pomegranate Documentation

Release 0.6.0

Jacob Schreiber

November 08, 2016

Contents

pomegranate is a python package which implements fast, efficient, and extremely flexible probabilistic models ranging from probability distributions to Bayesian networks to mixtures of hidden Markov models. The most basic level of probabilistic modeling is the a simple probability distribution. If we're modeling language, this may be a simple distribution over the frequency of all possible words a person can say.

1. *Probability Distributions*

The next level up are probabilistic models which use the simple distributions in more complex ways. A markov chain can extend a simple probability distribution to say that the probability of a certain word depends on the word(s) which have been said previously. A hidden Markov model may say that the probability of a certain words depends on the latent/hidden state of the previous word, such as a noun usually follows an adjective.

2. *Markov Chains*
3. *Naive Bayes Classifiers*
4. *General Mixture Models*
5. *Hidden Markov Models*
6. *Bayesian Networks*
7. *Factor Graphs*

The third level are stacks of probabilistic models which can model even more complex phenomena. If a single hidden Markov model can capture a dialect of a language (such as a certain persons speech usage) then a mixture of hidden Markov models may fine tune this to be situation specific. For example, a person may use more formal language at work and more casual language when speaking with friends. By modeling this as a mixture of HMMs, we represent the persons language as a "mixture" of these dialects.

8. GMM-HMMs
9. Mixtures of Models
10. Bayesian Classifiers of Models

CHAPTER 1

Installation

pomegranate is pip installable using `pip install pomegranate`. You can get the bleeding edge from github using the following:

```
git clone https://github.com/jmschrei/pomegranate
cd pomegranate
python setup.py install
```

On Windows machines you may need to download a C++ compiler. For Python 2 this minimal version of Visual Studio 2008 works well. For Python 3 this version of the Visual Studio build tools has been reported to work.

No good project is done alone, and so I'd like to thank all the previous contributors to YAHMM and all the current contributors to pomegranate as well as the graduate students whom I have pestered with ideas. Contributions are eagerly accepted! If you would like to contribute a feature then fork the master branch and be sure to run the tests before changing any code. Let us know what you want to do on the issue tracker just in case we're already working on an implementation of something similar. Also, please don't forget to add tests for any new functions.

1.1 FAQ

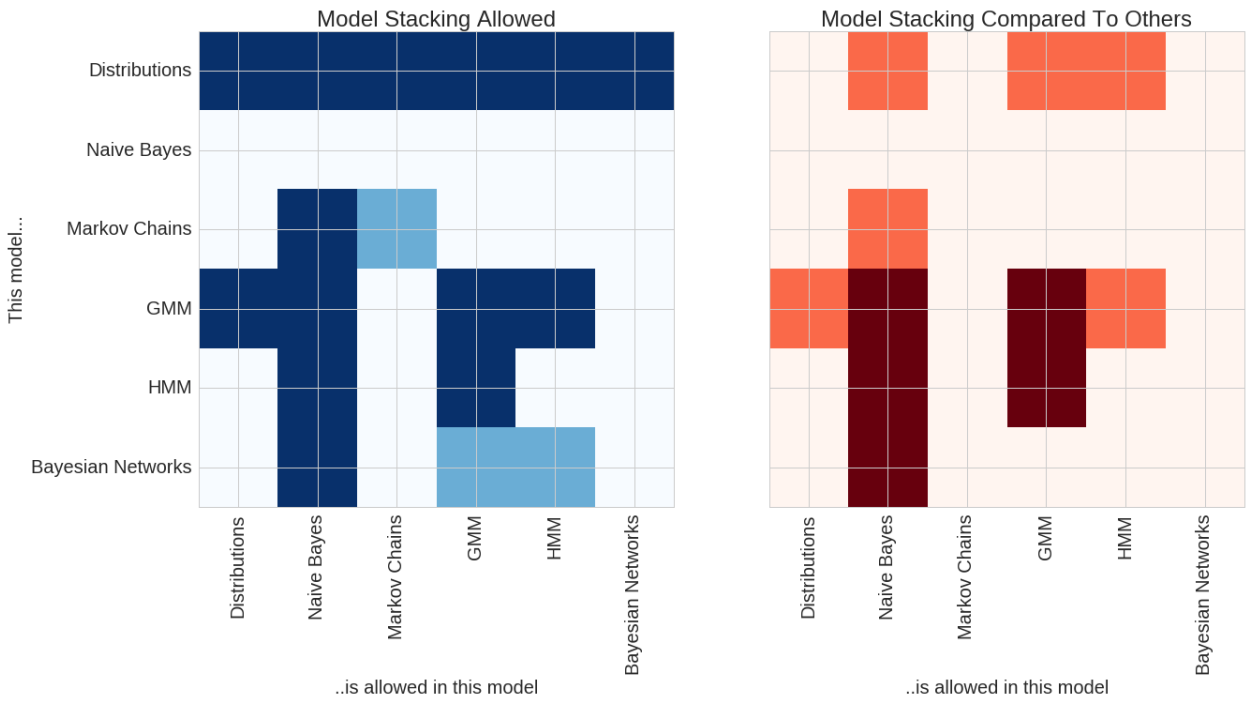
How can I cite pomegranate?

I don't currently have a research paper which can be cited, but the GitHub repository can be.

```
@misc{Schreiber2016,
        author = {Jacob Schreiber},
        title = {pomegranate},
        year = {2016},
        publisher = {GitHub},
        journal = {GitHub repository},
        howpublished = {\url{https://github.com/jmschrei/pomegranate}},
        commit = {enter commit that you used}
}
```

How does pomegranate compare to other packages?

A comparison of the features between pomegranate and others in the python ecosystem can be seen in the following two plots.



The plot on the left shows model stacks which are currently supported by pomegranate. The rows show each model, and the columns show which models those can fit in. Dark blue shows model stacks which currently are supported, and light blue shows model stacks which are currently being worked on and should be available soon. For example, all models use basic distributions as their main component. However, general mixture models (GMMs) can be fit into both Naive Bayes classifiers and hidden Markov models (HMMs). Conversely, HMMs can be fit into GMMs to form mixtures of HMMs. Soon pomegranate will support models like a mixture of Bayesian networks.

The plot on the right shows features compared to other packages in the python ecosystem. Dark red indicates features which no other package supports (to my knowledge!) and orange shows areas where pomegranate has an expanded feature set compared to other packages. For example, both pomegranate and sklearn support Gaussian naive Bayes classifiers. However, pomegranate supports naive Bayes of arbitrary distributions and combinations of distributions, such as one feature being Gaussian, one being log normal, and one being exponential (useful to classify things like ionic current segments or audio segments). pomegranate also extends naive Bayes past its "naivity" to allow for features to be dependent on each other, and allows input to be more complex things like hidden Markov models and Bayesian networks. There's no rule that each of the inputs to naive Bayes has to be the same type though, allowing you to do things like compare a markov chain to a HMM. No other package supports a HMM Naive Bayes! Packages like hmmlearn support the GMM-HMM, but for them GMM strictly means Gaussian mixture model, whereas in pomegranate it ~can~ be a Gaussian mixture model, but it can also be an arbitrary mixture model of any types of distributions. Lastly, no other package supports mixtures of HMMs despite their prominent use in things like audio decoding and biological sequence analysis.

Models can be stacked more than once, though. For example, a "naive" Bayes classifier can be used to compare multiple mixtures of HMMs to each other, or compare a HMM with GMM emissions to one without GMM emissions. You can also create mixtures of HMMs with GMM emissions, and so the most stacking currently supported is a "naive" Bayes classifier of mixtures of HMMs with GMM emissions, or four levels of stacking.

How can pomegranate be faster than numpy?

pomegranate has been shown to be faster than numpy at updating univariate and multivariate gaussians. One of the reasons is t

This allows pomegranate to calculate both mean and covariance in a single pass of the dataset. In addition, one of the reasons that numpy is so fast is its use of BLAS. pomegranate also uses BLAS, but uses the cython level calls to BLAS so that the data doesn't have to pass between cython and python multiple times.

Does pomegranate support parallelization?

Yes! pomegranate supports parallelized model fitting and model predictions, both in a data-parallel manner. Since the backend is written in cython the global interpreter lock (GIL) can be released and multi-threaded training can be supported via joblib. This means that parallelization is utilized time isn't spent piping data from one process to another nor are multiple copies of the model made.

Does pomegranate support GPUs?

Currently pomegranate does not support GPUs.

Does pomegranate support distributed computing?

Currently pomegranate is not set up for a distributed environment, though the pieces are currently there to make this possible.

1.2 Out of Core

Sometimes datasets which we'd like to train on can't fit in memory but we'd still like to get an exact update. pomegranate supports out of core training to allow this, by allowing models to summarize batches of data into sufficient statistics and then later on using these sufficient statistics to get an exact update for model parameters. These are done through the methods `model.summarize` and `model.from_summaries`. Let's see an example of using it to update a normal distribution.

```
>>> from pomegranate import *
>>> import numpy
>>>
>>> a = NormalDistribution(1, 1)
>>> b = NormalDistribution(1, 1)
>>> X = numpy.random.normal(3, 5, size=(5000,))
>>>
>>> a.fit(X)
>>> a
{
    "frozen" :false,
    "class" :"Distribution",
    "parameters" :[
        3.012692830297519,
        4.972082359070984
    ],
    "name" :"NormalDistribution"
}
>>> for i in range(5):
>>>     b.summarize(X[i*1000:(i+1)*1000])
>>> b.from_summaries()
>>> b
{
    "frozen" :false,
    "class" :"Distribution",
    "parameters" :[
        3.01269283029752,
        4.972082359070983
    ],
    "name" :"NormalDistribution"
}
```

This is a simple example with a simple distribution, but all models and model stacks support this type of learning. Lets next look at a simple Bayesian network.

We can see that before fitting to any data, the distribution in one of the states is equal for both. After fitting the first distribution they become different as would be expected. After fitting the second one through summarize the distributions become equal again, showing that it is recovering an exact update.

It's easy to see how one could use this to update models which don't use Expectation Maximization (EM) to train, since it is an iterative algorithm. For algorithms which use EM to train there is a `'fit'` wrapper which will allow you to load up batches of data from a numpy memory map to train on automatically.

1.3 Probability Distributions

IPython Notebook Tutorial

While probability distributions are frequently used as components of more complex models such as mixtures and hidden Markov models, they can also be used by themselves. Many data science tasks require fitting a distribution to data or generating samples under a distribution. pomegranate has a large library of both univariate and multivariate distributions which can be used with an intuitive interface.

Univariate Distributions

`UniformDistribution`	A uniform distribution between two values.
`BernoulliDistribution`	A Bernoulli distribution describing the probability of a binary variable.
`NormalDistribution`	A normal distribution based on a mean and standard deviation.
`LogNormalDistribution`	Represents a lognormal distribution over non-negative floats.
`ExponentialDistribution`	Represents an exponential distribution on non-negative floats.
`PoissonDistribution`	A discrete probability distribution which expresses the probability of a number of events occurin
`BetaDistribution`	This distribution represents a beta distribution, parameterized using alpha/beta, which are both sh
`GammaDistribution`	This distribution represents a gamma distribution, parameterized in the alpha/beta (shape/rate) pa
`DiscreteDistribution`	A discrete distribution, made up of characters and their probabilities, assuming that these probab

Kernel Densities

`GaussianKernelDensity`	A quick way of storing points to represent a Gaussian kernel density in one dimension.
`UniformKernelDensity`	A quick way of storing points to represent an Exponential kernel density in one dimension.
`TriangleKernelDensity`	A quick way of storing points to represent an Exponential kernel density in one dimension.

Multivariate Distributions

`IndependentComponentsDistribution`	Allows you to create a multivariate distribution, where each distribution is indepe
`MultivariateGaussianDistribution`	
`DirichletDistribution`	A Dirichlet distribution, usually a prior for the multinomial distributions.
`ConditionalProbabilityTable`	A conditional probability table, which is dependent on values from at least one pr
`JointProbabilityTable`	A joint probability table.

While there is a large variety of univariate distributions, multivariate distributions can be made from univariate distributions by using `'IndependentComponentsDistribution'` with the assumption that each column of data is independent from the other columns (instead of being related by a covariance matrix, like in multivariate gaussians). Here is an example:

```
>>> d1 = NormalDistribution(5, 2)
>>> d2 = LogNormalDistribution(1, 0.3)
>>> d3 = ExponentialDistribution(4)
>>> d = IndependentComponentsDistribution([d1, d2, d3])
```

1.3.1 Initialization

Initializing a distribution is simple and done just by passing in the distribution parameters. For example, the parameters of a normal distribution are the mean (mu) and the standard deviation (sigma). We can initialize it as follows:

```
>>> from pomegranate import *
>>> a = NormalDistribution(5, 2)
```

However, frequently we don't know the parameters of the distribution beforehand or would like to directly fit this distribution to some data. We can do this through the *from_samples* class method.

```
>>> b = NormalDistribution.from_samples([3, 4, 5, 6, 7], weights=[0.5, 1, 1.5, 1, 0.5])
```

If we want to fit the model to weighted samples, we can just pass in an array of the relative weights of each sample as well.

```
>>> b = NormalDistribution.from_samples([3, 4, 5, 6, 7], weights=[0.5, 1, 1.5, 1, 0.5])
```

1.3.2 Probability

Distributions are typically used to calculate the probability of some sample. This can be done using either the *probability* or *log_probability* methods.

```
>>> a = NormalDistribution(5, 2)
>>> a.log_probability(8)
-2.737085713764219
>>> a.probability(8)
0.064758797832971712
>>> b = NormalDistribution.from_samples([3, 4, 5, 6, 7], weights=[0.5, 1, 1.5, 1, 0.5])
>>> b.log_probability(8)
-4.437779569430167
```

These methods work for univariate distributions, kernel densities, and multivariate distributions all the same. For a multivariate distribution you'll have to pass in an array for the full sample.

```
>>> d1 = NormalDistribution(5, 2)
>>> d2 = LogNormalDistribution(1, 0.3)
>>> d3 = ExponentialDistribution(4)
>>> d = IndependentComponentsDistribution([d1, d2, d3])
>>>
>>> X = [6.2, 0.4, 0.9]
>>> d.log_probability(X)
-23.205411733352875
```

1.3.3 Fitting

We may wish to fit the distribution to new data, either overriding the previous parameters completely or moving the parameters to match the dataset more closely through inertia. Distributions are updated using maximum likelihood estimates (MLE). Kernel densities will either discard previous points or downweight them if inertia is used.

```
>>> d = NormalDistribution(5, 2)
>>> d.fit([1, 5, 7, 3, 2, 4, 3, 5, 7, 8, 2, 4, 6, 7, 2, 4, 5, 1, 3, 2, 1])
>>> d
{
    "frozen" :false,
    "class" :"Distribution",
```

```
    "parameters" :[
        3.9047619047619047,
        2.13596776114341
    ],
    "name" :"NormalDistribution"
}
```

Training can be done on weighted samples by passing an array of weights in along with the data for any of the training functions, like the following:

```
>>> d = NormalDistribution(5, 2)
>>> d.fit([1, 5, 7, 3, 2, 4], weights=[0.5, 0.75, 1, 1.25, 1.8, 0.33])
>>> d
{
    "frozen" :false,
    "class" :"Distribution",
    "parameters" :[
        3.538188277087034,
        1.954149818564894
    ],
    "name" :"NormalDistribution"
}
```

Training can also be done with inertia, where the new value will be some percentage the old value and some percentage the new value, used like *d.from_sample([5,7,8], inertia=0.5)* to indicate a 50-50 split between old and new values.

1.3.4 API Reference

class pomegranate.distributions.**Distribution**

A probability distribution.

Represents a probability distribution over the defined support. This is the base class which must be subclassed to specific probability distributions. All distributions have the below methods exposed.

Parameters **Varies on distribution.**

Attributes

name	(str) The name of the type of distributioon.
summaries	(list) Sufficient statistics to store the update.
frozen	(bool) Whether or not the distribution will be updated during training.
d	(int) The dimensionality of the data. Univariate distributions are all 1, while multivariate distributions are > 1.

clear_summaries()

Clear the summary statistics stored in the object. Parameters ———- None Returns ——- None

copy()

Return a deep copy of this distribution object.

This object will not be tied to any other distribution or connected in any form.

Returns **distribution** : Distribution

A copy of the distribution with the same parameters.

from_json()
Read in a serialized distribution and return the appropriate object.

Parameters **s** : str

A JSON formatted string containing the file.

Returns **model** : object

A properly initialized and baked model.

from_summaries()
Fit the distribution to the stored sufficient statistics. Parameters ———- inertia : double, optional

The weight of the previous parameters of the model. The new parameters will roughly be old_param*inertia + new_param*(1-inertia), so an inertia of 0 means ignore the old parameters, whereas an inertia of 1 means ignore the new parameters. Default is 0.0.

None

log_probability()
Return the log probability of the given symbol under this distribution.

Parameters **symbol** : double

The symbol to calculate the log probability of (overriden for DiscreteDistributions)

Returns **logp** : double

The log probability of that point under the distribution.

marginal()
Return the marginal of the distribution.

Parameters ***args** : optional

Arguments to pass in to specific distributions

****kwargs** : optional

Keyword arguments to pass in to specific distributions

Returns **distribution** : Distribution

The marginal distribution. If this is a multivariate distribution then this method is filled in. Otherwise returns self.

plot()
Plot the distribution by sampling from it.

This function will plot a histogram of samples drawn from a distribution on the current open figure.

Parameters **n** : int, optional

The number of samples to draw from the distribution. Default is 1000.

****kwargs** : arguments, optional

Arguments to pass to matplotlib's histogram function.

Returns None

summarize()
Summarize a batch of data into sufficient statistics for a later update. Parameters ———- items : array-like, shape (n_samples, n_dimensions)

This is the data to train on. Each row is a sample, and each column is a dimension to train on. For univariate distributions an array is used, while for multivariate distributions a 2d matrix is used.

weights [array-like, shape (n_samples,), optional] The initial weights of each sample in the matrix. If nothing is passed in then each sample is assumed to be the same weight. Default is None.

None

to_json()
Serialize the distribution to a JSON.

Parameters **separators** : tuple, optional

The two separaters to pass to the json.dumps function for formatting. Default is (',', ' : ').

indent : int, optional

The indentation to use at each level. Passed to json.dumps for formatting. Default is 4.

Returns **json** : str

A properly formatted JSON object.

1.4 General Mixture Models

IPython Notebook Tutorial

General Mixture Models (GMMs) are an unsupervised model composed of multiple distributions (commonly also referred to as components) and corresponding weights. This allows you to model more sophisticated phenomena probabilistically. A common task is to figure out which component a new data point comes from given only a large quantity of unlabelled data.

1.4.1 Initialization

General Mixture Models can be initialized in two ways depending on if you know the initial parameters of the distributions of not. If you do know the prior parameters of the distributions then you can pass them in as a list. These do not have to be the same distribution–you can mix and match distributions as you want. You can also pass in the weights, or the prior probability of a sample belonging to that component of the model.

```
>>> from pomegranate import *
>>> gmm = GeneralMixtureModel([NormalDistribution(5, 2), NormalDistribution(1, 2)], weights=[0.33, 0.
```

If you do not know the initial parameters, then the components can be initialized using kmeans to find initial clusters. Initial parameters for the models are then extracted from the clusters and EM is used to fine tune the model.

```
>>> from pomegranate import *
>>> gmm = GeneralMixtureModel( NormalDistribution, n_components=2 )
```

This allows any distribution in pomegranate to be natively used in GMMs.

1.4.2 Log Probability

The probability of a point is the sum of its probability under each of the components, multiplied by the weight of each component c, $P(D|M) = \sum_{c \in M} P(D|c)$. This is easily calculated by summing the probability under each distribution in the mixture model and multiplying by the appropriate weights, and then taking the log.

1.4.3 Prediction

The common prediction tasks involve predicting which component a new point falls under. This is done using Bayes rule $P(M|D) = \frac{P(D|M)P(M)}{P(D)}$ to determine the posterior probability $P(M|D)$ as opposed to simply the likelihood $P(D|M)$. Bayes rule indicates that it isn't simply the likelihood function which makes this prediction but the likelihood function multiplied by the probability that that distribution generated the sample. For example, if you have a distribution which has 100x as many samples fall under it, you would naively think that there is a ~99% chance that any random point would be drawn from it. Your belief would then be updated based on how well the point fit each distribution, but the proportion of points generated by each sample is important as well.

We can get the component label assignments using `model.predict(data)`, which will return an array of indexes corresponding to the maximally likely component. If what we want is the full matrix of $P(M|D)$, then we can use `model.predict_proba(data)`, which will return a matrix with each row being a sample, each column being a component, and each cell being the probability that that model generated that data. If we want log probabilities instead we can use `model.predict_log_proba(data)` instead.

1.4.4 Fitting

Training GMMs faces the classic chicken-and-egg problem that most unsupervised learning algorithms face. If we knew which component a sample belonged to, we could use MLE estimates to update the component. And if we knew the parameters of the components we could predict which sample belonged to which component. This problem is solved using expectation-maximization, which iterates between the two until convergence. In essence, an initialization point is chosen which usually is not a very good start, but through successive iteration steps, the parameters converge to a good ending.

These models are fit using `model.fit(data)`. A maximimum number of iterations can be specified as well as a stopping threshold for the improvement ratio. See the API reference for full documentation.

1.4.5 API Reference

class `pomegranate.gmm.`**`GeneralMixtureModel`**

A General Mixture Model.

This mixture model can be a mixture of any distribution as long as they are all of the same dimensionality. Any object can serve as a distribution as long as it has fit(X, weights), log_probability(X), and summarize(X, weights)/from_summaries() methods if out of core training is desired.

Parameters **distributions** : array-like, shape (n_components,) or callable

The components of the model. If array, corresponds to the initial distributions of the components. If callable, must also pass in the number of components and kmeans++ will be used to initialize them.

weights : array-like, optional, shape (n_components,)

The prior probabilities corresponding to each component. Does not need to sum to one, but will be normalized to sum to one internally. Defaults to None.

n_components : int, optional

If a callable is passed into distributions then this is the number of components to initialize using the kmeans++ algorithm. Defaults to None.

Examples

```
>>> from pomegranate import *
>>> clf = GeneralMixtureModel([
>>>     NormalDistribution(5, 2),
>>>     NormalDistribution(1, 1)])
>>> clf.log_probability(5)
-2.304562194038089
>>> clf.predict_proba([[5], [7], [1]])
array([[ 0.99932952,  0.00067048],
       [ 0.99999995,  0.00000005],
       [ 0.06337894,  0.93662106]])
>>> clf.fit([[1], [5], [7], [8], [2]])
>>> clf.predict_proba([[5], [7], [1]])
array([[ 1.        ,  0.        ],
       [ 1.        ,  0.        ],
       [ 0.00004383,  0.99995617]])
>>> clf.distributions
array([ {
    "frozen" :false,
    "class" :"Distribution",
    "parameters" :[
        6.6571359101390755,
        1.2639830514274502
    ],
    "name" :"NormalDistribution"
},
       {
    "frozen" :false,
    "class" :"Distribution",
    "parameters" :[
        1.498707696758334,
        0.4999983303277837
    ],
    "name" :"NormalDistribution"
}], dtype=object)
```

Attributes

distributions	(array-like, shape (n_components,)) The component distribution objects.
weights	(array-like, shape (n_components,)) The learned prior weight of each object

copy()

Return a deep copy of this distribution object.

This object will not be tied to any other distribution or connected in any form.

Parameters None

Returns **distribution** : Distribution

A copy of the distribution with the same parameters.

fit()

Fit the model to new data using EM.

This method fits the components of the model to new data using the EM method. It will iterate until either max iterations has been reached, or the stop threshold has been passed.

This is a sklearn wrapper for train method.

Parameters **X** : array-like, shape (n_samples, n_dimensions)

This is the data to train on. Each row is a sample, and each column is a dimension to train on.

weights : array-like, shape (n_samples,), optional

The initial weights of each sample in the matrix. If nothing is passed in then each sample is assumed to be the same weight. Default is None.

inertia : double, optional

The weight of the previous parameters of the model. The new parameters will roughly be old_param*inertia + new_param*(1-inertia), so an inertia of 0 means ignore the old parameters, whereas an inertia of 1 means ignore the new parameters. Default is 0.0.

stop_threshold : double, optional, positive

The threshold at which EM will terminate for the improvement of the model. If the model does not improve its fit of the data by a log probability of 0.1 then terminate. Default is 0.1.

max_iterations : int, optional, positive

The maximum number of iterations to run EM for. If this limit is hit then it will terminate training, regardless of how well the model is improving per iteration. Default is 1e8.

verbose : bool, optional

Whether or not to print out improvement information over iterations. Default is False.

Returns **improvement** : double

The total improvement in log probability P(D|M)

freeze()

Freeze the distribution, preventing updates from occuring.

from_summaries()

Fit the model to the collected sufficient statistics.

Fit the parameters of the model to the sufficient statistics gathered during the summarize calls. This should return an exact update.

Parameters **inertia** : double, optional

The weight of the previous parameters of the model. The new parameters will roughly be old_param*inertia + new_param*(1-inertia), so an inertia of 0 means ignore the old parameters, whereas an inertia of 1 means ignore the new parameters. Default is 0.0.

Returns None

log_probability()

Calculate the log probability of a point under the distribution.

The probability of a point is the sum of the probabilities of each distribution multiplied by the weights. Thus, the log probability is the sum of the log probability plus the log prior.

This is the python interface.

Parameters **X** : numpy.ndarray, shape=(n, d) or (n, m, d)

The samples to calculate the log probability of. Each row is a sample and each column is a dimension. If emissions are HMMs then shape is (n, m, d) where m is variable length for each obervation, and X becomes an array of n (m, d)-shaped arrays.

Returns **log_probability** : double

The log probabiltiy of the point under the distribution.

predict()

Predict the most likely component which generated each sample.

Calculate the posterior P(M|D) for each sample and return the index of the component most likely to fit it. This corresponds to a simple argmax over the responsibility matrix.

This is a sklearn wrapper for the maximum_a_posteriori method.

Parameters **X** : array-like, shape (n_samples, n_dimensions)

The samples to do the prediction on. Each sample is a row and each column corresponds to a dimension in that sample. For univariate distributions, a single array may be passed in.

Returns **y** : array-like, shape (n_samples,)

The predicted component which fits the sample the best.

predict_log_proba()

Calculate the posterior log P(M|D) for data.

Calculate the log probability of each item having been generated from each component in the model. This returns normalized log probabilities such that the probabilities should sum to 1

This is a sklearn wrapper for the original posterior function.

Parameters **X** : array-like, shape (n_samples, n_dimensions)

The samples to do the prediction on. Each sample is a row and each column corresponds to a dimension in that sample. For univariate distributions, a single array may be passed in.

Returns **y** : array-like, shape (n_samples, n_components)

The normalized log probability log P(M|D) for each sample. This is the probability that the sample was generated from each component.

predict_proba()

Calculate the posterior P(M|D) for data.

Calculate the probability of each item having been generated from each component in the model. This returns normalized probabilities such that each row should sum to 1.

Since calculating the log probability is much faster, this is just a wrapper which exponentiates the log probability matrix.

Parameters **X** : array-like, shape (n_samples, n_dimensions)

The samples to do the prediction on. Each sample is a row and each column corresponds to a dimension in that sample. For univariate distributions, a single array may be passed in.

Returns **probability** : array-like, shape (n_samples, n_components)

The normalized probability P(M|D) for each sample. This is the probability that the sample was generated from each component.

probability()
Return the probability of the given symbol under this distribution.

Parameters **symbol** : object

The symbol to calculate the probability of

Returns **probability** : double

The probability of that point under the distribution.

sample()
Generate a sample from the model.

First, randomly select a component weighted by the prior probability, Then, use the sample method from that component to generate a sample.

Parameters **n** : int, optional

The number of samples to generate. Defaults to 1.

Returns **sample** : array-like or object

A randomly generated sample from the model of the type modelled by the emissions. An integer if using most distributions, or an array if using multivariate ones, or a string for most discrete distributions. If n=1 return an object, if n>1 return an array of the samples.

summarize()
Summarize a batch of data and store sufficient statistics.

This will run the expectation step of EM and store sufficient statistics in the appropriate distribution objects. The summarization can be thought of as a chunk of the E step, and the from_summaries method as the M step.

Parameters **X** : array-like, shape (n_samples, n_dimensions)

This is the data to train on. Each row is a sample, and each column is a dimension to train on.

weights : array-like, shape (n_samples,), optional

The initial weights of each sample in the matrix. If nothing is passed in then each sample is assumed to be the same weight. Default is None.

Returns **logp** : double

The log probability of the data given the current model. This is used to speed up EM.

thaw()
Thaw the distribution, re-allowing updates to occur.

1.5 Hidden Markov Models

- IPython Notebook Tutorial
- IPython Notebook Sequence Alignment Tutorial

Hidden Markov models (HMMs) are a form of structured model in which a sequence of observations are labelled according to the hidden state they belong. A strength of HMMs is their ability to analyze variable length sequences whereas other models require a static set of features. This makes them extensively used in the fields of natural language processing and bioinformatics where data is routinely variable length sequences. They can be thought of as a structured form of General Mixture Models.

1.5.1 Initialization

Transitioning from YAHMM to pomegranate is simple because the only change is that the `Model` class is now `HiddenMarkovModel`. You can port your code over by either changing `Model` to `HiddenMarkovModel` or changing your imports at the top of the file as follows:

```
>>> from pomegranate import *
>>> from pomegranate import HiddenMarkovModel as Model
```

instead of

```
>>> from yahmm import *
```

Since hidden Markov models are graphical structures, that structure has to be defined. pomegranate allows you to define this structure either through matrices as is common in other packages, or build it up state by state and edge by edge. pomegranate differs from other packages in that it offers both explicit start and end states which you must begin in or end in. Explicit end states give you more control over the model because algorithms require ending there, as opposed to in any state in the model. It also offers silent states, which are states without explicit emission distributions but can be used to significantly simplify the graphical structure.

```
>>> from pomegranate import *
>>> dists = [NormalDistribution(5, 1), NormalDistribution(1, 7), NormalDistribution(8,2)]
>>> trans_mat = numpy.array([[0.7, 0.3, 0.0],
                             [0.0, 0.8, 0.2],
                             [0.0, 0.0, 0.9]])
>>> starts = numpy.array([1.0, 0.0, 0.0])
>>> ends = numpy.array([0.0, 0.0, 0.1])
>>> model = HiddenMarkovModel.from_matrix(trans_mat, dists, starts, ends)
```

Alternatively this model could be created edge by edge and state by state. This is helpful for large sparse graphs. You must add transitions using the explicit start and end states where the sum of probabilities leaving a state sums to 1.0.

```
>>> from pomegranate import *
>>> s1 = State( Distribution( NormalDistribution(5, 1) ) )
>>> s2 = State( Distribution( NormalDistribution(1, 7) ) )
>>> s3 = State( Distribution( NormalDistribution(8, 2) ) )
>>> model = HiddenMarkovModel()
>>> model.add_states(s1, s2, s3)
>>> model.add_transition(model.start, s1, 1.0)
>>> model.add_transition(s1, s1, 0.7)
>>> model.add_transition(s1, s2, 0.3)
>>> model.add_transition(s2, s2, 0.8)
>>> model.add_transition(s2, s3, 0.2)
>>> model.add_transition(s3, s3, 0.9)
>>> model.add_transition(s3, model.end, 0.1)
>>> model.bake()
```

Models built in this manner must be explicitly "baked" at the end. This finalizes the model topology and creates the internal sparse matrix which makes up the model. This removes "orphan" parts of the model, normalizes all transitions to make sure they sum to 1.0, stores information about tied distributions, edges, and pseudocounts, and merges unneccesary silent states in the model for computational efficiency. This can cause the *bake* step to take a little bit of time. If you want to reduce this overhead and are sure you specified the model correctly you can pass in *merge="None"* to the bake step to avoid model checking.

1.5.2 Log Probability

There are two common forms of the log probability which are used. The first is the log probability of the most likely path the sequence can take through the model, called the Viterbi probability. This can be calculated using `model.viterbi(sequence)`. However, this is $P(D|S_{ML}, S_{ML}, S_{ML})$ not $P(D|M)$. In order to get $P(D|M)$ we have to sum over all possible paths instead of just the single most likely path, which can be calculated using `model.log_probability(sequence)` using the forward or backward algorithms. On that note, the full forward matrix can be returned using `model.forward(sequence)` and the full backward matrix can be returned using `model.backward(sequence)`.

1.5.3 Prediction

A common prediction technique is calculating the Viterbi path, which is the most likely sequence of states that generated the sequence given the full model. This is solved using a simple dynamic programming algorithm similar to sequence alignment in bioinformatics. This can be called using `model.viterbi(sequence)`. A sklearn wrapper can be called using `model.predict(sequence, algorithm='viterbi')`.

Another prediction technique is called maximum a posteriori or forward-backward, which uses the forward and backward algorithms to calculate the most likely state per observation in the sequence given the entire remaining alignment. Much like the forward algorithm can calculate the sum-of-all-paths probability instead of the most likely single path, the forward-backward algorithm calculates the best sum-of-all-paths state assignment instead of calculating the single best path. This can be called using `model.predict(sequence, algorithm='map')` and the raw normalized probability matrices can be called using `model.predict_proba(sequence)`.

1.5.4 Fitting

A simple fitting algorithm for hidden Markov models is called Viterbi training. In this method, each observation is tagged with the most likely state to generate it using the Viterbi algorithm. The distributions (emissions) of each states are then updated using MLE estimates on the observations which were generated from them, and the transition matrix is updated by looking at pairs of adjacent state taggings. This can be done using `model.fit(sequence, algorithm='viterbi')`.

However, this is not the best way to do training and much like the other sections there is a way of doing training using sum-of-all-paths probabilities instead of maximally likely path. This is called Baum-Welch or forward-backward training. Instead of using hard assignments based on the Viterbi path, observations are given weights equal to the probability of them having been generated by that state. Weighted MLE can then be done to updates the distributions, and the soft transition matrix can give a more precise probability estimate. This is the default training algorithm, and can be called using either `model.fit(sequences)` or explicitly using `model.fit(sequences, algorithm='baum-welch')`.

Fitting in pomegranate also has a number of options, including the use of distribution or edge inertia, freezing certain states, tying distributions or edges, and using pseudocounts. See the tutorial linked to at the top of this page for full details on each of these options.

1.5.5 API Reference

class `pomegranate.hmm.`**`HiddenMarkovModel`**

A Hidden Markov Model

A Hidden Markov Model (HMM) is a directed graphical model where nodes are hidden states which contain an observed emission distribution and edges contain the probability of transitioning from one hidden state to another. HMMs allow you to tag each observation in a variable length sequence with the most likely hidden state according to the model.

Parameters **name** : str, optional

The name of the model. Default is None.

start : State, optional

An optional state to force the model to start in. Default is None.

end : State, optional

An optional state to force the model to end in. Default is None.

Examples

```
>>> from pomegranate import *
>>> d1 = DiscreteDistribution({'A' : 0.35, 'C' : 0.20, 'G' : 0.05, 'T' : 40})
>>> d2 = DiscreteDistribution({'A' : 0.25, 'C' : 0.25, 'G' : 0.25, 'T' : 25})
>>> d3 = DiscreteDistribution({'A' : 0.10, 'C' : 0.40, 'G' : 0.40, 'T' : 10})
>>>
>>> s1 = State( d1, name="s1" )
>>> s2 = State( d2, name="s2" )
>>> s3 = State( d3, name="s3" )
>>>
>>> model = HiddenMarkovModel('example')
>>> model.add_states([s1, s2, s3])
>>> model.add_transition( model.start, s1, 0.90 )
>>> model.add_transition( model.start, s2, 0.10 )
>>> model.add_transition( s1, s1, 0.80 )
>>> model.add_transition( s1, s2, 0.20 )
>>> model.add_transition( s2, s2, 0.90 )
>>> model.add_transition( s2, s3, 0.10 )
>>> model.add_transition( s3, s3, 0.70 )
>>> model.add_transition( s3, model.end, 0.30 )
>>> model.bake()
>>>
>>> print model.log_probability(list('ACGACTATTCGAT'))
-4.31828085576
>>> print ", ".join( state.name for i, state in model.viterbi(list('ACGACTATTCGAT'))[1] )
example-start, s1, s2, s2, s2, s2, s2, s2, s2, s2, s2, s2, s2, s3, example-end
```

Attributes

start	(State) A state object corresponding to the initial start of the model
end	(State) A state object corresponding to the forced end of the model
start_index	(int) The index of the start object in the state list
end_index	(int) The index of the end object in the state list
silent_start	(int) The index of the beginning of the silent states in the state list
states	(list) The list of all states in the model, with silent states at the end

add_edge()

Add a transition from state a to state b which indicates that B is dependent on A in ways specified by the distribution.

add_model()

Add the states and edges of another model to this model.

Parameters **other** : HiddenMarkovModel

The other model to add

Returns None

add_node()
Add a node to the graph.

add_nodes()
Add multiple states to the graph.

add_state()
Add a state to the given model.

The state must not already be in the model, nor may it be part of any other model that will eventually be combined with this one.

Parameters **state** : State

A state object to be added to the model.

Returns None

add_states()
Add multiple states to the model at the same time.

Parameters **states** : list or generator

Either a list of states which are entered sequentially, or just comma separated values, for example model.add_states(a, b, c, d).

Returns None

add_transition()
Add a transition from state a to state b.

Add a transition from state a to state b with the given (non-log) probability. Both states must be in the HMM already. self.start and self.end are valid arguments here. Probabilities will be normalized such that every node has edges summing to 1. leaving that node, but only when the model is baked. Psueodocounts are allowed as a way of using edge-specific pseudocounts for training.

By specifying a group as a string, you can tie edges together by giving them the same group. This means that a transition across one edge in the group counts as a transition across all edges in terms of training.

Parameters **a** : State

The state that the edge originates from

b : State

The state that the edge goes to

probability : double

The probability of transitioning from state a to state b in [0, 1]

pseudocount : double, optional

The pseudocount to use for this specific edge if using edge pseudocounts for training. Defaults to the probability. Default is None.

group : str, optional

The name of the group of edges to tie together during training. If groups are used, then a transition across any one edge counts as a transition across all edges. Default is None.

Returns None

add_transitions()

Add many transitions at the same time,

Parameters **a** : State or list

Either a state or a list of states where the edges originate.

b : State or list

Either a state or a list of states where the edges go to.

probabilities : list

The probabilities associated with each transition.

pseudocounts : list, optional

The pseudocounts associated with each transition. Default is None.

groups : list, optional

The groups of each edge. Default is None.

Returns None

Examples

```
>>> model.add_transitions([model.start, s1], [s1, model.end], [1., 1.])
>>> model.add_transitions([model.start, s1, s2, s3], s4, [0.2, 0.4, 0.3, 0.9])
>>> model.add_transitions(model.start, [s1, s2, s3], [0.6, 0.2, 0.05])
```

backward()

Run the backward algorithm on the sequence.

Calculate the probability of each observation being aligned to each state by going backward through a sequence. Returns the full backward matrix. Each index i, j corresponds to the sum-of-all-paths log probability of starting at the end of the sequence, and aligning observations to hidden states in such a manner that observation i was aligned to hidden state j. Uses row normalization to dynamically scale each row to prevent underflow errors.

If the sequence is impossible, will return a matrix of nans.

See also:

- Silent state handling taken from p. 71 of "Biological

Sequence Analysis" by Durbin et al., and works for anything which does not have loops of silent states.

•Row normalization technique explained by

http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf on p. 14.

Parameters **sequence** : array-like

An array (or list) of observations.

Returns **matrix** : array-like, shape (len(sequence), n_states)

The probability of aligning the sequences to states in a backward fashion.

bake()

Finalize the topology of the model.

Finalize the topology of the model and assign a numerical index to every state. This method must be called before any of the probability- calculating methods.

This fills in self.states (a list of all states in order) and self.transition_log_probabilities (log probabilities for transitions), as well as self.start_index and self.end_index, and self.silent_start (the index of the first silent state).

Parameters **verbose** : bool, optional

Return a log of changes made to the model during normalization or merging. Default is False.

merge : "None", "Partial, "All"

Merging has three options: "None": No modifications will be made to the model. "Partial": A silent state which only has a probability 1 transition

to another silent state will be merged with that silent state. This means that if silent state "S1" has a single transition to silent state "S2", that all transitions to S1 will now go to S2, with the same probability as before, and S1 will be removed from the model.

"All": A silent state with a probability 1 transition to any other state, silent or symbol emitting, will be merged in the manner described above. In addition, any orphan states will be removed from the model. An orphan state is a state which does not have any transitions to it OR does not have any transitions from it, except for the start and end of the model. This will iteratively remove orphan chains from the model. This is sometimes desirable, as all states should have both a transition in to get to that state, and a transition out, even if it is only to itself. If the state does not have either, the HMM will likely not work as intended.

Default is 'All'.

Returns None

clear_summaries()

Clear the summary statistics stored in the object.

Parameters **None**

Returns None

concatenate()

Concatenate this model to another model.

Concatenate this model to another model in such a way that a single probability 1 edge is added between self.end and other.start. Rename all other states appropriately by adding a suffix or prefix if needed.

Parameters **other** : HiddenMarkovModel

The other model to concatenate

suffix : str, optional

Add the suffix to the end of all state names in the other model. Default is ''.

prefix : str, optional

Add the prefix to the beginning of all state names in the other model. Default is ''.

Returns None

copy()

Returns a deep copy of the HMM.

Parameters **None**

Returns model : HiddenMarkovModel

A deep copy of the model with entirely new objects.

dense_transition_matrix()
Returns the dense transition matrix.

Parameters None

Returns matrix : numpy.ndarray, shape (n_states, n_states)

A dense transition matrix, containing the log probability of transitioning from each state to each other state.

edge_count()
Returns the number of edges present in the model.

fit()
Fit the model to data using either Baum-Welch or Viterbi training.

Given a list of sequences, performs re-estimation on the model parameters. The two supported algorithms are "baum-welch" and "viterbi," indicating their respective algorithm.

Training supports a wide variety of other options including using edge pseudocounts and either edge or distribution inertia.

Parameters sequences : array-like

An array of some sort (list, numpy.ndarray, tuple..) of sequences, where each sequence is a numpy array, which is 1 dimensional if the HMM is a one dimensional array, or multidimensional of the HMM supports multiple dimensions.

weights : array-like or None, optional

An array of weights, one for each sequence to train on. If None, all sequences are equaly weighted. Default is None.

stop_threshold : double, optional

The threshold the improvement ratio of the models log probability in fitting the scores. Default is 1e-9.

min_iterations : int, optional

The minimum number of iterations to run Baum-Welch training for. Default is 0.

max_iterations : int, optional

The maximum number of iterations to run Baum-Welch training for. Default is 1e8.

algorithm : 'baum-welch', 'viterbi'

The training algorithm to use. Baum-Welch uses the forward-backward algorithm to train using a version of structured EM. Viterbi iteratively runs the sequences through the Viterbi algorithm and then uses hard assignments of observations to states using that. Default is 'baum-welch'.

verbose : bool, optional

Whether to print the improvement in the model fitting at each iteration. Default is True.

transition_pseudocount : int, optional

A pseudocount to add to all transitions to add a prior to the MLE estimate of the transition probability. Default is 0.

use_pseudocount : bool, optional

Whether to use pseudocounts when updatiing the transition probability parameters. Default is False.

inertia : double or None, optional, range [0, 1]

If double, will set both edge_inertia and distribution_inertia to be that value. If None, will not override those values. Default is None.

edge_inertia : bool, optional, range [0, 1]

Whether to use inertia when updating the transition probability parameters. Default is 0.0.

distribution_inertia : double, optional, range [0, 1]

Whether to use inertia when updating the distribution parameters. Default is 0.0.

n_jobs : int, optional

The number of threads to use when performing training. This leads to exact updates. Default is 1.

Returns improvement : double

The total improvement in fitting the model to the data

forward()

Run the forward algorithm on the sequence.

Calculate the probability of each observation being aligned to each state by going forward through a sequence. Returns the full forward matrix. Each index i, j corresponds to the sum-of-all-paths log probability of starting at the beginning of the sequence, and aligning observations to hidden states in such a manner that observation i was aligned to hidden state j. Uses row normalization to dynamically scale each row to prevent underflow errors.

If the sequence is impossible, will return a matrix of nans.

See also:

- Silent state handling taken from p. 71 of "Biological

Sequence Analysis" by Durbin et al., and works for anything which does not have loops of silent states.

•Row normalization technique explained by

http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf on p. 14.

Parameters sequence : array-like

An array (or list) of observations.

Returns matrix : array-like, shape (len(sequence), n_states)

The probability of aligning the sequences to states in a forward fashion.

forward_backward()

Run the forward-backward algorithm on the sequence.

This algorithm returns an emission matrix and a transition matrix. The emission matrix returns the normalized probability that each each state generated that emission given both the symbol and the entire sequence. The transition matrix returns the expected number of times that a transition is used.

If the sequence is impossible, will return (None, None)

See also:

- Forward and backward algorithm implementations. A comprehensive

description of the forward, backward, and forward-background algorithm is here: http://en.wikipedia.org/wiki/Forward%E2%80%93backward_algorithm

Parameters **sequence** : array-like

An array (or list) of observations.

Returns **emissions** : array-like, shape (len(sequence), n_nonsilent_states)

The normalized probabilities of each state generating each emission.

transitions : array-like, shape (n_states, n_states)

The expected number of transitions across each edge in the model.

freeze()
Freeze the distribution, preventing updates from occuring.

freeze_distributions()
Freeze all the distributions in model.

Upon training only edges will be updated. The parameters of distributions will not be affected.

Parameters **None**

Returns None

from_json()
Read in a serialized model and return the appropriate classifier.

Parameters **s** : str

A JSON formatted string containing the file.

Returns

——-

model : object

A properly initialized and baked model.

from_matrix()
Create a model from a more standard matrix format.

Take in a 2D matrix of floats of size n by n, which are the transition probabilities to go from any state to any other state. May also take in a list of length n representing the names of these nodes, and a model name. Must provide the matrix, and a list of size n representing the distribution you wish to use for that state, a list of size n indicating the probability of starting in a state, and a list of size n indicating the probability of ending in a state.

Parameters **transition_probabilities** : array-like, shape (n_states, n_states)

The probabilities of each state transitioning to each other state.

distributions : array-like, shape (n_states)

The distributions for each state. Silent states are indicated by using None instead of a distribution object.

starts : array-like, shape (n_states)

The probabilities of starting in each of the states.

ends : array-like, shape (n_states), optional

If passed in, the probabilities of ending in each of the states. If ends is None, then assumes the model has no explicit end state. Default is None.

state_names : array-like, shape (n_states), optional

The name of the states. If None is passed in, default names are generated. Default is None

name : str, optional

The name of the model. Default is None

verbose : bool, optional

The verbose parameter for the underlying bake method. Default is False.

merge : 'None', 'Partial', 'All', optional

The merge parameter for the underlying bake method. Default is All

Returns **model** : HiddenMarkovModel

The baked model ready to go.

Examples

matrix = [[0.4, 0.5], [0.4, 0.5]] distributions = [NormalDistribution(1, .5), NormalDistribution(5, 2)] starts = [1., 0.] ends = [.1., .1] state_names= ["A", "B"]

model = Model.from_matrix(matrix, distributions, starts, ends, state_names, name="test_model")

from_summaries()

Fit the model to the stored summary statistics.

Parameters **inertia** : double or None, optional

The inertia to use for both edges and distributions without needing to set both of them. If None, use the values passed in to those variables. Default is None.

transition_pseudocount : int, optional

A pseudocount to add to all transitions to add a prior to the MLE estimate of the transition probability. Default is 0.

use_pseudocount : bool, optional

Whether to use pseudocounts when updatiing the transition probability parameters. Default is False.

edge_inertia : bool, optional, range [0, 1]

Whether to use inertia when updating the transition probability parameters. Default is 0.0.

distribution_inertia : double, optional, range [0, 1]

Whether to use inertia when updating the distribution parameters. Default is 0.0.

Returns None

log_probability()

Calculate the log probability of a single sequence.

If a path is provided, calculate the log probability of that sequence given the path.

Parameters **sequence** : array-like

Return the array of observations in a single sequence of data

check_input : bool, optional

Check to make sure that all emissions fall under the support of the emission distributions. Default is True.

Returns **logp** : double

The log probability of the sequence

maximum_a_posteriori()

Run posterior decoding on the sequence.

MAP decoding is an alternative to viterbi decoding, which returns the most likely state for each observation, based on the forward-backward algorithm. This is also called posterior decoding. This method is described on p. 14 of http://ai.stanford.edu/~serafim/CS262_2007/ notes/lecture5.pdf

WARNING: This may produce impossible sequences.

Parameters **sequence** : array-like

An array (or list) of observations.

Returns **logp** : double

The log probability of the sequence under the Viterbi path

path : list of tuples

Tuples of (state index, state object) of the states along the posterior path.

node_count()

Returns the number of nodes/states in the model

plot()

Draw this model's graph using NetworkX and matplotlib.

Note that this relies on networkx's built-in graphing capabilities (and not Graphviz) and thus can't draw self-loops.

See networkx.draw_networkx() for the keywords you can pass in.

Parameters **precision** : int, optional

The precision with which to round edge probabilities. Default is 4.

****kwargs** : any

The arguments to pass into networkx.draw_networkx()

Returns None

predict()

Calculate the most likely state for each observation.

This can be either the Viterbi algorithm or maximum a posteriori. It returns the probability of the sequence under that state sequence and the actual state sequence.

This is a sklearn wrapper for the Viterbi and maximum_a_posteriori methods.

Parameters **sequence** : array-like

An array (or list) of observations.

algorithm : "map", "viterbi"

The algorithm with which to decode the sequence

Returns **logp** : double

The log probability of the sequence under the Viterbi path

path : list of tuples

Tuples of (state index, state object) of the states along the Viterbi path.

predict_log_proba()

Calculate the state log probabilities for each observation in the sequence.

Run the forward-backward algorithm on the sequence and return the emission matrix. This is the log normalized probability that each each state generated that emission given both the symbol and the entire sequence.

This is a sklearn wrapper for the forward backward algorithm.

See also:

- Forward and backward algorithm implementations. A comprehensive

description of the forward, backward, and forward-background algorithm is here: http://en.wikipedia.org/wiki/Forward%E2%80%93backward_algorithm

Parameters **sequence** : array-like

An array (or list) of observations.

Returns **emissions** : array-like, shape (len(sequence), n_nonsilent_states)

The log normalized probabilities of each state generating each emission.

predict_proba()

Calculate the state probabilities for each observation in the sequence.

Run the forward-backward algorithm on the sequence and return the emission matrix. This is the normalized probability that each each state generated that emission given both the symbol and the entire sequence.

This is a sklearn wrapper for the forward backward algorithm.

See also:

- Forward and backward algorithm implementations. A comprehensive

description of the forward, backward, and forward-background algorithm is here: http://en.wikipedia.org/wiki/Forward%E2%80%93backward_algorithm

Parameters **sequence** : array-like

An array (or list) of observations.

Returns **emissions** : array-like, shape (len(sequence), n_nonsilent_states)

The normalized probabilities of each state generating each emission.

probability()

Return the probability of the given symbol under this distribution.

Parameters **symbol** : object

The symbol to calculate the probability of

Returns **probability** : double

The probability of that point under the distribution.

sample()
Generate a sequence from the model.

Returns the sequence generated, as a list of emitted items. The model must have been baked first in order to run this method.

If a length is specified and the HMM is infinite (no edges to the end state), then that number of samples will be randomly generated. If the length is specified and the HMM is finite, the method will attempt to generate a prefix of that length. Currently it will force itself to not take an end transition unless that is the only path, making it not a true random sample on a finite model.

WARNING: If the HMM has no explicit end state, must specify a length to use.

Parameters **length** : int, optional

Generate a sequence with a maximal length of this size. Used if you have no explicit end state. Default is 0.

path : bool, optional

Return the path of hidden states in addition to the emissions. If true will return a tuple of (sample, path). Default is False.

Returns **sample** : list or tuple

If path is true, return a tuple of (sample, path), otherwise return just the samples.

state_count()
Returns the number of states present in the model.

summarize()
Summarize data into stored sufficient statistics for out-of-core training. Only implemented for Baum-Welch training since Viterbi is less memory intensive.

Parameters **sequences** : array-like

An array of some sort (list, numpy.ndarray, tuple..) of sequences, where each sequence is a numpy array, which is 1 dimensional if the HMM is a one dimensional array, or multidimensional of the HMM supports multiple dimensions.

weights : array-like or None, optional

An array of weights, one for each sequence to train on. If None, all sequences are equaly weighted. Default is None.

n_jobs : int, optional

The number of threads to use when performing training. This leads to exact updates. Default is 1.

algorithm : 'baum-welch' or 'viterbi', optional

The algorithm to use to collect the statistics, either Baum-Welch or Viterbi training. Defaults to Baum-Welch.

parallel : joblib.Parallel or None, optional

The joblib threadpool. Passed between iterations of Baum-Welch so that a new thread-pool doesn't have to be created each iteration. Default is None.

check_input : bool, optional

Check the input. This casts the input sequences as numpy arrays, and converts non-numeric inputs into numeric inputs for faster processing later. Default is True.

Returns **logp** : double

The log probability of the sequences.

thaw()
Thaw the distribution, re-allowing updates to occur.

thaw_distributions()
Thaw all distributions in the model.

Upon training distributions will be updated again.

Parameters **None**

Returns None

to_json()
Serialize the model to a JSON.

Parameters **separators** : tuple, optional

The two separaters to pass to the json.dumps function for formatting.

indent : int, optional

The indentation to use at each level. Passed to json.dumps for formatting.

Returns **json** : str

A properly formatted JSON object.

viterbi()
Run the Viteri algorithm on the sequence.

Run the Viterbi algorithm on the sequence given the model. This finds the ML path of hidden states given the sequence. Returns a tuple of the log probability of the ML path, or (-inf, None) if the sequence is impossible under the model. If a path is returned, it is a list of tuples of the form (sequence index, state object).

This is fundamentally the same as the forward algorithm using max instead of sum, except the traceback is more complicated, because silent states in the current step can trace back to other silent states in the current step as well as states in the previous step.

See also:

- Viterbi implementation described well in the wikipedia article

http://en.wikipedia.org/wiki/Viterbi_algorithm

Parameters **sequence** : array-like

An array (or list) of observations.

Returns **logp** : double

The log probability of the sequence under the Viterbi path

path : list of tuples

Tuples of (state index, state object) of the states along the Viterbi path.

pomegranate.hmm.**log**()
Return the natural log of the value or -infinity if the value is 0.

1.6 Naive Bayes Classifiers

IPython Notebook Tutorial

The Naive Bayes classifier is a simple probabilistic classification model based on Bayes Theorem. Since Naive Bayes classifiers classifies sets of data by which class has the highest conditional probability, Naive Bayes classifiers can use any distribution or model which has a probabilistic interpretation of the data as one of its components. Basically if it can output a log probability, then it can be used in Naive Bayes.

An IPython notebook example demonstrating a Naive Bayes classifier using multivariate distributions can be found here.

1.6.1 Initialization

Naive Bayes can be initialized in two ways, either by (1) passing in pre-initialized models as a list, or by (2) passing in the constructor and the number of components for simple distributions. For example, here is how you can create a Naive bayes classifier which compares a normal distribution to a uniform distribution to an exponential distribution:

```
>>> from pomegranate import *
>>> clf = NaiveBayes([ NormalDistribution(5, 2), UniformDistribution(0, 10), ExponentialDistribution
```

An advantage of initializing the classifier this way is that you can use pre-trained or known-before-hand models to make predictions. A disadvantage is that if we don't have any prior knowledge as to what the distributions should be then we have to make up distributions to start off with. If all of the models in the classifier use the same type of model then we can pass in the constructor for that model and the number of classes that there are.

```
>>> from pomegranate import *
>>> clf = NaiveBayes(NormalDistribution, n_components=5)
```

Warning: If we initialize a naive Bayes classifier in this manner we must fit the model before we can use it to predict.

An advantage of doing it this way is that we don't need to make dummy distributions just to train, but a disadvantage is that we have to train the model before we can use it.

Since Naive Bayes classifiers simply compares the likelihood of a sample occurring under different models, it can be initialized with any model in pomegranate. This is assuming that all the models take the same type of input.

```
>>> from pomegranate import *
>>> d1 = MultivariateGaussianDistribution([5, 5], [[1, 0], [0, 1]])
>>> d2 = IndependentComponentsDistribution([NormalDistribution(5, 2), NormalDistribution(5, 2)])
>>> clf = NaiveBayes([d1, d2])
```

Note: This is no longer strictly a "naive" Bayes classifier if we are using more complicated models. However, much of the underlying math still holds.

1.6.2 Prediction

Naive Bayes supports the same three prediction methods that the other models support, namely `predict`, `predict_proba`, and `predict_log_proba`. These methods return the most likely class given the data, the probability of each class given the data, and the log probability of each class given the data.

The `predict` method takes in samples and returns the most likely class given the data.

```
>>> from pomegranate import *
>>> clf = NaiveBayes([ NormalDistribution( 5, 2 ), UniformDistribution( 0, 10 ), ExponentialDistribut
>>> clf.predict( np.array([ 0, 1, 2, 3, 4 ]) )
[ 2, 2, 2, 0, 0 ]
```

Calling `predict_proba` on five samples for a Naive Bayes with univariate components would look like the following.

```
>>> from pomegranate import *
>>> clf = NaiveBayes([NormalDistribution(5, 2), UniformDistribution(0, 10), ExponentialDistribution(1
>>> clf.predict_proba(np.array([ 0, 1, 2, 3, 4]))
[[ 0.00790443  0.09019051  0.90190506]
 [ 0.05455011  0.20207126  0.74337863]
 [ 0.21579499  0.33322883  0.45097618]
 [ 0.44681566  0.36931382  0.18387052]
 [ 0.59804205  0.33973357  0.06222437]]
```

Multivariate models work the same way except that the input has to have the same number of columns as are represented in the model, like the following.

```
>>> from pomegranate import *
>>> d1 = MultivariateGaussianDistribution([5, 5], [[1, 0], [0, 1]])
>>> d2 = IndependentComponentsDistribution([NormalDistribution(5, 2), NormalDistribution(5, 2)])
>>> clf = NaiveBayes([d1, d2])
>>> clf.predict_proba(np.array([[0, 4],
                                                          [1, 3],
                                                          [2, 2],
                                                          [3, 1],
                                                          [4, 0]]))
array([[ 0.00023312,  0.99976688],
       [ 0.00220745,  0.99779255],
       [ 0.00466169,  0.99533831],
       [ 0.00220745,  0.99779255],
       [ 0.00023312,  0.99976688]])
```

`predict_log_proba` works in a similar way except that it returns the log probabilities instead of the actual probabilities.

1.6.3 Fitting

Naive Bayes has a fit method, in which the models in the classifier are trained to "fit" to a set of data. The method takes two numpy arrays as input, an array of samples and an array of correct classifications for each sample. Here is an example for a Naive Bayes made up of two bivariate distributions.

```
>>> from pomegranate import *
>>> d1 = MultivariateGaussianDistribution([5, 5], [[1, 0], [0, 1]])
>>> d2 = IndependentComponentsDistribution(NormalDistribution(5, 2), NormalDistribution(5, 2)])
>>> clf = NaiveBayes([d1, d2])
>>> X = np.array([[6.0, 5.0],
                  [3.5, 4.0],
                  [7.5, 1.5],
                      [7.0, 7.0 ]])
>>> y = np.array([0, 0, 1, 1])
>>> clf.fit(X, y)
```

As we can see, there are four samples, with the first two samples labeled as class 0 and the last two samples labeled as class 1. Keep in mind that the training samples must match the input requirements for the models used. So if using

a univariate distribution, then each sample must contain one item. A bivariate distribution, two. For hidden markov models, the sample can be a list of observations of any length. An example using hidden markov models would be the following.

```
>>> X = np.array([list( 'HHHHHTHTHTTTTH' ),
                                            list( 'HHTHHTTHHHHHTH' ),
                                            list( 'TH' ),
                                            list( 'HHHHT' )])
>>> y = np.array([2, 2, 1, 0])
>>> clf.fit(X, y)
```

1.6.4 API Reference

Naive Bayes estimator, for anything with a log_probability method.

class pomegranate.NaiveBayes.**NaiveBayes**

A Naive Bayes model, a supervised alternative to GMM.

Parameters **models** : list or constructor

Must either be a list of initialized distribution/model objects, or the constructor for a distribution object:

- Initialized : NaiveBayes([NormalDistribution(1, 2), NormalDistribution(0, 1)])
- Constructor : NaiveBayes(NormalDistribution)

weights : list or numpy.ndarray or None, default None

The prior probabilities of the components. If None is passed in then defaults to the uniformly distributed priors.

Examples

```
>>> from pomegranate import *
>>> clf = NaiveBayes( NormalDistribution )
>>> X = [0, 2, 0, 1, 0, 5, 6, 5, 7, 6]
>>> y = [0, 0, 0, 0, 0, 1, 1, 0, 1, 1]
>>> clf.fit(X, y)
>>> clf.predict_proba([6])
array([[ 0.01973451,  0.98026549]])
```

```
>>> from pomegranate import *
>>> clf = NaiveBayes([NormalDistribution(1, 2), NormalDistribution(0, 1)])
>>> clf.predict_log_proba([[0], [1], [2], [-1]])
array([[-1.1836569 , -0.36550972],
           [-0.79437677, -0.60122959],
           [-0.26751248, -1.4493653 ],
           [-1.09861229, -0.40546511]])
```

Attributes

models	(list) The model objects, either initialized by the user or fit to data.
weights	(numpy.ndarray) The prior probability of each component of the model.

clear_summaries()

Clear the summary statistics stored in the object.

copy()
Return a deep copy of this distribution object.

This object will not be tied to any other distribution or connected in any form.

Parameters **None**

Returns **distribution** : Distribution

A copy of the distribution with the same parameters.

fit()
Fit the Naive Bayes model to the data by passing data to their components.

Parameters **X** : numpy.ndarray or list

The dataset to operate on. For most models this is a numpy array with columns corresponding to features and rows corresponding to samples. For markov chains and HMMs this will be a list of variable length sequences.

y : numpy.ndarray or list or None, optional

Data labels for supervised training algorithms. Default is None

weights : array-like or None, shape (n_samples,), optional

The initial weights of each sample in the matrix. If nothing is passed in then each sample is assumed to be the same weight. Default is None.

n_jobs : int

The number of jobs to use to parallelize, either the number of threads or the number of processes to use. Default is 1.

inertia : double, optional

Inertia used for the training the distributions.

Returns **self** : object

Returns the fitted model

freeze()
Freeze the distribution, preventing updates from occuring.

from_summaries()
Fit the Naive Bayes model to the stored sufficient statistics.

Parameters **inertia** : double, optional

Inertia used for the training the distributions.

Returns **self** : object

Returns the fitted model

log_probability()
Return the log probability of the given symbol under this distribution.

Parameters **symbol** : double

The symbol to calculate the log probability of (overriden for DiscreteDistributions)

Returns **logp** : double

The log probability of that point under the distribution.

predict()
Predict the most likely component which generated each sample.

Calculate the posterior P(M|D) for each sample and return the index of the component most likely to fit it. This corresponds to a simple argmax over the responsibility matrix.

This is a sklearn wrapper for the maximum_a_posteriori method.

Parameters **X** : array-like, shape (n_samples, n_dimensions)

The samples to do the prediction on. Each sample is a row and each column corresponds to a dimension in that sample. For univariate distributions, a single array may be passed in.

Returns **y** : array-like, shape (n_samples,)

The predicted component which fits the sample the best.

predict_log_proba()
Calculate the posterior log P(M|D) for data.

Calculate the log probability of each item having been generated from each component in the model. This returns normalized log probabilities such that the probabilities should sum to 1

This is a sklearn wrapper for the original posterior function.

Parameters **X** : array-like, shape (n_samples, n_dimensions)

The samples to do the prediction on. Each sample is a row and each column corresponds to a dimension in that sample. For univariate distributions, a single array may be passed in.

Returns **y** : array-like, shape (n_samples, n_components)

The normalized log probability log P(M|D) for each sample. This is the probability that the sample was generated from each component.

predict_proba()
Calculate the posterior P(M|D) for data.

Calculate the probability of each item having been generated from each component in the model. This returns normalized probabilities such that each row should sum to 1.

Since calculating the log probability is much faster, this is just a wrapper which exponentiates the log probability matrix.

Parameters **X** : array-like, shape (n_samples, n_dimensions)

The samples to do the prediction on. Each sample is a row and each column corresponds to a dimension in that sample. For univariate distributions, a single array may be passed in.

Returns **probability** : array-like, shape (n_samples, n_components)

The normalized probability P(M|D) for each sample. This is the probability that the sample was generated from each component.

probability()
Return the probability of the given symbol under this distribution.

Parameters **symbol** : object

The symbol to calculate the probability of

Returns **probability** : double

The probability of that point under the distribution.

sample()
Return a random item sampled from this distribution.

Parameters **n** : int or None, optional

The number of samples to return. Default is None, which is to generate a single sample.

Returns **sample** : double or object

Returns a sample from the distribution of a type in the support of the distribution.

summarize()
Summarize data into stored sufficient statistics for out-of-core training.

Parameters **X** : array-like, shape (n_samples, variable)

Array of the samples, which can be either fixed size or variable depending on the underlying components.

y : array-like, shape (n_samples,)

Array of the known labels as integers

weights : array-like, shape (n_samples,) optional

Array of the weight of each sample, a positive float

n_jobs : int

The number of jobs to use to parallelize, either the number of threads or the number of processes to use. Default is 1.

Returns None

thaw()
Thaw the distribution, re-allowing updates to occur.

1.7 Markov Chains

IPython Notebook Tutorial

Markov chains are form of structured model over sequences. They represent the probability of each character in the sequence as a conditional probability of the last k symbols. For example, a 3rd order Markov chain would have each symbol depend on the last three symbols. A 0th order Markov chain is a naive predictor where each symbol is independent of all other symbols. Currently pomegranate only supports discrete emission Markov chains where each symbol is a discrete symbol versus a continuous number (like 'A' 'B' 'C' instead of 17.32 or 19.65).

1.7.1 Initialization

Markov chains can almost be represented by a single conditional probability table (CPT), except that the probability of the first k elements (for a k-th order Markov chain) cannot be appropriately represented except by using special characters. Due to this pomegranate takes in a series of k+1 distributions representing the first k elements. For example for a second order Markov chain:

```
>>> from pomegranate import *
>>> d1 = DiscreteDistribution({'A': 0.25, 'B': 0.75})
>>> d2 = ConditionalProbabilityTable([['A', 'A', 0.1],
                                      ['A', 'B', 0.9],
```

```
                                        ['B', 'A', 0.6],
                                        ['B', 'B', 0.4]], [d1])
>>> d3 = ConditionalProbabilityTable([['A', 'A', 'A', 0.4],
                                      ['A', 'A', 'B', 0.6],
                                      ['A', 'B', 'A', 0.8],
                                      ['A', 'B', 'B', 0.2],
                                      ['B', 'A', 'A', 0.9],
                                      ['B', 'A', 'B', 0.1],
                                      ['B', 'B', 'A', 0.2],
                                      ['B', 'B', 'B', 0.8]], [d1, d2])
>>> model = MarkovChain([d1, d2, d3])
```

1.7.2 Probability

The probability of a sequence under the Markov chain is just the probabiliy of the first character under the first distribution times the probability of the second character under the second distribution and so forth until you go past the (k+1)th character, which remains evaluated under the (k+1)th distribution. We can calculate the probability or log probability in the same manner as any of the other models. Given the model shown before:

```
>>> model.log_probability(['A', 'B', 'B', 'B'])
-3.324236340526027
>>> model.log_probability(['A', 'A', 'A', 'A'])
-5.521460917862246
```

1.7.3 Fitting

Markov chains are not very complicated to chain. For each sequence the appropriate symbols are sent to the appropriate distributions and maximum likelihood estimates are used to update the parameters of the distributions. There are no latent factors to train and so no expectation maximization or iterative algorithms are needed to train anything.

1.7.4 API Reference

class pomegranate.MarkovChain.**MarkovChain**

A Markov Chain.

Implemented as a series of conditional distributions, the Markov chain models P(X_i | X_i-1...X_i-k) for a k-th order Markov network. The conditional dependencies are directly on the emissions, and not on a hidden state as in a hidden Markov model.

Parameters **distributions** : list, shape (k+1)

A list of the conditional distributions which make up the markov chain. Begins with P(X_i), then P(X_i | X_i-1). For a k-th order markov chain you must put in k+1 distributions.

Examples

```
>>> from pomegranate import *
>>> d1 = DiscreteDistribution({'A': 0.25, 'B': 0.75})
>>> d2 = ConditionalProbabilityTable([['A', 'A', 0.33],
                                      ['B', 'A', 0.67],
                                      ['A', 'B', 0.82],
                                      ['B', 'B', 0.18]], [d1])
```

```
>>> mc = MarkovChain([d1, d2])
>>> mc.log_probability(list('ABBAABABABAABABA'))
-8.9119890701808213
```

Attributes

distributions	(list, shape (k+1)) The distributions which make up the chain.

fit()

Fit the model to new data using MLE.

The underlying distributions are fed in their appropriate points and weights and are updated.

Parameters **sequences** : array-like, shape (n_samples, variable)

This is the data to train on. Each row is a sample which contains a sequence of variable length

weights : array-like, shape (n_samples,), optional

The initial weights of each sample. If nothing is passed in then each sample is assumed to be the same weight. Default is None.

inertia : double, optional

The weight of the previous parameters of the model. The new parameters will roughly be old_param*inertia + new_param*(1-inertia), so an inertia of 0 means ignore the old parameters, whereas an inertia of 1 means ignore the new parameters. Default is 0.0.

Returns None

from_json()

Read in a serialized model and return the appropriate classifier.

Parameters **s** : str

A JSON formatted string containing the file.

Returns **model** : object

A properly initialized and baked model.

from_samples()

Learn the Markov chain from data.

Takes in the memory of the chain (k) and learns the initial distribution and probability tables associated with the proper parameters.

Parameters **X** : array-like, list or numpy.array

The data to fit the structure too as a list of sequences of variable length. Since the data will be of variable length, there is no set form

weights : array-like, shape (n_nodes), optional

The weight of each sample as a positive double. Default is None.

k : int, optional

The number of samples back to condition on in the model. Default is 1.

Returns **model** : MarkovChain

The learned markov chain model.

from_summaries()
Fit the model to the collected sufficient statistics.

Fit the parameters of the model to the sufficient statistics gathered during the summarize calls. This should return an exact update.

Parameters **inertia** : double, optional

The weight of the previous parameters of the model. The new parameters will roughly be old_param*inertia + new_param * (1-inertia), so an inertia of 0 means ignore the old parameters, whereas an inertia of 1 means ignore the new parameters. Default is 0.0.

Returns None

log_probability()
Calculate the log probability of the sequence under the model.

This calculates the first slices of increasing size under the corresponding first few components of the model until size k is reached, at which all slices are evaluated under the final component.

Parameters **sequence** : array-like

An array of observations

Returns **logp** : double

The log probability of the sequence under the model.

sample()
Create a random sample from the model.

Parameters **length** : int or Distribution

Give either the length of the sample you want to generate, or a distribution object which will be randomly sampled for the length. Continuous distributions will have their sample rounded to the nearest integer, minimum 1.

Returns **sequence** : array-like, shape = (length,)

A sequence randomly generated from the markov chain.

summarize()
Summarize a batch of data and store sufficient statistics.

This will summarize the sequences into sufficient statistics stored in each distribution.

Parameters **sequences** : array-like, shape (n_samples, variable)

This is the data to train on. Each row is a sample which contains a sequence of variable length

weights : array-like, shape (n_samples,), optional

The initial weights of each sample. If nothing is passed in then each sample is assumed to be the same weight. Default is None.

Returns None

to_json()
Serialize the model to a JSON.

Parameters **separators** : tuple, optional

The two separaters to pass to the json.dumps function for formatting. Default is (‘,’, ‘ : ‘).

indent : int, optional

The indentation to use at each level. Passed to json.dumps for formatting. Default is 4.

Returns **json** : str

A properly formatted JSON object.

1.8 Bayesian Networks

IPython Notebook Tutorial

Bayesian networks are a powerful inference tool, in which nodes represent some random variable we care about, edges represent dependencies and a lack of an edge between two nodes represents a conditional independence. A powerful algorithm called the sum-product or forward-backward algorithm allows for inference to be done on this network, calculating posteriors on unobserved ("hidden") variables when limited information is given. The more information is known, the better the inference will be, but there is no requirement on the number of nodes which must be observed. If no information is given, the marginal of the graph is trivially calculated. The hidden and observed variables do not need to be explicitly defined when the network is set, they simply exist based on what information is given.

Lets test out the Bayesian Network framework on the Monty Hall problem. The Monty Hall problem arose from the gameshow *Let's Make a Deal*, where a guest had to choose which one of three doors had a prize behind it. The twist was that after the guest chose, the host, originally Monty Hall, would then open one of the doors the guest did not pick and ask if the guest wanted to switch which door they had picked. Initial inspection may lead you to believe that if there are only two doors left, there is a 50-50 chance of you picking the right one, and so there is no advantage one way or the other. However, it has been proven both through simulations and analytically that there is in fact a 66% chance of getting the prize if the guest switches their door, regardless of the door they initially went with.

We can reproduce this result using Bayesian networks with three nodes, one for the guest, one for the prize, and one for the door Monty chooses to open. The door the guest initially chooses and the door the prize is behind are completely random processes across the three doors, but the door which Monty opens is dependent on both the door the guest chooses (it cannot be the door the guest chooses), and the door the prize is behind (it cannot be the door with the prize behind it).

```
import math
from pomegranate import *

# The guests initial door selection is completely random
guest = DiscreteDistribution( { 'A': 1./3, 'B': 1./3, 'C': 1./3 } )

# The door the prize is behind is also completely random
prize = DiscreteDistribution( { 'A': 1./3, 'B': 1./3, 'C': 1./3 } )

        # Monty is dependent on both the guest and the prize.
        monty = ConditionalProbabilityTable(
                [[ 'A', 'A', 'A', 0.0 ],
                 [ 'A', 'A', 'B', 0.5 ],
                 [ 'A', 'A', 'C', 0.5 ],
                 [ 'A', 'B', 'A', 0.0 ],
                 [ 'A', 'B', 'B', 0.0 ],
                 [ 'A', 'B', 'C', 1.0 ],
                 [ 'A', 'C', 'A', 0.0 ],
                 [ 'A', 'C', 'B', 1.0 ],
                 [ 'A', 'C', 'C', 0.0 ],
                 [ 'B', 'A', 'A', 0.0 ],
                 [ 'B', 'A', 'B', 0.0 ],
                 [ 'B', 'A', 'C', 1.0 ],
                 [ 'B', 'B', 'A', 0.5 ],
                 [ 'B', 'B', 'B', 0.0 ],
```

```
                  [ 'B', 'B', 'C', 0.5 ],
                  [ 'B', 'C', 'A', 1.0 ],
                  [ 'B', 'C', 'B', 0.0 ],
                  [ 'B', 'C', 'C', 0.0 ],
                  [ 'C', 'A', 'A', 0.0 ],
                  [ 'C', 'A', 'B', 1.0 ],
                  [ 'C', 'A', 'C', 0.0 ],
                  [ 'C', 'B', 'A', 1.0 ],
                  [ 'C', 'B', 'B', 0.0 ],
                  [ 'C', 'B', 'C', 0.0 ],
                  [ 'C', 'C', 'A', 0.5 ],
                  [ 'C', 'C', 'B', 0.5 ],
                  [ 'C', 'C', 'C', 0.0 ]], [guest, prize] )

s1 = State( guest, name="guest" )
s2 = State( prize, name="prize" )
s3 = State( monty, name="monty" )

network = BayesianNetwork( "Monty Hall Problem" )
network.add_states(s1, s2, s3)
network.add_edge(s1, s3)
network.add_edge(s2, s3)
network.bake()
```

Bayesian Networks utilize ConditionalProbabilityTable objects to represent conditional distributions. This distribution is made up of a table where each column represents the parent (or self) values except for the last column which represents the probability of the variable taking on that value given its parent values. It also takes in a list of parent distribution objects in the same order that they are used in the table. In the Monty Hall example, the monty distribution is dependent on both the guest and the prize distributions in that order and so the first column of the CPT is the value the guest takes and the second column is the value that the prize takes.

The next step is to make predictions using this model. One of the strengths of Bayesian networks is their ability to infer the values of arbitrary 'hidden variables' given the values from 'observed variables.' These hidden and observed variables do not need to be specified beforehand, and the more variables which are observed the better the inference will be on the hidden variables.

Lets say that the guest chooses door 'A'. guest becomes an observed variable, while both prize and monty are hidden variables.

... code-block:: python

```
>>> beliefs = network.predict_proba({ 'guest' : 'A' })
>>> beliefs = map(str, beliefs)
>>> print "\n".join( "{}\t{}".format( state.name, belief ) for state, belief in zip( network.states,
prize   DiscreteDistribution({'A': 0.3333333333333335, 'C': 0.3333333333333333, 'B': 0.3333333333333
guest   DiscreteDistribution({'A': 1.0, 'C': 0.0, 'B': 0.0})
monty   DiscreteDistribution({'A': 0.0, 'C': 0.5, 'B': 0.5})
```

Since we've observed the value that guest takes, we know there is a 100% chance it is that value. The prize distribution is unaffected because it is independent of the guest variable given that we don't know the door that Monty opens.

Now the next step is for Monty to open a door. Let's say that Monty opens door 'b':

```
>>> beliefs = network.predict_proba({'guest' : 'A', 'monty' : 'B'})
>>> print "\n".join( "{}\t{}".format( state.name, str(belief) ) for state, belief in zip( network.sta
guest   DiscreteDistribution({'A': 1.0, 'C': 0.0, 'B': 0.0})
monty   DiscreteDistribution({'A': 0.0, 'C': 0.0, 'B': 1.0})
prize   DiscreteDistribution({'A': 0.3333333333333333, 'C': 0.6666666666666666, 'B': 0.0})
```

We've observed both guest and Monty so there is a 100% chance for those values. However, we see that probability of prize being 'C' is 66% mimicking the mystery behind the Monty hall problem!

1.8.1 API Reference

class pomegranate.BayesianNetwork.**BayesianNetwork**

A Bayesian Network Model.

A Bayesian network is a directed graph where nodes represent variables, edges represent conditional dependencies of the children on their parents, and the lack of an edge represents a conditional independence.

Parameters **name** : str, optional

The name of the model. Default is None

Attributes

states	(list, shape (n_states,)) A list of all the state objects in the model
graph	(networkx.DiGraph) The underlying graph object.

add_edge()

Add a transition from state a to state b which indicates that B is dependent on A in ways specified by the distribution.

add_node()

Add a node to the graph.

add_nodes()

Add multiple states to the graph.

add_state()

Another name for a node.

add_states()

Another name for a node.

add_transition()

Transitions and edges are the same.

bake()

Finalize the topology of the model.

Assign a numerical index to every state and create the underlying arrays corresponding to the states and edges between the states. This method must be called before any of the probability-calculating methods. This includes converting conditional probability tables into joint probability tables and creating a list of both marginal and table nodes.

Parameters **None**

Returns None

clear_summaries()

Clear the summary statistics stored in the object.

copy()

Return a deep copy of this distribution object.

This object will not be tied to any other distribution or connected in any form.

Parameters **None**

Returns **distribution** : Distribution

A copy of the distribution with the same parameters.

dense_transition_matrix()
Returns the dense transition matrix. Useful if the transitions of somewhat small models need to be analyzed.

edge_count()
Returns the number of edges present in the model.

fit()
Fit the model to data using MLE estimates.

Fit the model to the data by updating each of the components of the model, which are univariate or multivariate distributions. This uses a simple MLE estimate to update the distributions according to their summarize or fit methods.

This is a wrapper for the summarize and from_summaries methods.

Parameters **items** : array-like, shape (n_samples, n_nodes)

The data to train on, where each row is a sample and each column corresponds to the associated variable.

weights : array-like, shape (n_nodes), optional

The weight of each sample as a positive double. Default is None.

inertia : double, optional

The inertia for updating the distributions, passed along to the distribution method. Default is 0.0.

Returns None

freeze()
Freeze the distribution, preventing updates from occuring.

from_json()
Read in a serialized Bayesian Network and return the appropriate object.

Parameters **s** : str

A JSON formatted string containing the file.

Returns **model** : object

A properly initialized and baked model.

from_samples()
Learn the structure of the network from data.

Find the structure of the network from data using a Bayesian structure learning score. This currently enumerates all the exponential number of structures and finds the best according to the score. This allows weights on the different samples as well.

Parameters **X** : array-like, shape (n_samples, n_nodes)

The data to fit the structure too, where each row is a sample and each column corresponds to the associated variable.

weights : array-like, shape (n_nodes), optional

The weight of each sample as a positive double. Default is None.

algorithm : str, one of 'chow-liu', 'exact' optional

The algorithm to use for learning the Bayesian network. Default is 'chow-liu' which returns a tree structure.

max_parents : int, optional

The maximum number of parents a node can have. If used, this means using the k-learn procedure. Can drastically speed up algorithms. If -1, no max on parents. Default is -1.

root : int, optional

For algorithms which require a single root ('chow-liu'), this is the root for which all edges point away from. User may specify which column to use as the root. Default is the first column.

constraint_graph : networkx.DiGraph or None, optional

A directed graph showing valid parent sets for each variable. Each node is a set of variables, and edges represent which variables can be valid parents of those variables. The naive structure learning task is just all variables in a single node with a self edge, meaning that you know nothing about

pseudocount : double, optional

A pseudocount to add to each possibility.

Returns model : BayesianNetwork

The learned BayesianNetwork.

from_structure()

Return a Bayesian network from a predefined structure.

Pass in the structure of the network as a tuple of tuples and get a fit network in return. The tuple should contain n tuples, with one for each node in the graph. Each inner tuple should be of the parents for that node. For example, a three node graph where both node 0 and 1 have node 2 as a parent would be specified as ((2,), (2,), ()).

Parameters X : array-like, shape (n_samples, n_nodes)

The data to fit the structure too, where each row is a sample and each column corresponds to the associated variable.

structure : tuple of tuples

The parents for each node in the graph. If a node has no parents, then do not specify any parents.

weights : array-like, shape (n_nodes), optional

The weight of each sample as a positive double. Default is None.

name : str, optional

The name of the model. Default is None.

Returns model : BayesianNetwoork

A Bayesian network with the specified structure.

from_summaries()

Use MLE on the stored sufficient statistics to train the model.

This uses MLE estimates on the stored sufficient statistics to train the model.

Parameters inertia : double, optional

The inertia for updating the distributions, passed along to the distribution method. Default is 0.0.

Returns None

log_probability()
Return the log probability of a sample under the Bayesian network model.

The log probability is just the sum of the log probabilities under each of the components. The log probability of a sample under the graph A -> B is just P(A)*P(B|A).

Parameters **sample** : array-like, shape (n_nodes)

The sample is a vector of points where each dimension represents the same variable as added to the graph originally. It doesn't matter what the connections between these variables are, just that they are all ordered the same.

Returns **logp** : double

The log probability of that sample.

marginal()
Return the marginal probabilities of each variable in the graph.

This is equivalent to a pass of belief propogation on a graph where no data has been given. This will calculate the probability of each variable being in each possible emission when nothing is known.

Parameters **None**

Returns **marginals** : array-like, shape (n_nodes)

An array of univariate distribution objects showing the marginal probabilities of that variable.

node_count()
Returns the number of nodes/states in the model

plot()
Draw this model's graph using NetworkX and matplotlib.

Note that this relies on networkx's built-in graphing capabilities (and not Graphviz) and thus can't draw self-loops.

See networkx.draw_networkx() for the keywords you can pass in.

Parameters ****kwargs** : any

The arguments to pass into networkx.draw_networkx()

Returns None

predict()
Predict missing values of a data matrix using MLE.

Impute the missing values of a data matrix using the maximally likely predictions according to the forward-backward algorithm. Run each sample through the algorithm (predict_proba) and replace missing values with the maximally likely predicted emission.

Parameters **items** : array-like, shape (n_samples, n_nodes)

Data matrix to impute. Missing values must be either None (if lists) or np.nan (if numpy.ndarray). Will fill in these values with the maximally likely ones.

max_iterations : int, optional

Number of iterations to run loopy belief propogation for. Default is 100.

Returns **items** : array-like, shape (n_samples, n_nodes)

This is the data matrix with the missing values imputed.

predict_proba()

Returns the probabilities of each variable in the graph given evidence.

This calculates the marginal probability distributions for each state given the evidence provided through loopy belief propogation. Loopy belief propogation is an approximate algorithm which is exact for certain graph structures.

Parameters **data** : dict or array-like, shape <= n_nodes, optional

The evidence supplied to the graph. This can either be a dictionary with keys being state names and values being the observed values (either the emissions or a distribution over the emissions) or an array with the values being ordered according to the nodes incorporation in the graph (the order fed into .add_states/add_nodes) and None for variables which are unknown. If nothing is fed in then calculate the marginal of the graph. Default is {}.

max_iterations : int, optional

The number of iterations with which to do loopy belief propogation. Usually requires only 1. Default is 100.

check_input : bool, optional

Check to make sure that the observed symbol is a valid symbol for that distribution to produce. Default is True.

Returns **probabilities** : array-like, shape (n_nodes)

An array of univariate distribution objects showing the probabilities of each variable.

probability()

Return the probability of the given symbol under this distribution.

Parameters **symbol** : object

The symbol to calculate the probability of

Returns **probability** : double

The probability of that point under the distribution.

sample()

Return a random item sampled from this distribution.

Parameters **n** : int or None, optional

The number of samples to return. Default is None, which is to generate a single sample.

Returns **sample** : double or object

Returns a sample from the distribution of a type in the support of the distribution.

state_count()

Returns the number of states present in the model.

summarize()

Summarize a batch of data and store the sufficient statistics.

This will partition the dataset into columns which belong to their appropriate distribution. If the distribution has parents, then multiple columns are sent to the distribution. This relies mostly on the summarize function of the underlying distribution.

Parameters **items** : array-like, shape (n_samples, n_nodes)

The data to train on, where each row is a sample and each column corresponds to the associated variable.

weights : array-like, shape (n_nodes), optional

The weight of each sample as a positive double. Default is None.

Returns None

thaw()
Thaw the distribution, re-allowing updates to occur.

to_json()
Serialize the model to a JSON.

Parameters **separators** : tuple, optional

The two separaters to pass to the json.dumps function for formatting.

indent : int, optional

The indentation to use at each level. Passed to json.dumps for formatting.

Returns **json** : str

A properly formatted JSON object.

1.9 Factor Graphs

1.9.1 API Reference

class pomegranate.FactorGraph.**FactorGraph**
A Factor Graph model.

A biparte graph where conditional probability tables are on one side, and marginals for each of the variables involved are on the other side.

Parameters **name** : str, optional

The name of the model. Default is None.

bake()
Finalize the topology of the model.

Assign a numerical index to every state and create the underlying arrays corresponding to the states and edges between the states. This method must be called before any of the probability-calculating methods. This is the same as the HMM bake, except that at the end it sets current state information.

Parameters **None**

Returns None

marginal()
Return the marginal probabilities of each variable in the graph.

This is equivalent to a pass of belief propogation on a graph where no data has been given. This will calculate the probability of each variable being in each possible emission when nothing is known.

Parameters **None**

Returns **marginals** : array-like, shape (n_nodes)

An array of univariate distribution objects showing the marginal probabilities of that variable.

plot()

Draw this model's graph using NetworkX and matplotlib.

Note that this relies on networkx's built-in graphing capabilities (and not Graphviz) and thus can't draw self-loops.

See networkx.draw_networkx() for the keywords you can pass in.

Parameters ****kwargs** : any

The arguments to pass into networkx.draw_networkx()

Returns None

predict_proba()

Returns the probabilities of each variable in the graph given evidence.

This calculates the marginal probability distributions for each state given the evidence provided through loopy belief propogation. Loopy belief propogation is an approximate algorithm which is exact for certain graph structures.

Parameters **data** : dict or array-like, shape <= n_nodes, optional

The evidence supplied to the graph. This can either be a dictionary with keys being state names and values being the observed values (either the emissions or a distribution over the emissions) or an array with the values being ordered according to the nodes incorporation in the graph (the order fed into .add_states/add_nodes) and None for variables which are unknown. If nothing is fed in then calculate the marginal of the graph.

max_iterations : int, optional

The number of iterations with which to do loopy belief propogation. Usually requires only 1.

check_input : bool, optional

Check to make sure that the observed symbol is a valid symbol for that distribution to produce.

Returns **probabilities** : array-like, shape (n_nodes)

An array of univariate distribution objects showing the probabilities of each variable.

p

A

B

C

D

E

F

G

H

L

M

N

P

S

T

V