.ee.surrey.ac.uk/Personal/P.Jackson/tutorial/

LICENSE: Some of the code in this file is based on reading Kanungo's GPLv2+ implementation of discrete HMM's, hence the present code must be licensed with a GPLv2+ compatible license.

AUTHOR:

- William Stein, 2010-03

class sage.stats.hmm.hmm.**DiscreteHiddenMarkovModel**

Bases: *sage.stats.hmm.hmm.HiddenMarkovModel*

A discrete Hidden Markov model implemented using double precision floating point arithmetic.

INPUT:

- `A` – a list of lists or a square N x N matrix, whose (i,j) entry gives the probability of transitioning from state i to state j.
- `B` – a list of N lists or a matrix with N rows, such that B[i,k] gives the probability of emitting symbol k while in state i.
- `pi` – the probabilities of starting in each initial state, i.e,. pi[i] is the probability of starting in state i.
- `emission_symbols` – None or list (default: None); if None, the emission_symbols are the ints [0..N-1], where N is the number of states. Otherwise, they are the entries of the list emissions_symbols, which must all be hashable.
- `normalize` –bool (default: True); if given, input is normalized to define valid probability distributions, e.g., the entries of A are made nonnegative and the rows sum to 1, and the probabilities in pi are normalized.

EXAMPLES:

```
sage: m = hmm.DiscreteHiddenMarkovModel([[0.4,0.6],[0.1,0.9]], [[0.1,0.9],[0.5,0.
↪5]], [.5,.5]); m
Discrete Hidden Markov Model with 2 States and 2 Emissions
Transition matrix:
[0.4 0.6]
[0.1 0.9]
Emission matrix:
[0.1 0.9]
[0.5 0.5]
```

(continues on next page)

(continued from previous page)

```
Initial probabilities: [0.5000, 0.5000]
sage: m.log_likelihood([0,1,0,1,0,1])
-4.66693474691329...
sage: m.viterbi([0,1,0,1,0,1])
([1, 1, 1, 1, 1, 1], -5.378832842208748)
sage: m.baum_welch([0,1,0,1,0,1])
(0.0, 22)
sage: m  # rel tol 1e-10
Discrete Hidden Markov Model with 2 States and 2 Emissions
Transition matrix:
[1.0134345614745788e-70                    1.0]
[                   1.0 3.9974352713558623e-19]
Emission matrix:
[ 7.380221566254936e-54                    1.0]
[                   1.0 3.9974352626002193e-19]
Initial probabilities: [0.0000, 1.0000]
sage: m.sample(10)
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
sage: m.graph().plot()
Graphics object consisting of 6 graphics primitives
```

A 3-state model that happens to always outputs 'b':

```
sage: m = hmm.DiscreteHiddenMarkovModel([[1/3]*3]*3, [[0,1,0]]*3, [1/3]*3, ['a','b
↪','c'])
sage: m.sample(10)
['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b']
```

baum_welch(*obs*, *max_iter=100*, *log_likelihood_cutoff=0.0001*, *fix_emissions=False*)

Given an observation sequence obs, improve this HMM using the Baum-Welch algorithm to increase the probability of observing obs.

INPUT:

- `obs` – list of emissions
- `max_iter` – integer (default: 100) maximum number of Baum-Welch steps to take
- `log_likelihood_cutoff` – positive float (default: 1e-4); the minimal improvement in likelihood with respect to the last iteration required to continue. Relative value to log likelihood.
- `fix_emissions` – bool (default: False); if True, do not change emissions when updating

OUTPUT:

- changes the model in places, and returns the log likelihood and number of iterations.

EXAMPLES:

```
sage: m = hmm.DiscreteHiddenMarkovModel([[0.1,0.9],[0.9,0.1]], [[.5,.5],[0,
↪1]], [.2,.8])
sage: m.baum_welch([1,0]*20, log_likelihood_cutoff=0)
(0.0, 4)
sage: m  # rel tol 1e-14
Discrete Hidden Markov Model with 2 States and 2 Emissions
Transition matrix:
[1.3515269707707603e-51                    1.0]
[                   1.0                    0.0]
Emission matrix:
```

(continues on next page)

(continued from previous page)

```
[                    1.0 6.462537138850569e-52]
[                    0.0                   1.0]
Initial probabilities: [0.0000, 1.0000]
```

The following illustrates how Baum-Welch is only a local optimizer, i.e., the above model is far more likely to produce the sequence [1,0]*20 than the one we get below:

```
sage: m = hmm.DiscreteHiddenMarkovModel([[0.5,0.5],[0.5,0.5]], [[.5,.5],[.5,.
↪5]], [.5,.5])
sage: m.baum_welch([1,0]*20, log_likelihood_cutoff=0)
(-27.725887222397784, 1)
sage: m
Discrete Hidden Markov Model with 2 States and 2 Emissions
Transition matrix:
[0.5 0.5]
[0.5 0.5]
Emission matrix:
[0.5 0.5]
[0.5 0.5]
Initial probabilities: [0.5000, 0.5000]
```

We illustrate fixing emissions:

```
sage: m = hmm.DiscreteHiddenMarkovModel([[0.1,0.9],[0.9,0.1]], [[.5,.5],[.2,.
↪8]], [.2,.8])
sage: set_random_seed(0); v = m.sample(100)
sage: m.baum_welch(v,fix_emissions=True)
(-66.98630856918774, 100)
sage: m.emission_matrix()
[0.5 0.5]
[0.2 0.8]
sage: m = hmm.DiscreteHiddenMarkovModel([[0.1,0.9],[0.9,0.1]], [[.5,.5],[.2,.
↪8]], [.2,.8])
sage: m.baum_welch(v)
(-66.782360659293..., 100)
sage: m.emission_matrix()  # rel tol 1e-14
[ 0.5303085748626447 0.46969142513735535]
[ 0.2909775550173978  0.7090224449826023]
```

emission_matrix()

Return the matrix whose i-th row specifies the emission probability distribution for the i-th state. More precisely, the i,j entry of the matrix is the probability of the Markov model outputing the j-th symbol when it is in the i-th state.

OUTPUT:

- a Sage matrix with real double precision (RDF) entries.

EXAMPLES:

```
sage: m = hmm.DiscreteHiddenMarkovModel([[0.4,0.6],[0.1,0.9]], [[0.1,0.9],[0.
↪5,0.5]], [.5,.5])
sage: E = m.emission_matrix(); E
[0.1 0.9]
[0.5 0.5]
```

The returned matrix is mutable, but changing it does not change the transition matrix for the model:

```
sage: E[0,0] = 0; E[0,1] = 1
sage: m.emission_matrix()
[0.1 0.9]
[0.5 0.5]
```

generate_sequence(*length*, *starting_state=None*)

Return a sample of the given length from this HMM.

INPUT:

- `length` – positive integer
- `starting_state` – int (or None); if specified then generate a sequence using this model starting with the given state instead of the initial probabilities to determine the starting state.

OUTPUT:

- an IntList or list of emission symbols
- IntList of the actual states the model was in when emitting the corresponding symbols

EXAMPLES:

In this example, the emission symbols are not set:

```
sage: set_random_seed(0)
sage: a = hmm.DiscreteHiddenMarkovModel([[0.1,0.9],[0.1,0.9]], [[1,0],[0,1]],␣
↪[0,1])
sage: a.generate_sequence(5)
([1, 0, 1, 1, 1], [1, 0, 1, 1, 1])
sage: list(a.generate_sequence(1000)[0]).count(0)
90
```

Here the emission symbols are set:

```
sage: set_random_seed(0)
sage: a = hmm.DiscreteHiddenMarkovModel([[0.5,0.5],[0.1,0.9]], [[1,0],[0,1]],␣
↪[0,1], ['up', 'down'])
sage: a.generate_sequence(5)
(['down', 'up', 'down', 'down', 'down'], [1, 0, 1, 1, 1])
```

Specify the starting state:

```
sage: set_random_seed(0); a.generate_sequence(5, starting_state=0)
(['up', 'up', 'down', 'down', 'down'], [0, 0, 1, 1, 1])
```

log_likelihood(*obs*, *scale=True*)

Return the logarithm of the probability that this model produced the given observation sequence. Thus the output is a non-positive number.

INPUT:

- `obs` – sequence of observations
- `scale` – boolean (default: True); if True, use rescaling to overoid loss of precision due to the very limited dynamic range of floats. You should leave this as True unless the obs sequence is very small.

EXAMPLES:

```
sage: m = hmm.DiscreteHiddenMarkovModel([[0.4,0.6],[0.1,0.9]], [[0.1,0.9],[0.
↪5,0.5]], [.2,.8])
sage: m.log_likelihood([0, 1, 0, 1, 1, 0, 1, 0, 0, 0])
-7.3301308009370825
sage: m.log_likelihood([0, 1, 0, 1, 1, 0, 1, 0, 0, 0], scale=False)
-7.330130800937082
sage: m.log_likelihood([])
0.0

sage: m = hmm.DiscreteHiddenMarkovModel([[0.4,0.6],[0.1,0.9]], [[0.1,0.9],[0.
↪5,0.5]], [.2,.8], ['happy','sad'])
sage: m.log_likelihood(['happy','happy'])
-1.6565295199679506
sage: m.log_likelihood(['happy','sad'])
-1.4731602941415523
```

Overflow from not using the scale option:

```
sage: m = hmm.DiscreteHiddenMarkovModel([[0.4,0.6],[0.1,0.9]], [[0.1,0.9],[0.
↪5,0.5]], [.2,.8])
sage: m.log_likelihood([0,1]*1000, scale=True)
-1433.820666652728
sage: m.log_likelihood([0,1]*1000, scale=False)
-inf
```

viterbi (*obs*, *log_scale=True*)

Determine “the” hidden sequence of states that is most likely to produce the given sequence seq of observations, along with the probability that this hidden sequence actually produced the observation.

INPUT:

- `seq` – sequence of emitted ints or symbols
- `log_scale` – bool (default: True) whether to scale the sequence in order to avoid numerical overflow.

OUTPUT:

- `list` – “the” most probable sequence of hidden states, i.e., the Viterbi path.
- `float` – log of probability that the observed sequence was produced by the Viterbi sequence of states.

EXAMPLES:

```
sage: a = hmm.DiscreteHiddenMarkovModel([[0.1,0.9],[0.1,0.9]], [[0.9,0.1],[0.
↪1,0.9]], [0.5,0.5])
sage: a.viterbi([1,0,0,1,0,0,1,1])
([1, 0, 0, 1, ..., 0, 1, 1], -11.06245322477221...)
```

We predict the state sequence when the emissions are 3/4 and ‘abc’.:

```
sage: a = hmm.DiscreteHiddenMarkovModel([[0.1,0.9],[0.1,0.9]], [[0.9,0.1],[0.
↪1,0.9]], [0.5,0.5], [3/4, 'abc'])
```

Note that state 0 is common below, despite the model trying hard to switch to state 1:

```
sage: a.viterbi([3/4, 'abc', 'abc'] + [3/4]*10)
([0, 1, 1, 0, 0 ... 0, 0, 0, 0, 0], -25.299405845367794)
```

class sage.stats.hmm.hmm.**HiddenMarkovModel**

Bases: object

Abstract base class for all Hidden Markov Models.

graph (*eps=0.001*)

Create a weighted directed graph from the transition matrix, not including any edge with a probability less than eps.

INPUT:

- eps – nonnegative real number

OUTPUT:

- a digraph

EXAMPLES:

```
sage: m = hmm.DiscreteHiddenMarkovModel([[.3,0,.7],[0,0,1],[.5,.5,0]], [[.5,.
↪5,.2]]*3, [1/3]*3)
sage: G = m.graph(); G
Looped digraph on 3 vertices
sage: G.edges()
[(0, 0, 0.3), (0, 2, 0.7), (1, 2, 1.0), (2, 0, 0.5), (2, 1, 0.5)]
sage: G.plot()
Graphics object consisting of 11 graphics primitives
```

initial_probabilities ()

Return the initial probabilities, which as a TimeSeries of length N, where N is the number of states of the Markov model.

EXAMPLES:

```
sage: m = hmm.DiscreteHiddenMarkovModel([[0.4,0.6],[0.1,0.9]], [[0.1,0.9],[0.
↪5,0.5]], [.2,.8])
sage: pi = m.initial_probabilities(); pi
[0.2000, 0.8000]
sage: type(pi)
<... 'sage.finance.time_series.TimeSeries'>
```

The returned time series is a copy, so changing it does not change the model.

sage: pi[0] = .1; pi[1] = .9 sage: m.initial_probabilities() [0.2000, 0.8000]

Some other models:

```
sage: hmm.GaussianHiddenMarkovModel([[.1,.9],[.5,.5]], [(1,1), (-1,1)], [.1,.
↪9]).initial_probabilities()
[0.1000, 0.9000]
sage: hmm.GaussianMixtureHiddenMarkovModel([[.9,.1],[.4,.6]], [[(.4,(0,1)), (.
↪6,(1,0.1))],[(1,(0,1))]], [.7,.3]).initial_probabilities()
[0.7000, 0.3000]
```

sample (*length, number=None, starting_state=None*)

Return number samples from this HMM of given length.

INPUT:

- length – positive integer
- number – (default: None) if given, compute list of this many sample sequences

- `starting_state` – int (or None); if specified then generate a sequence using this model starting with the given state instead of the initial probabilities to determine the starting state.

OUTPUT:

- if number is not given, return a single TimeSeries.
- if number is given, return a list of TimeSeries.

EXAMPLES:

```
sage: set_random_seed(0)
sage: a = hmm.DiscreteHiddenMarkovModel([[0.1,0.9],[0.1,0.9]], [[1,0],[0,1]],␣
→[0,1])
sage: print(a.sample(10, 3))
[[1, 0, 1, 1, 1, 1, 0, 1, 1, 1], [1, 1, 0, 0, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1,␣
→0, 1, 0, 1, 1, 1]]
sage: a.sample(15)
[1, 1, 1, 1, 0 ... 1, 1, 1, 1, 1]
sage: a.sample(3, 1)
[[1, 1, 1]]
sage: list(a.sample(1000)).count(0)
88
```

If the emission symbols are set:

```
sage: set_random_seed(0)
sage: a = hmm.DiscreteHiddenMarkovModel([[0.5,0.5],[0.1,0.9]], [[1,0],[0,1]],␣
→[0,1], ['up', 'down'])
sage: a.sample(10)
['down', 'up', 'down', 'down', 'down', 'down', 'up', 'up', 'up', 'up']
```

Force a starting state:

```
sage: set_random_seed(0); a.sample(10, starting_state=0)
['up', 'up', 'down', 'down', 'down', 'down', 'up', 'up', 'up', 'up']
```

transition_matrix()

Return the state transition matrix.

OUTPUT:

- a Sage matrix with real double precision (RDF) entries.

EXAMPLES:

```
sage: M = hmm.DiscreteHiddenMarkovModel([[0.7,0.3],[0.9,0.1]], [[0.5,.5],[.1,.
→9]], [0.3,0.7])
sage: T = M.transition_matrix(); T
[0.7 0.3]
[0.9 0.1]
```

The returned matrix is mutable, but changing it does not change the transition matrix for the model:

```
sage: T[0,0] = .1; T[0,1] = .9
sage: M.transition_matrix()
[0.7 0.3]
[0.9 0.1]
```

Transition matrices for other types of models:

```
sage: hmm.GaussianHiddenMarkovModel([[.1,.9],[.5,.5]], [(1,1), (-1,1)], [.5,.
↪5]).transition_matrix()
[0.1 0.9]
[0.5 0.5]
sage: hmm.GaussianMixtureHiddenMarkovModel([[.9,.1],[.4,.6]], [[(.4,(0,1)), (.
↪6,(1,0.1))],[(1,(0,1))]], [.7,.3]).transition_matrix()
[0.9 0.1]
[0.4 0.6]
```

sage.stats.hmm.hmm.**unpickle_discrete_hmm_v0**(*A*, *B*, *pi*, *emission_symbols*, *name*)

sage.stats.hmm.hmm.**unpickle_discrete_hmm_v1**(*A*, *B*, *pi*, *n_out*, *emission_symbols*, *emission_symbols_dict*)

Return a *DiscreteHiddenMarkovModel*, restored from the arguments.

This function is used internally for unpickling.

CHAPTER

FOUR

CONTINUOUS EMISSION HIDDEN MARKOV MODELS

AUTHOR:

- William Stein, 2010-03

class sage.stats.hmm.chmm.**GaussianHiddenMarkovModel**

Bases: *sage.stats.hmm.hmm.HiddenMarkovModel*

GaussianHiddenMarkovModel(A, B, pi)

Gaussian emissions Hidden Markov Model.

INPUT:

- A – matrix; the N x N transition matrix
- B – list of pairs (mu,sigma) that define the distributions
- pi – initial state probabilities
- normalize –bool (default: True)

EXAMPLES:

We illustrate the primary functions with an example 2-state Gaussian HMM:

```
sage: m = hmm.GaussianHiddenMarkovModel([[.1,.9],[.5,.5]], [(1,1), (-1,1)], [.5,.
↪5]); m
Gaussian Hidden Markov Model with 2 States
Transition matrix:
[0.1 0.9]
[0.5 0.5]
Emission parameters:
[(1.0, 1.0), (-1.0, 1.0)]
Initial probabilities: [0.5000, 0.5000]
```

We query the defining transition matrix, emission parameters, and initial state probabilities:

```
sage: m.transition_matrix()
[0.1 0.9]
[0.5 0.5]
sage: m.emission_parameters()
[(1.0, 1.0), (-1.0, 1.0)]
sage: m.initial_probabilities()
[0.5000, 0.5000]
```

We obtain a sample sequence with 10 entries in it, and compute the logarithm of the probability of obtaining his sequence, given the model:

```
sage: obs = m.sample(10); obs
[-1.6835, 0.0635, -2.1688, 0.3043, -0.3188, -0.7835, 1.0398, -1.3558, 1.0882, 0.
↪4050]
sage: m.log_likelihood(obs)
-15.2262338077988...
```

We compute the Viterbi path, and probability that the given path of states produced obs:

```
sage: m.viterbi(obs)
([1, 0, 1, 0, 1, 1, 0, 1, 0, 1], -16.67738270170788)
```

We use the Baum-Welch iterative algorithm to find another model for which our observation sequence is more likely:

```
sage: m.baum_welch(obs)
(-10.6103334957397..., 14)
sage: m.log_likelihood(obs)
-10.6103334957397...
```

Notice that running Baum-Welch changed our model:

```
sage: m  # rel tol 3e-14
Gaussian Hidden Markov Model with 2 States
Transition matrix:
[   0.4154981366185841     0.584501863381416]
[   0.9999993174253741 6.825746258991804e-07]
Emission parameters:
[(0.4178882427119503, 0.5173109664360919), (-1.5025208631331122, 0.
↪5085512836055119)]
Initial probabilities: [0.0000, 1.0000]
```

baum_welch(*obs*, *max_iter=500*, *log_likelihood_cutoff=0.0001*, *min_sd=0.01*, *fix_emissions=False*, *v=False*)

Given an observation sequence obs, improve this HMM using the Baum-Welch algorithm to increase the probability of observing obs.

INPUT:

- obs – a time series of emissions
- max_iter – integer (default: 500) maximum number of Baum-Welch steps to take
- log_likelihood_cutoff – positive float (default: 1e-4); the minimal improvement in likelihood with respect to the last iteration required to continue. Relative value to log likelihood.
- min_sd – positive float (default: 0.01); when reestimating, the standard deviation of emissions is not allowed to be less than min_sd.
- fix_emissions – bool (default: False); if True, do not change emissions when updating

OUTPUT:

- changes the model in places, and returns the log likelihood and number of iterations.

EXAMPLES:

```
sage: m = hmm.GaussianHiddenMarkovModel([[.1,.9],[.5,.5]], [(1,.5), (-1,3)],␣
↪[.1,.9])
sage: m.log_likelihood([-2,-1,.1,0.1])
-8.858282215986275
```

(continues on next page)

(continued from previous page)

```
sage: m.baum_welch([-2,-1,.1,0.1])
(4.534646052182..., 7)
sage: m.log_likelihood([-2,-1,.1,0.1])
4.534646052182...
sage: m  # rel tol 3e-14
Gaussian Hidden Markov Model with 2 States
Transition matrix:
[   0.9999999992430161 7.569839394440382e-10]
[  0.49998462791192644    0.5000153720880736]
Emission parameters:
[(0.09999999999999999, 0.01), (-1.4999508147591902, 0.5000710504895474)]
Initial probabilities: [0.0000, 1.0000]
```

We illustrate bounding the standard deviation below. Note that above we had different emission parameters when the min_sd was the default of 0.01:

```
sage: m = hmm.GaussianHiddenMarkovModel([[.1,.9],[.5,.5]], [(1,.5), (-1,3)],␣
↪[.1,.9])
sage: m.baum_welch([-2,-1,.1,0.1], min_sd=1)
(-4.07939572755..., 32)
sage: m.emission_parameters()
[(-0.2663018798..., 1.0), (-1.99850979..., 1.0)]
```

We watch the log likelihoods of the model converge, step by step:

```
sage: m = hmm.GaussianHiddenMarkovModel([[.1,.9],[.5,.5]], [(1,.5), (-1,3)],␣
↪[.1,.9])
sage: v = m.sample(10)
sage: stats.TimeSeries([m.baum_welch(v,max_iter=1)[0] for _ in range(len(v))])
[-20.1167, -17.7611, -16.9814, -16.9364, -16.9314, -16.9309, -16.9309, -16.
↪9309, -16.9309, -16.9309]
```

We illustrate fixing emissions:

```
sage: m = hmm.GaussianHiddenMarkovModel([[.1,.9],[.9,.1]], [(1,2),(-1,.5)], [.
↪3,.7])
sage: set_random_seed(0); v = m.sample(100)
sage: m.baum_welch(v,fix_emissions=True)
(-164.72944548204..., 23)
sage: m.emission_parameters()
[(1.0, 2.0), (-1.0, 0.5)]
sage: m = hmm.GaussianHiddenMarkovModel([[.1,.9],[.9,.1]], [(1,2),(-1,.5)], [.
↪3,.7])
sage: m.baum_welch(v)
(-162.854370397998..., 49)
sage: m.emission_parameters()  # rel tol 3e-14
[(1.2722419172602375, 2.371368751761901), (-0.9486174675179113, 0.
↪5762360385123765)]
```

emission_parameters()

Return the parameters that define the normal distributions associated to all of the states.

OUTPUT:

- a list B of pairs B[i] = (mu, std), such that the distribution associated to state i is normal with mean mu and standard deviation std.

EXAMPLES:

```
sage: hmm.GaussianHiddenMarkovModel([[.1,.9],[.5,.5]], [(1,.5), (-1,3)], [.1,.
↪9]).emission_parameters()
[(1.0, 0.5), (-1.0, 3.0)]
```

generate_sequence (*length*, *starting_state=None*)

Return a sample of the given length from this HMM.

INPUT:

- length – positive integer
- starting_state – int (or None); if specified then generate a sequence using this model starting with the given state instead of the initial probabilities to determine the starting state.

OUTPUT:

- an IntList or list of emission symbols
- TimeSeries of emissions

EXAMPLES:

```
sage: m = hmm.GaussianHiddenMarkovModel([[.1,.9],[.5,.5]], [(1,.5), (-1,3)],␣
↪[.1,.9])
sage: m.generate_sequence(5)
([-3.0505, 0.5317, -4.5065, 0.6521, 1.0435], [1, 0, 1, 0, 1])
sage: m.generate_sequence(0)
([], [])
sage: m.generate_sequence(-1)
Traceback (most recent call last):
...
ValueError: length must be nonnegative
```

Example in which the starting state is 0 (see trac ticket #11452):

```
sage: set_random_seed(23);  m.generate_sequence(2)
([0.6501, -2.0151], [0, 1])
```

Force a starting state of 1 even though as we saw above it would be 0:

```
sage: set_random_seed(23);  m.generate_sequence(2, starting_state=1)
([-3.1491, -1.0244], [1, 1])
```

Verify numerically that the starting state is 0 with probability about 0.1:

```
sage: set_random_seed(0)
sage: v = [m.generate_sequence(1)[1][0] for i in range(10^5)]
sage: 1.0 * v.count(int(0)) / len(v)
0.0998200000000000
```

log_likelihood (*obs*)

Return the logarithm of a continuous analogue of the probability that this model produced the given observation sequence.

Note that the "continuous analogue of the probability" above can be bigger than 1, hence the logarithm can be positive.

INPUT:

- obs – sequence of observations

OUTPUT:

- float

EXAMPLES:

```
sage: m = hmm.GaussianHiddenMarkovModel([[.1,.9],[.5,.5]], [(1,.5), (-1,3)],
↪[.1,.9])
sage: m.log_likelihood([1,1,1])
-4.297880766072486
sage: set_random_seed(0); s = m.sample(20)
sage: m.log_likelihood(s)
-40.115714129484...
```

viterbi (*obs*)

Determine "the" hidden sequence of states that is most likely to produce the given sequence seq of observations, along with the probability that this hidden sequence actually produced the observation.

INPUT:

- seq – sequence of emitted ints or symbols

OUTPUT:

- list – "the" most probable sequence of hidden states, i.e., the Viterbi path.
- float – log of probability that the observed sequence was produced by the Viterbi sequence of states.

EXAMPLES:

We find the optimal state sequence for a given model:

```
sage: m = hmm.GaussianHiddenMarkovModel([[0.5,0.5],[0.5,0.5]], [(0,1),(10,1)],
↪ [0.5,0.5])
sage: m.viterbi([0,1,10,10,1])
([0, 0, 1, 1, 0], -9.0604285688230...)
```

Another example in which the most likely states change based on the last observation:

```
sage: m = hmm.GaussianHiddenMarkovModel([[.1,.9],[.5,.5]], [(1,.5), (-1,3)],
↪[.1,.9])
sage: m.viterbi([-2,-1,.1,0.1])
([1, 1, 0, 1], -9.61823698847639...)
sage: m.viterbi([-2,-1,.1,0.3])
([1, 1, 1, 0], -9.566023653378513)
```

class sage.stats.hmm.chmm.**GaussianMixtureHiddenMarkovModel**

Bases: *sage.stats.hmm.chmm.GaussianHiddenMarkovModel*

GaussianMixtureHiddenMarkovModel(A, B, pi)

Gaussian mixture Hidden Markov Model.

INPUT:

- A – matrix; the N x N transition matrix
- B – list of mixture definitions for each state. Each state may have a varying number of gaussians with selection probabilities that sum to 1 and encoded as (p,(mu,sigma))
- pi – initial state probabilities
- normalize –bool (default: True); if given, input is normalized to define valid probability distributions, e.g., the entries of A are made nonnegative and the rows sum to 1, and the probabilities in pi are normalized.

EXAMPLES:

```
sage: A  = [[0.5,0.5],[0.5,0.5]]
sage: B  = [[(0.9,(0.0,1.0)), (0.1,(1,10000))],[(1,(1,1)), (0,(0,0.1))]]
sage: hmm.GaussianMixtureHiddenMarkovModel(A, B, [1,0])
Gaussian Mixture Hidden Markov Model with 2 States
Transition matrix:
[0.5 0.5]
[0.5 0.5]
Emission parameters:
[0.9*N(0.0,1.0) + 0.1*N(1.0,10000.0), 1.0*N(1.0,1.0) + 0.0*N(0.0,0.1)]
Initial probabilities: [1.0000, 0.0000]
```

baum_welch(*obs*, *max_iter=1000*, *log_likelihood_cutoff=1e-12*, *min_sd=0.01*, *fix_emissions=False*)

Given an observation sequence obs, improve this HMM using the Baum-Welch algorithm to increase the probability of observing obs.

INPUT:

- obs – a time series of emissions
- max_iter – integer (default: 1000) maximum number of Baum-Welch steps to take
- log_likelihood_cutoff – positive float (default: 1e-12); the minimal improvement in likelihood with respect to the last iteration required to continue. Relative value to log likelihood.
- min_sd – positive float (default: 0.01); when reestimating, the standard deviation of emissions is not allowed to be less than min_sd.
- fix_emissions – bool (default: False); if True, do not change emissions when updating

OUTPUT:

- changes the model in places, and returns the log likelihood and number of iterations.

EXAMPLES:

```
sage: m = hmm.GaussianMixtureHiddenMarkovModel([[.9,.1],[.4,.6]], [[(.4,(0,
↪1)), (.6,(1,0.1))],[(1,(0,1))]], [.7,.3])
sage: set_random_seed(0); v = m.sample(10); v
[0.3576, -0.9365, 0.9449, -0.6957, 1.0217, 0.9644, 0.9987, -0.5950, -1.0219,␣
↪0.6477]
sage: m.log_likelihood(v)
-8.31408655939536...
sage: m.baum_welch(v)
(2.18905068682..., 15)
sage: m.log_likelihood(v)
2.18905068682...
sage: m  # rel tol 6e-12
Gaussian Mixture Hidden Markov Model with 2 States
Transition matrix:
[   0.8746363339773399   0.12536366602266016]
[                  1.0 1.451685202290174e-40]
Emission parameters:
[0.500161629343*N(-0.812298726239,0.173329026744) + 0.499838370657*N(0.
↪982433690378,0.029719932009), 1.0*N(0.503260056832,0.145881515324)]
Initial probabilities: [0.0000, 1.0000]
```

We illustrate bounding the standard deviation below. Note that above we had different emission parameters when the min_sd was the default of 0.01:

```
sage: m = hmm.GaussianMixtureHiddenMarkovModel([[.9,.1],[.4,.6]], [[(.4,(0,
↪1)), (.6,(1,0.1))],[(1,(0,1))]], [.7,.3])
sage: m.baum_welch(v, min_sd=1)
(-12.617885761692..., 1000)
sage: m.emission_parameters()  # rel tol 6e-12
[0.503545634447*N(0.200166509595,1.0) + 0.496454365553*N(0.200166509595,1.0),␣
↪1.0*N(0.0543433426535,1.0)]
```

We illustrate fixing all emissions:

```
sage: m = hmm.GaussianMixtureHiddenMarkovModel([[.9,.1],[.4,.6]], [[(.4,(0,
↪1)), (.6,(1,0.1))],[(1,(0,1))]], [.7,.3])
sage: set_random_seed(0); v = m.sample(10)
sage: m.baum_welch(v, fix_emissions=True)
(-7.58656858997..., 36)
sage: m.emission_parameters()
[0.4*N(0.0,1.0) + 0.6*N(1.0,0.1), 1.0*N(0.0,1.0)]
```

emission_parameters()

Returns a list of all the emission distributions.

OUTPUT:

- list of Gaussian mixtures

EXAMPLES:

```
sage: m = hmm.GaussianMixtureHiddenMarkovModel([[.9,.1],[.4,.6]], [[(.4,(0,
↪1)), (.6,(1,0.1))],[(1,(0,1))]], [.7,.3])
sage: m.emission_parameters()
[0.4*N(0.0,1.0) + 0.6*N(1.0,0.1), 1.0*N(0.0,1.0)]
```

sage.stats.hmm.chmm.**unpickle_gaussian_hmm_v0**(*A, B, pi, name*)

EXAMPLES:

```
sage: m = hmm.GaussianHiddenMarkovModel([[1]], [(0,1)], [1])
sage: sage.stats.hmm.chmm.unpickle_gaussian_hmm_v0(m.transition_matrix(), m.
↪emission_parameters(), m.initial_probabilities(), 'test')
Gaussian Hidden Markov Model with 1 States
Transition matrix:
[1.0]
Emission parameters:
[(0.0, 1.0)]
Initial probabilities: [1.0000]
```

sage.stats.hmm.chmm.**unpickle_gaussian_hmm_v1**(*A, B, pi, prob, n_out*)

EXAMPLES:

```
sage: m = hmm.GaussianHiddenMarkovModel([[1]], [(0,1)], [1])
sage: loads(dumps(m)) == m   # indirect test
True
```

sage.stats.hmm.chmm.**unpickle_gaussian_mixture_hmm_v1**(*A, B, pi, mixture*)

EXAMPLES:

```
sage: m = hmm.GaussianMixtureHiddenMarkovModel([[1]], [[(.4,(0,1)), (.6,(1,0.
↪1))]], [1])
sage: loads(dumps(m)) == m   # indirect test
```

(continues on next page)

(continued from previous page)

```
True
```

CHAPTER

FIVE

DISTRIBUTIONS USED IN IMPLEMENTING HIDDEN MARKOV MODELS

These distribution classes are designed specifically for HMM's and not for general use in statistics. For example, they have fixed or non-fixed status, which only make sense relative to being used in a hidden Markov model.

AUTHOR:

- William Stein, 2010-03

class sage.stats.hmm.distributions.**DiscreteDistribution**
Bases: *sage.stats.hmm.distributions.Distribution*

class sage.stats.hmm.distributions.**Distribution**
Bases: object

A distribution.

plot (**args*, ***kwds*)
Return a plot of the probability density function.

INPUT:

- args and kwds, passed to the Sage plot function

OUTPUT:

- a Graphics object

EXAMPLES:

```
sage: P = hmm.GaussianMixtureDistribution([(.2,-10,.5),(.6,1,1),(.2,20,.5)])
sage: P.plot(-10,30)
Graphics object consisting of 1 graphics primitive
```

prob (*x*)
The probability density function evaluated at x.

INPUT:

- x – object

OUTPUT:

- float

EXAMPLES:

This method must be defined in a derived class:

```
sage: import sage.stats.hmm.distributions
sage: sage.stats.hmm.distributions.Distribution().prob(0)
Traceback (most recent call last):
...
NotImplementedError
```

sample (*n=None*)

Return either a single sample (the default) or n samples from this probability distribution.

INPUT:

- n – None or a positive integer

OUTPUT:

- a single sample if n is 1; otherwise many samples

EXAMPLES:

This method must be defined in a derived class:

```
sage: import sage.stats.hmm.distributions
sage: sage.stats.hmm.distributions.Distribution().sample()
Traceback (most recent call last):
...
NotImplementedError
```

class sage.stats.hmm.distributions.**GaussianDistribution**

Bases: *sage.stats.hmm.distributions.Distribution*

class sage.stats.hmm.distributions.**GaussianMixtureDistribution**

Bases: *sage.stats.hmm.distributions.Distribution*

A probability distribution defined by taking a weighted linear combination of Gaussian distributions.

EXAMPLES:

```
sage: P = hmm.GaussianMixtureDistribution([(.3,1,2),(.7,-1,1)]); P
0.3*N(1.0,2.0) + 0.7*N(-1.0,1.0)
sage: P[0]
(0.3, 1.0, 2.0)
sage: P.is_fixed()
False
sage: P.fix(1)
sage: P.is_fixed(0)
False
sage: P.is_fixed(1)
True
sage: P.unfix(1)
sage: P.is_fixed(1)
False
```

fix (*i=None*)

Set that this GaussianMixtureDistribution (or its ith component) is fixed when using Baum-Welch to update the corresponding HMM.

INPUT:

- i – None (default) or integer; if given, only fix the i-th component

EXAMPLES:

```
sage: P = hmm.GaussianMixtureDistribution([(.2,-10,.5),(.6,1,1),(.2,20,.5)])
sage: P.fix(1); P.is_fixed()
False
sage: P.is_fixed(1)
True
sage: P.fix(); P.is_fixed()
True
```

is_fixed(*i=None*)

Return whether or not this GaussianMixtureDistribution is fixed when using Baum-Welch to update the corresponding HMM.

INPUT:

- i – None (default) or integer; if given, only return whether the i-th component is fixed

EXAMPLES:

```
sage: P = hmm.GaussianMixtureDistribution([(.2,-10,.5),(.6,1,1),(.2,20,.5)])
sage: P.is_fixed()
False
sage: P.is_fixed(0)
False
sage: P.fix(0); P.is_fixed()
False
sage: P.is_fixed(0)
True
sage: P.fix(); P.is_fixed()
True
```

prob(*x*)

Return the probability of x.

Since this is a continuous distribution, this is defined to be the limit of the p's such that the probability of [x,x+h] is p*h.

INPUT:

- x – float

OUTPUT:

- float

EXAMPLES:

```
sage: P = hmm.GaussianMixtureDistribution([(.2,-10,.5),(.6,1,1),(.2,20,.5)])
sage: P.prob(.5)
0.21123919605857971
sage: P.prob(-100)
0.0
sage: P.prob(20)
0.1595769121605731
```

prob_m(*x*, *m*)

Return the probability of x using just the m-th summand.

INPUT:

- x – float
- m – integer

OUTPUT:

- float

EXAMPLES:

```
sage: P = hmm.GaussianMixtureDistribution([(.2,-10,.5),(.6,1,1),(.2,20,.5)])
sage: P.prob_m(.5, 0)
2.7608117680508...e-97
sage: P.prob_m(.5, 1)
0.21123919605857971
sage: P.prob_m(.5, 2)
0.0
```

sample (*n=None*)

Return a single sample from this distribution (by default), or if n>1, return a TimeSeries of samples.

INPUT:

- n – integer or None (default: None)

OUTPUT:

- float if n is None (default); otherwise a TimeSeries

EXAMPLES:

```
sage: P = hmm.GaussianMixtureDistribution([(.2,-10,.5),(.6,1,1),(.2,20,.5)])
sage: P.sample()
19.65824361087513
sage: P.sample(1)
[-10.4683]
sage: P.sample(5)
[-0.1688, -10.3479, 1.6812, 20.1083, -9.9801]
sage: P.sample(0)
[]
sage: P.sample(-3)
Traceback (most recent call last):
...
ValueError: n must be nonnegative
```

unfix (*i=None*)

Set that this GaussianMixtureDistribution (or its ith component) is not fixed when using Baum-Welch to update the corresponding HMM.

INPUT:

- i – None (default) or integer; if given, only fix the i-th component

EXAMPLES:

```
sage: P = hmm.GaussianMixtureDistribution([(.2,-10,.5),(.6,1,1),(.2,20,.5)])
sage: P.fix(1); P.is_fixed(1)
True
sage: P.unfix(1); P.is_fixed(1)
False
sage: P.fix(); P.is_fixed()
True
sage: P.unfix(); P.is_fixed()
False
```

`sage.stats.hmm.distributions.`**`unpickle_gaussian_mixture_distribution_v1`**(*c0*, *c1*, *param*, *fixed*)

Used in unpickling GaussianMixtureDistribution's.

EXAMPLES:

```
sage: P = hmm.GaussianMixtureDistribution([(.2,-10,.5),(.6,1,1),(.2,20,.5)])
sage: loads(dumps(P)) == P          # indirect doctest
True
```

CHAPTER

SIX

HIDDEN MARKOV MODELS – UTILITY FUNCTIONS

AUTHOR:

- William Stein, 2010-03

class sage.stats.hmm.util.**HMM_Util**

Bases: object

A class used in order to share cdef's methods between different files.

initial_probs_to_TimeSeries(*pi*, *normalize*)

This function is used internally by the __init__ methods of various Hidden Markov Models.

INPUT:

- pi – vector, list, or TimeSeries
- normalize – if True, replace negative entries by 0 and rescale to ensure that the sum of the entries in each row is equal to 1. If the sum of the entries in a row is 0, replace them all by 1/N.

OUTPUT:

- a TimeSeries of length N

EXAMPLES:

```
sage: import sage.stats.hmm.util
sage: u = sage.stats.hmm.util.HMM_Util()
sage: u.initial_probs_to_TimeSeries([0.1,0.2,0.9], True)
[0.0833, 0.1667, 0.7500]
sage: u.initial_probs_to_TimeSeries([0.1,0.2,0.9], False)
[0.1000, 0.2000, 0.9000]
```

normalize_probability_TimeSeries(*T*, *i*, *j*)

This function is used internally by the Hidden Markov Models code.

Replace entries of T[i:j] in place so that they are all nonnegative and sum to 1. Negative entries are replaced by 0 and T[i:j] is then rescaled to ensure that the sum of the entries in each row is equal to 1. If all entries are 0, replace them by 1/(j-i).

INPUT:

- T – a TimeSeries
- i – nonnegative integer
- j – nonnegative integer

OUTPUT:

- T is modified

EXAMPLES:

```
sage: import sage.stats.hmm.util
sage: T = stats.TimeSeries([.1, .3, .7, .5])
sage: u = sage.stats.hmm.util.HMM_Util()
sage: u.normalize_probability_TimeSeries(T,0,3)
sage: T
[0.0909, 0.2727, 0.6364, 0.5000]
sage: u.normalize_probability_TimeSeries(T,0,4)
sage: T
[0.0606, 0.1818, 0.4242, 0.3333]
sage: abs(T.sum()-1) < 1e-8    # might not exactly equal 1 due to rounding
True
```

state_matrix_to_TimeSeries (*A*, *N*, *normalize*)

This function is used internally by the __init__ methods of Hidden Markov Models to make a transition matrix from A.

INPUT:

- A – matrix, list, list of lists, or TimeSeries
- N – number of states
- normalize – if True, replace negative entries by 0 and rescale to ensure that the sum of the entries in each row is equal to 1. If the sum of the entries in a row is 0, replace them all by 1/N.

OUTPUT:

- a TimeSeries

EXAMPLES:

```
sage: import sage.stats.hmm.util
sage: u = sage.stats.hmm.util.HMM_Util()
sage: u.state_matrix_to_TimeSeries([[.1,.7],[3/7,4/7]], 2, True)
[0.1250, 0.8750, 0.4286, 0.5714]
sage: u.state_matrix_to_TimeSeries([[.1,.7],[3/7,4/7]], 2, False)
[0.1000, 0.7000, 0.4286, 0.5714]
```

CHAPTER SEVEN

DISCRETE GAUSSIAN SAMPLERS OVER THE INTEGERS

This class realizes oracles which returns integers proportionally to $\exp(-(x-c)^2/(2\sigma^2))$. All oracles are implemented using rejection sampling. See *DiscreteGaussianDistributionIntegerSampler.__init__()* for which algorithms are available.

AUTHORS:

- Martin Albrecht (2014-06-28): initial version

EXAMPLES:

We construct a sampler for the distribution $D_{3,c}$ with width $\sigma = 3$ and center $c = 0$:

```
sage: from sage.stats.distributions.discrete_gaussian_integer import␣
↪DiscreteGaussianDistributionIntegerSampler
sage: sigma = 3.0
sage: D = DiscreteGaussianDistributionIntegerSampler(sigma=sigma)
```

We ask for 100000 samples:

```
sage: from six.moves import range
sage: n=100000; l = [D() for _ in range(n)]
```

These are sampled with a probability proportional to $\exp(-x^2/18)$. More precisely we have to normalise by dividing by the overall probability over all integers. We use the fact that hitting anything more than 6 standard deviations away is very unlikely and compute:

```
sage: bound = (6*sigma).floor()
sage: norm_factor = sum([exp(-x^2/(2*sigma^2)) for x in range(-bound,bound+1)])
sage: norm_factor
7.519...
```

With this normalisation factor, we can now test if our samples follow the expected distribution:

```
sage: x=0; l.count(x), ZZ(round(n*exp(-x^2/(2*sigma^2))/norm_factor))
(13355, 13298)
sage: x=4; l.count(x), ZZ(round(n*exp(-x^2/(2*sigma^2))/norm_factor))
(5479, 5467)
sage: x=-10; l.count(x), ZZ(round(n*exp(-x^2/(2*sigma^2))/norm_factor))
(53, 51)
```

We construct an instance with a larger width:

```
sage: from sage.stats.distributions.discrete_gaussian_integer import␣
↪DiscreteGaussianDistributionIntegerSampler
sage: sigma = 127
sage: D = DiscreteGaussianDistributionIntegerSampler(sigma=sigma, algorithm=
↪'uniform+online')
```

ask for 100000 samples:

```
sage: from six.moves import range
sage: n=100000; l = [D() for _ in range(n)] # long time
```

and check if the proportions fit:

```
sage: x=0;   y=1; float(l.count(x))/l.count(y), exp(-x^2/(2*sigma^2))/exp(-y^2/
↪(2*sigma^2)).n() # long time
(1.0, 1.00...)
sage: x=0; y=-100; float(l.count(x))/l.count(y), exp(-x^2/(2*sigma^2))/exp(-y^2/
↪(2*sigma^2)).n() # long time
(1.32..., 1.36...)
```

We construct a sampler with $c\%1! = 0$:

```
sage: from sage.stats.distributions.discrete_gaussian_integer import␣
↪DiscreteGaussianDistributionIntegerSampler
sage: sigma = 3
sage: D = DiscreteGaussianDistributionIntegerSampler(sigma=sigma, c=1/2)
sage: from six.moves import range
sage: n=100000; l = [D() for _ in range(n)] # long time
sage: mean(l).n() # long time
0.486650000000000
```

REFERENCES:

- [DDLL2013]

class sage.stats.distributions.discrete_gaussian_integer.**DiscreteGaussianDistributionInteger**

Bases: `sage.structure.sage_object.SageObject`

A Discrete Gaussian Sampler using rejection sampling.

__init__(*sigma*, *c=0*, *tau=6*, *algorithm=None*, *precision='mp'*)

Construct a new sampler for a discrete Gaussian distribution.

INPUT:

- `sigma` - samples x are accepted with probability proportional to $\exp(-(x-c)/(2\sigma))$
- `c` - the mean of the distribution. The value of `c` does not have to be an integer. However, some algorithms only support integer-valued `c` (default: `0`)
- `tau` - samples outside the range $(\lfloor c\rceil - \lceil \sigma\tau\rceil, ..., \lfloor c\rceil + \lceil \sigma\tau\rceil)$ are considered to have probability zero. This bound applies to algorithms which sample from the uniform distribution (default: `6`)
- **`algorithm` - see list below (default: `"uniform+table"` for σt bounded by** `DiscreteGaussianDistributionIntegerSampler.table_cutoff` and `"uniform+online"` for bigger $\sigma\tau$)
- `precision` - either `"mp"` for multi-precision where the actual precision used is taken from sigma or `"dp"` for double precision. In the latter case results are not reproducible. (default: `"mp"`)

ALGORITHMS:

- "uniform+table" - classical rejection sampling, sampling from the uniform distribution and accepted with probability proportional to $\exp(-(x-c)/(2\sigma))$ where $\exp(-(x-c)/(2\sigma))$ is precomputed and stored in a table. Any real-valued c is supported.
- "uniform+logtable" - samples are drawn from a uniform distribution and accepted with probability proportional to $\exp(-(x-c)/(2\sigma))$ where $\exp(-(x-c)/(2\sigma))$ is computed using logarithmically many calls to Bernoulli distributions. See [DDLL2013] for details. Only integer-valued c are supported.
- "uniform+online" - samples are drawn from a uniform distribution and accepted with probability proportional to $\exp(-(x-c)/(2\sigma))$ where $\exp(-(x-c)/(2\sigma))$ is computed in each invocation. Typically this is very slow. See [DDLL2013] for details. Any real-valued c is accepted.
- "sigma2+logtable" - samples are drawn from an easily samplable distribution with $\sigma = k \cdot \sigma_2$ with $\sigma_2 = \sqrt{1/(2 \log 2)}$ and accepted with probability proportional to $\exp(-(x-c)/(2\sigma))$ where $\exp(-(x-c)/(2\sigma))$ is computed using logarithmically many calls to Bernoulli distributions (but no calls to exp). See [DDLL2013] for details. Note that this sampler adjusts σ to match $k \cdot \sigma_2$ for some integer k. Only integer-valued c are supported.

EXAMPLES:

```
sage: from sage.stats.distributions.discrete_gaussian_integer import DiscreteGaussianDistributionIntegerSampler
sage: DiscreteGaussianDistributionIntegerSampler(3.0, algorithm="uniform+online")
Discrete Gaussian sampler over the Integers with sigma = 3.000000 and c = 0
sage: DiscreteGaussianDistributionIntegerSampler(3.0, algorithm="uniform+table")
Discrete Gaussian sampler over the Integers with sigma = 3.000000 and c = 0
sage: DiscreteGaussianDistributionIntegerSampler(3.0, algorithm="uniform+logtable")
Discrete Gaussian sampler over the Integers with sigma = 3.000000 and c = 0
```

Note that "sigma2+logtable" adjusts σ:

```
sage: DiscreteGaussianDistributionIntegerSampler(3.0, algorithm="sigma2+logtable")
Discrete Gaussian sampler over the Integers with sigma = 3.397287 and c = 0
```

__call__()

Return a new sample.

EXAMPLES:

```
sage: from sage.stats.distributions.discrete_gaussian_integer import DiscreteGaussianDistributionIntegerSampler
sage: DiscreteGaussianDistributionIntegerSampler(3.0, algorithm="uniform+online")()
-3
sage: DiscreteGaussianDistributionIntegerSampler(3.0, algorithm="uniform+table")()
3
```

algorithm

c

sigma

tau

CHAPTER

EIGHT

DISCRETE GAUSSIAN SAMPLERS FOR $\mathbf{Z}[X]$

This class realizes oracles which returns polynomials in $\mathbf{Z}[x]$ where each coefficient is sampled independently with a probability proportional to $\exp(-(x-c)/(2\sigma))$.

AUTHORS:

- Martin Albrecht, Robert Fitzpatrick, Daniel Cabracas, Florian Göpfert, Michael Schneider: initial version

EXAMPLES:

```
sage: from sage.stats.distributions.discrete_gaussian_polynomial import␣
↪DiscreteGaussianDistributionPolynomialSampler
sage: sigma = 3.0; n=1000
sage: l = [DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], 64, sigma)() for _␣
↪in range(n)]
sage: l = [vector(f).norm().n() for f in l]
sage: mean(l), sqrt(64)*sigma
(23.83..., 24.0...)
```

class sage.stats.distributions.discrete_gaussian_polynomial.**DiscreteGaussianDistributionPol**

Bases: sage.structure.sage_object.SageObject

Discrete Gaussian sampler for polynomials.

EXAMPLES:

```
sage: from sage.stats.distributions.discrete_gaussian_polynomial import␣
↪DiscreteGaussianDistributionPolynomialSampler
sage: DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], 8, 3.0)()
3*x^7 + 3*x^6 - 3*x^5 - x^4 - 5*x^2 + 3
sage: gs = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], 8, 3.0)
sage: [gs() for _ in range(3)]
[4*x^7 + 4*x^6 - 4*x^5 + 2*x^4 + x^3 - 4*x + 7, -5*x^6 + 4*x^5 - 3*x^3 + 4*x^2 +␣
↪x, 2*x^7 + 2*x^6 + 2*x^5 - x^4 - 2*x^2 + 3*x + 1]
```

__init__(*P*, *n*, *sigma*)

Construct a sampler for univariate polynomials of degree n-1 where coefficients are drawn independently with standard deviation sigma.

INPUT:

- P - a univariate polynomial ring over the Integers
- n - number of coefficients to be sampled

- `sigma` - coefficients x are accepted with probability proportional to $\exp(-x/(2\sigma))$. If an object of type *sage.stats.distributions.discrete_gaussian_integer.DiscreteGaussianDistributionIntegerSampler* is passed, then this sampler is used to sample coefficients.

EXAMPLES:

```
sage: from sage.stats.distributions.discrete_gaussian_polynomial import␣
↪DiscreteGaussianDistributionPolynomialSampler
sage: DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], 8, 3.0)()
3*x^7 + 3*x^6 - 3*x^5 - x^4 - 5*x^2 + 3
sage: gs = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], 8, 3.0)
sage: [gs() for _ in range(3)]
[4*x^7 + 4*x^6 - 4*x^5 + 2*x^4 + x^3 - 4*x + 7, -5*x^6 + 4*x^5 - 3*x^3 + 4*x^
↪2 + x, 2*x^7 + 2*x^6 + 2*x^5 - x^4 - 2*x^2 + 3*x + 1]
```

__call__()

Return a new sample.

EXAMPLES:

```
sage: from sage.stats.distributions.discrete_gaussian_polynomial import␣
↪DiscreteGaussianDistributionPolynomialSampler
sage: sampler = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], 8, 12.
↪0)
sage: sampler()
8*x^7 - 11*x^5 - 19*x^4 + 6*x^3 - 34*x^2 - 21*x + 9
```

CHAPTER NINE

DISCRETE GAUSSIAN SAMPLERS OVER LATTICES

This file implements oracles which return samples from a lattice following a discrete Gaussian distribution. That is, if σ is big enough relative to the provided basis, then vectors are returned with a probability proportional to $\exp(-|x - c|_2^2/(2\sigma^2))$. More precisely lattice vectors in $x \in \Lambda$ are returned with probability:

$$\exp(-|x - c|_2^2/(2\sigma))/(\sum_{x\in\Lambda} \exp(-|x|_2^2/(2\sigma)))$$

AUTHORS:

- Martin Albrecht (2014-06-28): initial version

EXAMPLES:

```
sage: from sage.stats.distributions.discrete_gaussian_lattice import␣
↪DiscreteGaussianDistributionLatticeSampler
sage: D = DiscreteGaussianDistributionLatticeSampler(ZZ^10, 3.0)
sage: D(), D(), D()
((3, 0, -5, 0, -1, -3, 3, 3, -7, 2), (4, 0, 1, -2, -4, -4, 4, 0, 1, -4), (-3, 0, 4, 5,
↪ 0, 1, 3, 2, 0, -1))
```

class sage.stats.distributions.discrete_gaussian_lattice.**DiscreteGaussianDistributionLatti**

Bases: sage.structure.sage_object.SageObject

GPV sampler for Discrete Gaussians over Lattices.

EXAMPLES:

```
sage: from sage.stats.distributions.discrete_gaussian_lattice import␣
↪DiscreteGaussianDistributionLatticeSampler
sage: D = DiscreteGaussianDistributionLatticeSampler(ZZ^10, 3.0); D
Discrete Gaussian sampler with σ = 3.000000, c=(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)␣
↪over lattice with basis

[1 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 1 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 1]
```

We plot a histogram:

```
sage: from sage.stats.distributions.discrete_gaussian_lattice import␣
↪DiscreteGaussianDistributionLatticeSampler
sage: import warnings
sage: warnings.simplefilter('ignore', UserWarning)
sage: D = DiscreteGaussianDistributionLatticeSampler(identity_matrix(2), 3.0)
sage: S = [D() for _ in range(2^12)]
sage: l = [vector(v.list() + [S.count(v)]) for v in set(S)]
sage: list_plot3d(l, point_list=True, interpolation='nn')
Graphics3d Object
```

REFERENCES:

- [GPV2008]

__init__(*B, sigma=1, c=None, precision=None*)

Construct a discrete Gaussian sampler over the lattice $\Lambda(B)$ with parameter `sigma` and center c.

INPUT:

- `B` – a basis for the lattice, one of the following:
  - an integer matrix,
  - an object with a `matrix()` method, e.g. `ZZ^n`, or
  - an object where `matrix(B)` succeeds, e.g. a list of vectors.
- `sigma` – Gaussian parameter $\sigma > 0$.
- `c` – center c, any vector in $\mathbf{Z}^n$ is supported, but $c \in \Lambda(B)$ is faster.
- `precision` – bit precision ≥ 53.

EXAMPLES:

```
sage: from sage.stats.distributions.discrete_gaussian_lattice import␣
↪DiscreteGaussianDistributionLatticeSampler
sage: n = 2; sigma = 3.0; m = 5000
sage: D = DiscreteGaussianDistributionLatticeSampler(ZZ^n, sigma)
sage: f = D.f
sage: c = D._normalisation_factor_zz(); c
56.2162803067524

sage: l = [D() for _ in range(m)]
sage: v = vector(ZZ, n, (-3,-3))
sage: l.count(v), ZZ(round(m*f(v)/c))
(39, 33)

sage: target = vector(ZZ, n, (0,0))
sage: l.count(target), ZZ(round(m*f(target)/c))
(116, 89)

sage: from sage.stats.distributions.discrete_gaussian_lattice import␣
↪DiscreteGaussianDistributionLatticeSampler
sage: qf = QuadraticForm(matrix(3, [2, 1, 1,  1, 2, 1,  1, 1, 2]))
```

(continues on next page)

(continued from previous page)

```
sage: D = DiscreteGaussianDistributionLatticeSampler(qf, 3.0); D
Discrete Gaussian sampler with σ = 3.000000, c=(0, 0, 0) over lattice with
↪basis

[2 1 1]
[1 2 1]
[1 1 2]
sage: D()
(0, 1, -1)
```

__call__()

Return a new sample.

EXAMPLES:

```
sage: from sage.stats.distributions.discrete_gaussian_lattice import
↪DiscreteGaussianDistributionLatticeSampler
sage: D = DiscreteGaussianDistributionLatticeSampler(ZZ^3, 3.0, c=(1,0,0))
sage: L = [D() for _ in range(2^12)]
sage: abs(mean(L).n() - D.c)
0.08303258...

sage: D = DiscreteGaussianDistributionLatticeSampler(ZZ^3, 3.0, c=(1/2,0,0))
sage: L = [D() for _ in range(2^12)] # long time
sage: mean(L).n() - D.c # long time
(0.0607910156250000, -0.128417968750000, 0.0239257812500000)
```

c

Center c.

Samples from this sampler will be centered at c.

EXAMPLES:

```
sage: from sage.stats.distributions.discrete_gaussian_lattice import
↪DiscreteGaussianDistributionLatticeSampler
sage: D = DiscreteGaussianDistributionLatticeSampler(ZZ^3, 3.0, c=(1,0,0)); D
Discrete Gaussian sampler with σ = 3.000000, c=(1, 0, 0) over lattice with
↪basis

[1 0 0]
[0 1 0]
[0 0 1]

sage: D.c
(1, 0, 0)
```

static compute_precision(*precision*, *sigma*)

Compute precision to use.

INPUT:

- `precision` - an integer > 53 nor `None`.
- `sigma` - if `precision` is `None` then the precision of `sigma` is used.

EXAMPLES:

```
sage: from sage.stats.distributions.discrete_gaussian_lattice import␣
↪DiscreteGaussianDistributionLatticeSampler
sage: DiscreteGaussianDistributionLatticeSampler.compute_precision(100, RR(3))
100
sage: DiscreteGaussianDistributionLatticeSampler.compute_precision(100,␣
↪RealField(200)(3))
100
sage: DiscreteGaussianDistributionLatticeSampler.compute_precision(100, 3)
100
sage: DiscreteGaussianDistributionLatticeSampler.compute_precision(None,␣
↪RR(3))
53
sage: DiscreteGaussianDistributionLatticeSampler.compute_precision(None,␣
↪RealField(200)(3))
200
sage: DiscreteGaussianDistributionLatticeSampler.compute_precision(None, 3)
53
```

sigma

Gaussian parameter σ.

Samples from this sampler will have expected norm $\sqrt{n}\sigma$ where n is the dimension of the lattice.

EXAMPLES:

```
sage: from sage.stats.distributions.discrete_gaussian_lattice import␣
↪DiscreteGaussianDistributionLatticeSampler
sage: D = DiscreteGaussianDistributionLatticeSampler(ZZ^3, 3.0, c=(1,0,0))
sage: D.sigma
3.00000000000000
```

CHAPTER

TEN

T-TEST USING R

`sage.stats.r.ttest(x, y, conf_level=0.95, **kw)`

T-Test using R

Arguments:

- x, y – vectors of same length
- conf_level – confidence level of the interval, [0,1) in percent

Result:

Tuple: (p-value, R return object)

Example:

```
sage: a, b = ttest([1,2,3,4,5],[1,2,3,3.5,5.121]); a
0.941026372027427
```

CHAPTER

ELEVEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

S

T

U

V