pymc-learn Documentation

Release 0.0.1.rc0

pymc-learn Developers Team

Nov 05, 2018

Getting Started

Contents:

CHAPTER 1

What is pymc-learn?

pymc-learn is a library for practical probabilistic machine learning in Python.

It provides a variety of state-of-the art probabilistic models for supervised and unsupervised machine learning. **It is inspired by** scikit-learn **and focuses on bringing probabilistic machine learning to non-specialists**. It uses a syntax that mimics scikit-learn. Emphasis is put on ease of use, productivity, flexibility, performance, documentation, and an API consistent with scikit-learn. It depends on scikit-learn and PyMC3 and is distributed under the new BSD-3 license, encouraging its use in both academia and industry.

Users can now have calibrated quantities of uncertainty in their models using powerful inference algorithms – such as MCMC or Variational inference – provided by PyMC3. See *Why pymc-learn?* for a more detailed description of why `pymc-learn` was created.

Note: `pymc-learn` leverages and extends the Base template provided by the PyMC3 Models project: https://github.com/parsing-science/pymc3_models

1.1 Transitioning from PyMC3 to PyMC4

CHAPTER 2

Familiar user interface

`pymc-learn` mimics scikit-learn. You don't have to completely rewrite your scikit-learn ML code.

```
from sklearn.linear_model \
  import LinearRegression
lr = LinearRegression()
lr.fit(X, y)
```

```
from pmlearn.linear_model \
  import LinearRegression
lr = LinearRegression()
lr.fit(X, y)
```

The difference between the two models is that `pymc-learn` estimates model parameters using Bayesian inference algorithms such as MCMC or variational inference. This produces calibrated quantities of uncertainty for model parameters and predictions.

CHAPTER 3

Quick Install

You can install `pymc-learn` from PyPi using pip as follows:

```
pip install pymc-learn
```

Or from source as follows:

```
pip install git+https://github.com/pymc-learn/pymc-learn
```

Caution: `pymc-learn` is under heavy development.

3.1 Dependencies

`pymc-learn` is tested on Python 2.7, 3.5 & 3.6 and depends on Theano, PyMC3, Scikit-learn, NumPy, SciPy, and Matplotlib (see `requirements.txt` for version information).

CHAPTER 4

Quick Start

```
# For regression using Bayesian Nonparametrics
>>> from sklearn.datasets import make_friedman2
>>> from pmlearn.gaussian_process import GaussianProcessRegressor
>>> from pmlearn.gaussian_process.kernels import DotProduct, WhiteKernel
>>> X, y = make_friedman2(n_samples=500, noise=0, random_state=0)
>>> kernel = DotProduct() + WhiteKernel()
>>> gpr = GaussianProcessRegressor(kernel=kernel).fit(X, y)
>>> gpr.score(X, y)
0.3680...
>>> gpr.predict(X[:2,:], return_std=True)
(array([653.0..., 592.1...]), array([316.6..., 316.6...]))
```

CHAPTER 5

Scales to Big Data & Complex Models

Recent research has led to the development of variational inference algorithms that are fast and almost as flexible as MCMC. For instance Automatic Differentation Variational Inference (ADVI) is illustrated in the code below.

```
from pmlearn.neural_network import MLPClassifier
model = MLPClassifier()
model.fit(X_train, y_train, inference_type="advi")
```

Instead of drawing samples from the posterior, these algorithms fit a distribution (e.g. normal) to the posterior turning a sampling problem into an optimization problem. ADVI is provided PyMC3.

CHAPTER 6

Citing pymc-learn

To cite `pymc-learn` in publications, please use the following:

```
Emaasit, Daniel (2018). Pymc-learn: Practical probabilistic machine
learning in Python. arXiv preprint arXiv:1811.00542.
```

Or using BibTex as follows:

```
@article{emaasit2018pymc,
  title={Pymc-learn: Practical probabilistic machine learning in {P}ython},
  author={Emaasit, Daniel and others},
  journal={arXiv preprint arXiv:1811.00542},
  year={2018}
}
```

If you want to cite `pymc-learn` for its API, you may also want to consider this reference:

```
Carlson, Nicole (2018). Custom PyMC3 models built on top of the scikit-learn
API. https://github.com/parsing-science/pymc3_models
```

Or using BibTex as follows:

```
@article{Pymc3_models,
  title={pymc3_models: Custom PyMC3 models built on top of the scikit-learn API,
  author={Carlson, Nicole},
  journal={},
  url={https://github.com/parsing-science/pymc3_models}
  year={2018}
}
```

6.1 License

New BSD-3 license

CHAPTER 7

Index

Getting Started

- *Install pymc-learn*
- *Community*
- *Why pymc-learn?*

7.1 Install pymc-learn

`pymc-learn` requires a working Python interpreter (2.7 or 3.3+). It is recommend installing Python and key numerical libraries using the Anaconda Distribution, which has one-click installers available on all major platforms.

Assuming a standard Python environment is installed on your machine (including pip), `pymc-learn` itself can be installed in one line using pip:

You can install `pymc-learn` from PyPi using pip as follows:

```
pip install pymc-learn
```

Or from source as follows:

```
pip install git+https://github.com/pymc-learn/pymc-learn
```

Caution: `pymc-learn` is under heavy development.

This also installs required dependencies including Theano. For alternative Theano installations (e.g., gpu), please see the instructions on the main Theano webpage.

7.1.1 Transitioning from PyMC3 to PyMC4

7.2 Community

`pymc-learn` is used and developed by individuals at some institutions.

7.2.1 Discussion

Conversation happens in the following places:

1. **Usage questions** are directed to Stack Overflow with the #pymc_learn tag. Pymc-learn developers monitor this tag and get e-mails whenever a question is asked.
2. **Bug reports and feature requests** are managed on the GitHub issue tracker

7.2.2 Asking for help

We welcome usage questions and bug reports from all users, even those who are new to using the project. There are a few things you can do to improve the likelihood of quickly getting a good answer.

1. **Ask questions in the right place**: We strongly prefer the use of StackOverflow or Github issues over Twitter. Github and StackOverflow are more easily searchable by future users and so is more efficient for everyone's time.

   If you have a general question about how something should work or want best practices then use Stack Overflow. If you think you have found a bug then use GitHub.
2. **Ask only in one place**: Please restrict yourself to posting your question in only one place (likely Stack Overflow or Github) and don't post in both.
3. **Create a minimal example**: It is ideal to create minimal, complete, verifiable examples. This significantly reduces the time that answerers spend understanding your situation and so results in higher quality answers more quickly.

   See also this blogpost about crafting minimal bug reports. These have a much higher likelihood of being answered.

7.3 Why pymc-learn?

There are several probabilistic machine learning frameworks available today. Why use `pymc-learn` rather than any other? Here are some of the reasons why you may be compelled to use `pymc-learn`.

7.3.1 pymc-learn prioritizes user experience

- *Familiarity*: `pymc-learn` mimics the syntax of scikit-learn – a popular Python library for machine learning – which has a consistent & simple API, and is very user friendly.
- *Ease of use*: This makes `pymc-learn` easy to learn and use for first-time users.
- *Productivity*: For scikit-learn users, you don't have to completely rewrite your code. Your code looks almost the same. You are more productive, allowing you to try more ideas faster.

```
from sklearn.linear_model \
  import LinearRegression
lr = LinearRegression()
lr.fit(X, y)
```

```
from pmlearn.linear_model \
  import LinearRegression
lr = LinearRegression()
lr.fit(X, y)
```

- *Flexibility*: This ease of use does not come at the cost of reduced flexibility. Given that `pymc-learn` integrates with PyMC3, it enables you to implement anything you could have built in the base language.
- *Performance*. The primary inference algorithm is gradient-based automatic differentiation variational inference (ADVI) (Kucukelbir et al., 2017), which estimates a divergence measure between approximate and true posterior distributions. Pymc-learn scales to complex, high-dimensional models thanks to GPU-accelerated tensor math and reverse-mode automatic differentiation via Theano (Theano Development Team, 2016), and it scales to large datasets thanks to estimates computed over mini-batches of data in ADVI.

7.3.2 Why do we need pymc-learn?

Currently, there is a growing need for principled machine learning approaches by non-specialists in many fields including the pure sciences (e.g. biology, physics, chemistry), the applied sciences (e.g. political science, biostatistics), engineering (e.g. transportation, mechanical), medicine (e.g. medical imaging), the arts (e.g visual art), and software industries.

This has lead to increased adoption of probabilistic modeling. This trend is attributed in part to three major factors:

1. the need for transparent models with calibrated quantities of uncertainty, i.e. "models should know when they don't know",
2. the ever-increasing number of promising results achieved on a variety of fundamental problems in AI (Ghahramani, 2015), and
3. the emergency of probabilistic programming languages (PPLs) that provide a fexible framework to build richly structured probabilistic models that incorporate domain knowledge.

However, usage of PPLs requires a specialized understanding of probability theory, probabilistic graphical modeling, and probabilistic inference. Some PPLs also require a good command of software coding. These requirements make it difficult for non-specialists to adopt and apply probabilistic machine learning to their domain problems.

`Pymc-learn` seeks to address these challenges by providing state-of-the art implementations of several popular probabilistic machine learning models. **It is inspired by scikit-learn** (Pedregosa et al., 2011) **and focuses on bringing probabilistic machine learning to non-specialists**. It puts emphasis on:

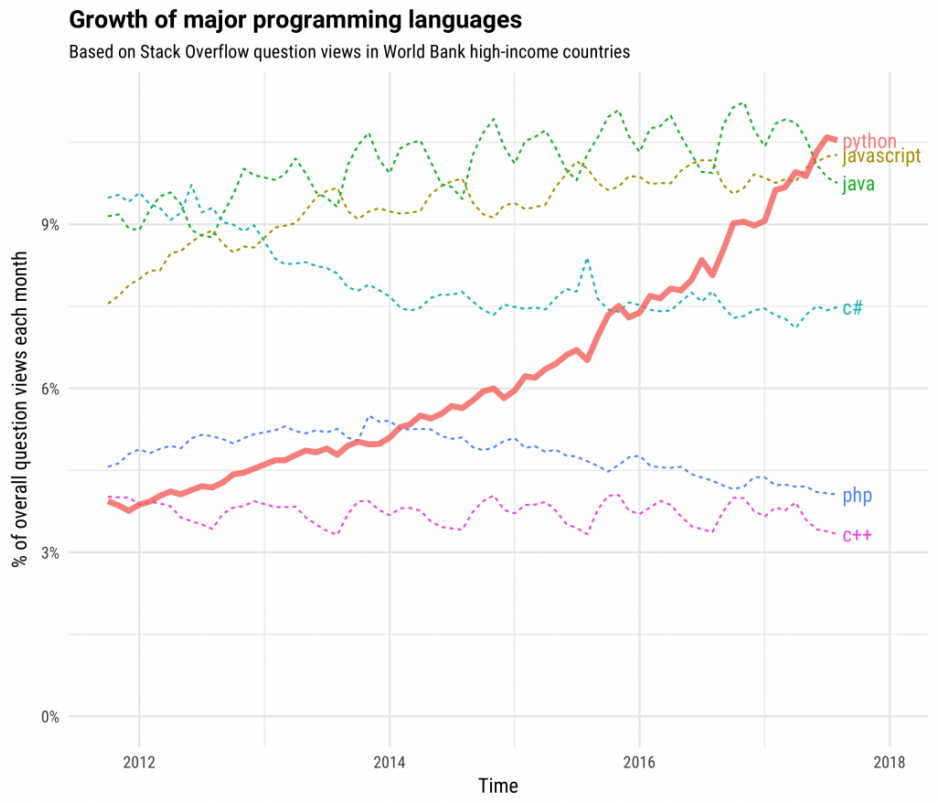
1. ease of use,
2. productivity,
3. fexibility,
4. performance,
5. documentation, and
6. an API consistent with scikit-learn.

The underlying probabilistic models are built using pymc3 (Salvatier et al., 2016).

Transitioning from PyMC3 to PyMC4

7.3.3 Python is the lingua franca of Data Science

Python has become the dominant language for both data science, and general programming:



This popularity is driven both by computational libraries like Numpy, Pandas, and Scikit-Learn and by a wealth of libraries for visualization, interactive notebooks, collaboration, and so forth.

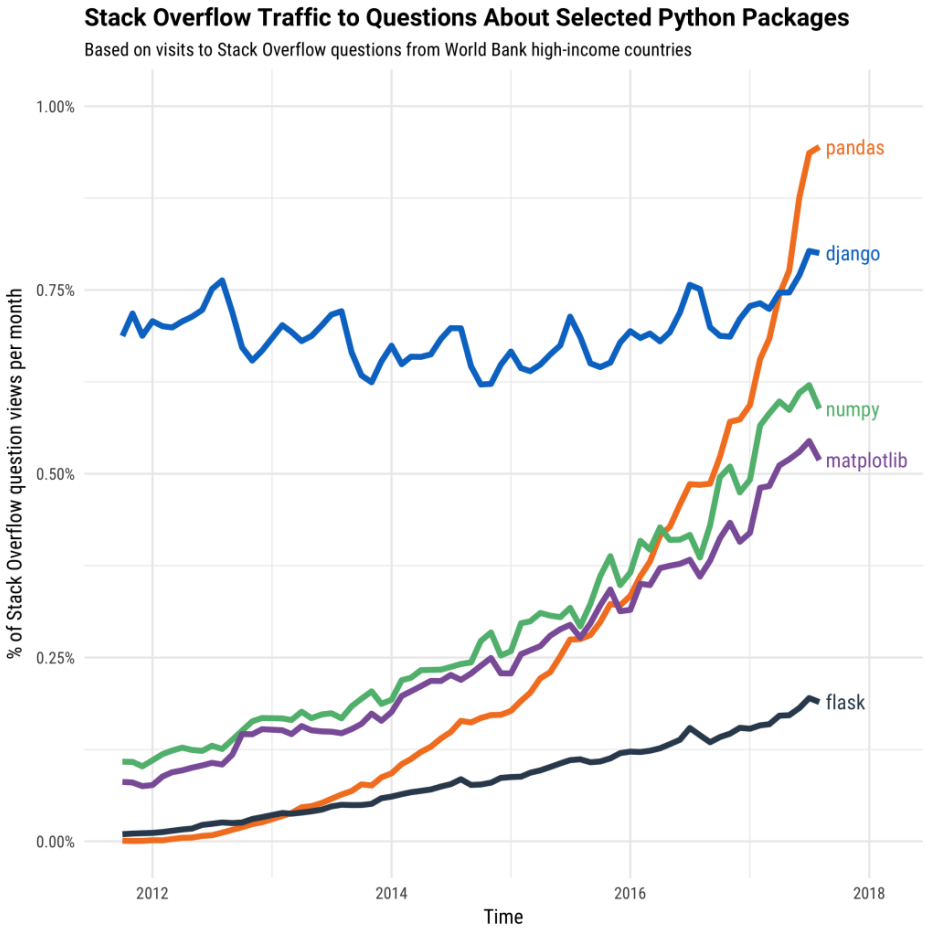


Image credit to Stack Overflow blogposts #1 and #2

7.3.4 Why scikit-learn and PyMC3

PyMC3 is a Python package for probabilistic machine learning that enables users to build bespoke models for their specific problems using a probabilistic modeling framework. However, PyMC3 lacks the steps between creating a model and reusing it with new data in production. The missing steps include: scoring a model, saving a model for later use, and loading the model in production systems.

In contrast, *scikit-learn* which has become the standard library for machine learning provides a simple API that makes it very easy for users to train, score, save and load models in production. However, *scikit-learn* may not have the model for a user's specific problem. These limitations have led to the development of the open source *pymc3-models* library which provides a template to build bespoke PyMC3 models on top of the *scikit-learn* API and reuse them in production. This enables users to easily and quickly train, score, save and load their bespoke models just like in *scikit-learn*.

The `pymc-learn` project adopted and extended the template in *pymc3-models* to develop probabilistic versions of the estimators in *scikit-learn*. This provides users with probabilistic models in a simple workflow that mimics the scikit-learn API.

7.3.5 Quantification of uncertainty

Today, many data-driven solutions are seeing a heavy use of machine learning for understanding phenomena and predictions. For instance, in cyber security, this may include monitoring streams of network data and predicting unusual events that deviate from the norm. For example, an employee downloading large volumes of intellectual property (IP) on a weekend. **Immediately**, we are faced with our first challenge, that is, we are dealing with quantities (unusual volume & unusual period) whose values are uncertain. To be more concrete, we start off very uncertain whether this download event is unusually large and then slowly get more and more certain as we uncover more clues such as the period of the week, performance reviews for the employee, or did they visit WikiLeaks?, etc.

In fact, the need to deal with uncertainty arises throughout our increasingly data-driven world. Whether it is Uber autonomous vehicles dealing with predicting pedestrians on roadways or Amazon's logistics apparatus that has to optimize its supply chain system. All these applications have to handle and manipulate uncertainty. Consequently, we need a principled framework for quantifying uncertainty which will allow us to create applications and build solutions in ways that can represent and process uncertain values.

Fortunately, there is a simple framework for manipulating uncertain quantities which uses probability to quantify the degree of uncertainty. To quote Prof. Zhoubin Ghahramani, Uber's Chief Scientist and Professor of AI at University of Cambridge:

> Just as Calculus is the fundamental mathematical principle for calculating rates of change, Probability is the fundamental mathematical principle for quantifying uncertainty.

The probabilistic approach to machine learning is an exciting area of research that is currently receiving a lot of attention in many conferences and Journals such as NIPS, UAI, AISTATS, JML, IEEE PAMI, etc.

7.3.6 References

1. Ghahramani, Z. (2015). Probabilistic machine learning and artificial intelligence. Nature, 521(7553), 452.
2. Bishop, C. M. (2013). Model-based machine learning. Phil. Trans. R. Soc. A, 371(1984), 20120222.
3. Murphy, K. P. (2012). Machine learning: a probabilistic perspective. MIT Press.
4. Barber, D. (2012). Bayesian reasoning and machine learning. Cambridge University Press.
5. Salvatier, J., Wiecki, T. V., & Fonnesbeck, C. (2016). Probabilistic programming in Python using PyMC3. PeerJ Computer Science, 2, e55.
6. Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M Blei. Automatic differentiation variational inference. The Journal of Machine Learning Research, 18(1):430{474, 2017.
7. Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. Journal of machine learning research, 12(Oct): 2825-2830, 2011.
8. Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. arXiv e-prints, abs/1605.02688, May 2016. URL http://arxiv.org/abs/1605.02688.

User Guide

The main documentation. This contains an in-depth description of all models and how to apply them.

- *User Guide*

7.4 User Guide

7.4.1 Supervised learning

Generalized Linear Models

The following are a set of methods intended for regression in which the target value is expected to be a linear combination of the input variables. In mathematical notion, if $\hat{y}$ is the predicted value.

$$\hat{y}(\beta, x) = \beta_0 + \beta_1 x_1 + ... + \beta_p x_p$$

Where $\beta = (\beta_1, ..., \beta_p)$ are the coefficients and β_0 is the y-intercept.

To perform classification with generalized linear models, see *Bayesian Logistic regression*.

Bayesian Linear Regression

To obtain a fully probabilistic model, the output y is assumed to be Gaussian distributed around Xw:

$$p(y|X, w, \alpha) = \mathcal{N}(y|Xw, \alpha)$$

Alpha is again treated as a random variable that is to be estimated from the data.

References

- A good introduction to Bayesian methods is given in C. Bishop: Pattern Recognition and Machine learning
- Original Algorithm is detailed in the book *Bayesian learning for neural networks* by Radford M. Neal

Bayesian Logistic regression

Bayesian Logistic regression, despite its name, is a linear model for classification rather than regression. Logistic regression is also known in the literature as logit regression, maximum-entropy classification (MaxEnt) or the log-linear classifier. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logistic function.

The implementation of logistic regression in pymc-learn can be accessed from class `LogisticRegression`.

Gaussian Processes

Gaussian Processes (GP) are a generic supervised learning method designed to solve *regression* and *probabilistic classification* problems.

Gaussian Process Regression (GPR)

The `GaussianProcessRegressor` implements Gaussian processes (GP) for regression purposes. For this, the prior of the GP needs to be specified. The prior mean is assumed to be constant and zero (for `normalize_y=False`) or the training data's mean (for `normalize_y=True`). The prior's covariance is specified by a passing a *kernel* object.

Kernels for Gaussian Processes

Kernels (also called "covariance functions" in the context of GPs) are a crucial ingredient of GPs which determine the shape of prior and posterior of the GP. They encode the assumptions on the function being learned by defining the "similarity" of two data points combined with the assumption that similar data points should have similar target values. Two categories of kernels can be distinguished: stationary kernels depend only on the distance of two data points and not on their absolute values $k(x_i, x_j) = k(d(x_i, x_j))$ and are thus invariant to translations in the input space, while non-stationary kernels depend also on the specific values of the data points. Stationary kernels can further be subdivided into isotropic and anisotropic kernels, where isotropic kernels are also invariant to rotations in the input space. For more details, we refer to Chapter 4 of *[RW2006]*.

References

Naive Bayes

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable. Bayes' theorem states the following relationship, given class variable y and dependent feature vector x_1 through x_n, :

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y)P(x_1, \ldots x_n \mid y)}{P(x_1, \ldots, x_n)}$$

Using the naive conditional independence assumption that

$$P(x_i|y, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = P(x_i|y),$$

for all i, this relationship is simplified to

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \ldots, x_n)}$$

Since $P(x_1, \ldots, x_n)$ is constant given the input, we can use the following classification rule:

$$P(y \mid x_1, \ldots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

$$\Downarrow$$

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y),$$

and we can use Maximum A Posteriori (MAP) estimation to estimate $P(y)$ and $P(x_i \mid y)$; the former is then the relative frequency of class y in the training set.

The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of $P(x_i \mid y)$.

In spite of their apparently over-simplified assumptions, naive Bayes classifiers have worked quite well in many real-world situations, famously document classification and spam filtering. They require a small amount of training data to estimate the necessary parameters. (For theoretical reasons why naive Bayes works well, and on which types of data it does, see the references below.)

Naive Bayes learners and classifiers can be extremely fast compared to more sophisticated methods. The decoupling of the class conditional feature distributions means that each distribution can be independently estimated as a one dimensional distribution. This in turn helps to alleviate problems stemming from the curse of dimensionality.

On the flip side, although naive Bayes is known as a decent classifier, it is known to be a bad estimator, so the probability outputs from `predict_proba` are not to be taken too seriously.

References:

- H. Zhang (2004). The optimality of Naive Bayes. Proc. FLAIRS.

Gaussian Naive Bayes

GaussianNB implements the Gaussian Naive Bayes algorithm for classification. The likelihood of the features is assumed to be Gaussian:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

The parameters σ_y and μ_y are estimated using maximum likelihood.

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> from pmlearn.naive_bayes import GaussianNB
>>> gnb = GaussianNB()
>>> y_pred = gnb.fit(iris.data, iris.target).predict(iris.data)
>>> print("Number of mislabeled points out of a total %d points : %d"
...       % (iris.data.shape[0],(iris.target != y_pred).sum()))
Number of mislabeled points out of a total 150 points : 6
```

Neural network models (supervised)

Warning: This implementation is not intended for large-scale applications. In particular, scikit-learn offers no GPU support. For much faster, GPU-based implementations, as well as frameworks offering much more flexibility to build deep learning architectures, see related_projects.

Multi-layer Perceptron

Multi-layer Perceptron (MLP) is a supervised learning algorithm that learns a function $f(\cdot) : R^m \rightarrow R^o$ by training on a dataset, where m is the number of dimensions for input and o is the number of dimensions for output. Given a set of features $X = x_1, x_2, ..., x_m$ and a target y, it can learn a non-linear function approximator for either classification or regression.

Classification

Class *MLPClassifier* implements a multi-layer perceptron (MLP) algorithm that trains using Backpropagation.

MLP trains on two arrays: array X of size (n_samples, n_features), which holds the training samples represented as floating point feature vectors; and array y of size (n_samples,), which holds the target values (class labels) for the training samples:

```
>>> from pmlearn.neural_network import MLPClassifier
>>> X = [[0., 0.], [1., 1.]]
>>> y = [0, 1]
>>> clf = MLPClassifier()
```

(continues on next page)

(continued from previous page)

```
...
>>> clf.fit(X, y)
>>> clf.predict([[2., 2.], [-1., -2.]])
array([1, 0])
```

References:

- "Learning representations by back-propagating errors." Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams.
- "Stochastic Gradient Descent" L. Bottou - Website, 2010.
- "Backpropagation" Andrew Ng, Jiquan Ngiam, Chuan Yu Foo, Yifan Mai, Caroline Suen - Website, 2011.
- "Efficient BackProp" Y. LeCun, L. Bottou, G. Orr, K. Müller - In Neural Networks: Tricks of the Trade 1998.
- "Adam: A method for stochastic optimization." Kingma, Diederik, and Jimmy Ba. arXiv preprint arXiv:1412.6980 (2014).

7.4.2 Unsupervised learning

Gaussian mixture models

`pmlearn.mixture` is a package which enables one to learn Gaussian Mixture Models.

A Gaussian mixture model is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters.

pymc-learn implements different classes to estimate Gaussian mixture models, that correspond to different estimation strategies, detailed below.

Gaussian Mixture

A `GaussianMixture.fit()` method is provided that learns a Gaussian Mixture Model from train data. Given test data, it can assign to each sample the Gaussian it mostly probably belong to using the `GaussianMixture.predict()` method.

The Dirichlet Process

Here we describe variational inference algorithms on Dirichlet process mixture. The Dirichlet process is a prior probability distribution on *clusterings with an infinite, unbounded, number of partitions*. Variational techniques let us incorporate this prior structure on Gaussian mixture models at almost no penalty in inference time, comparing with a finite Gaussian mixture model.

Examples

Pymc-learn provides probabilistic models for machine learning, in a familiar scikit-learn syntax.

- *Regression*
- *Classification*

- *Mixture Models*
- *Bayesian Neural Networks*
- *API*

7.5 Regression

7.5.1 Linear Regression

Let's set some setting for this Jupyter Notebook.

```
In [1]: %matplotlib inline
        from warnings import filterwarnings
        filterwarnings("ignore")
        import os
        os.environ['MKL_THREADING_LAYER'] = 'GNU'
        os.environ['THEANO_FLAGS'] = 'device=cpu'

        import numpy as np
        import pandas as pd
        import pymc3 as pm
        import seaborn as sns
        import matplotlib.pyplot as plt
        np.random.seed(12345)
        rc = {'xtick.labelsize': 20, 'ytick.labelsize': 20, 'axes.labelsize': 20, 'font.size': 20,
              'legend.fontsize': 12.0, 'axes.titlesize': 10, "figure.figsize": [12, 6]}
        sns.set(rc = rc)
        from IPython.core.interactiveshell import InteractiveShell
        InteractiveShell.ast_node_interactivity = "all"
```

Now, let's import the `LinearRegression` model from the `pymc-learn` package.

```
In [2]: import pmlearn
        from pmlearn.linear_model import LinearRegression
        print('Running on pymc-learn v{}'.format(pmlearn.__version__))
```

```
Running on pymc-learn v0.0.1.rc0
```

Step 1: Prepare the data

Generate synthetic data.

```
In [3]: X = np.random.randn(1000, 1)
        noise = 2 * np.random.randn(1000, 1)
        slope = 4
        intercept = 3
        y = slope * X + intercept + noise
        y = np.squeeze(y)

        fig, ax = plt.subplots()
        ax.scatter(X, y);
```


```
In [4]: from sklearn.model_selection import train_test_split
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

Step 2: Instantiate a model

```
In [5]: model = LinearRegression()
```

Step 3: Perform Inference

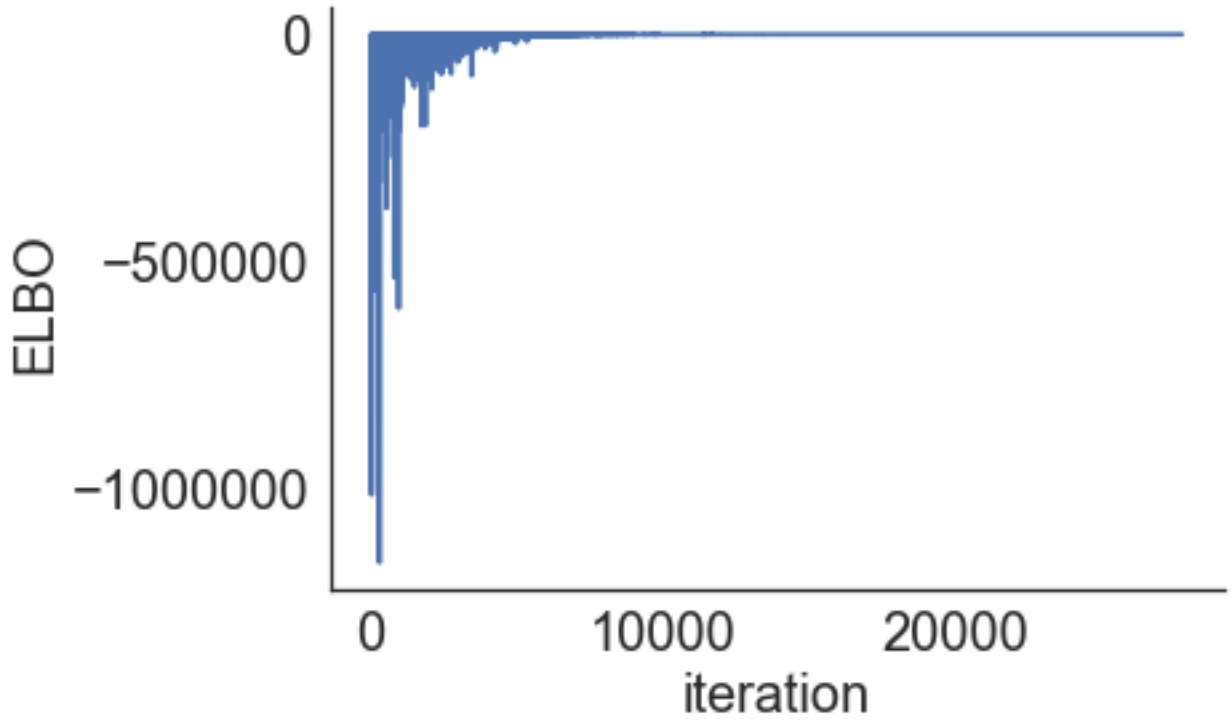
```
In [6]: model.fit(X_train, y_train)
```

```
Average Loss = 1,512:  14%|        | 27662/200000 [00:14<01:29, 1923.95it/s]
Convergence archived at 27700
Interrupted at 27,699 [13%]: Average Loss = 3,774.9
```

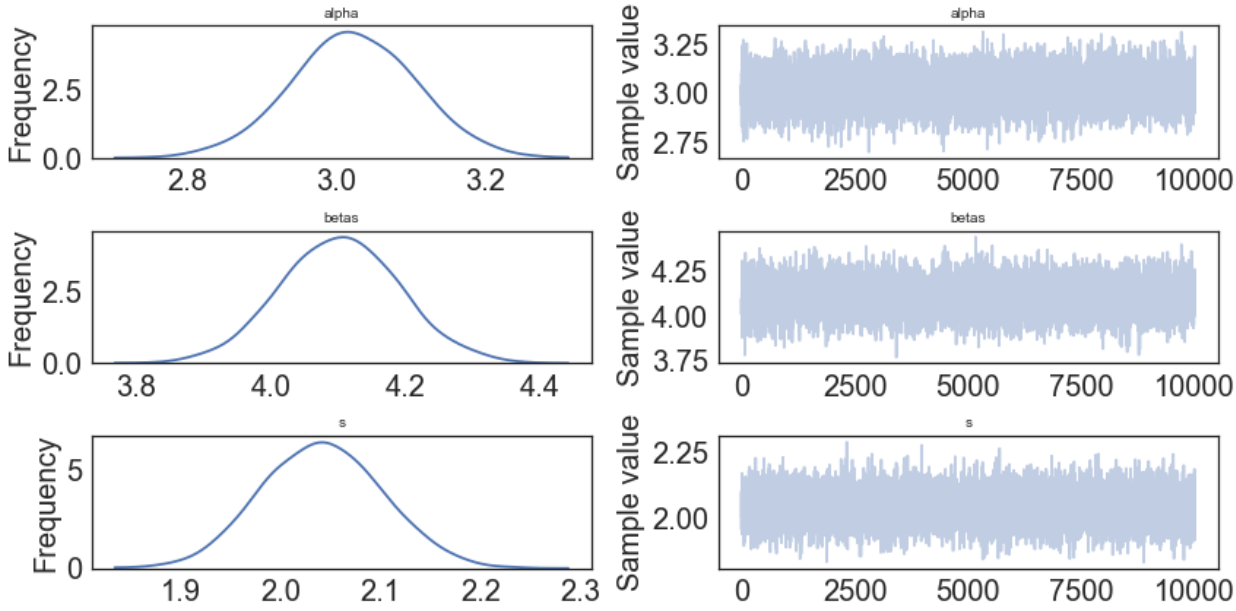
```
Out[6]: LinearRegression()
```

Step 4: Diagnose convergence

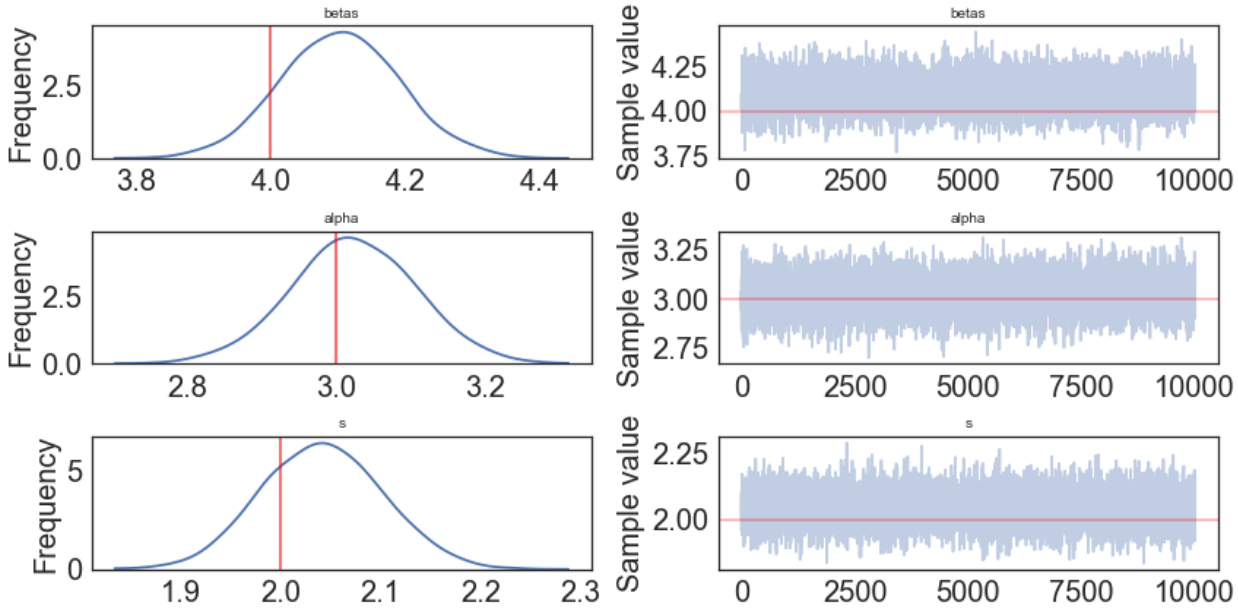
```
In [7]: model.plot_elbo()
```



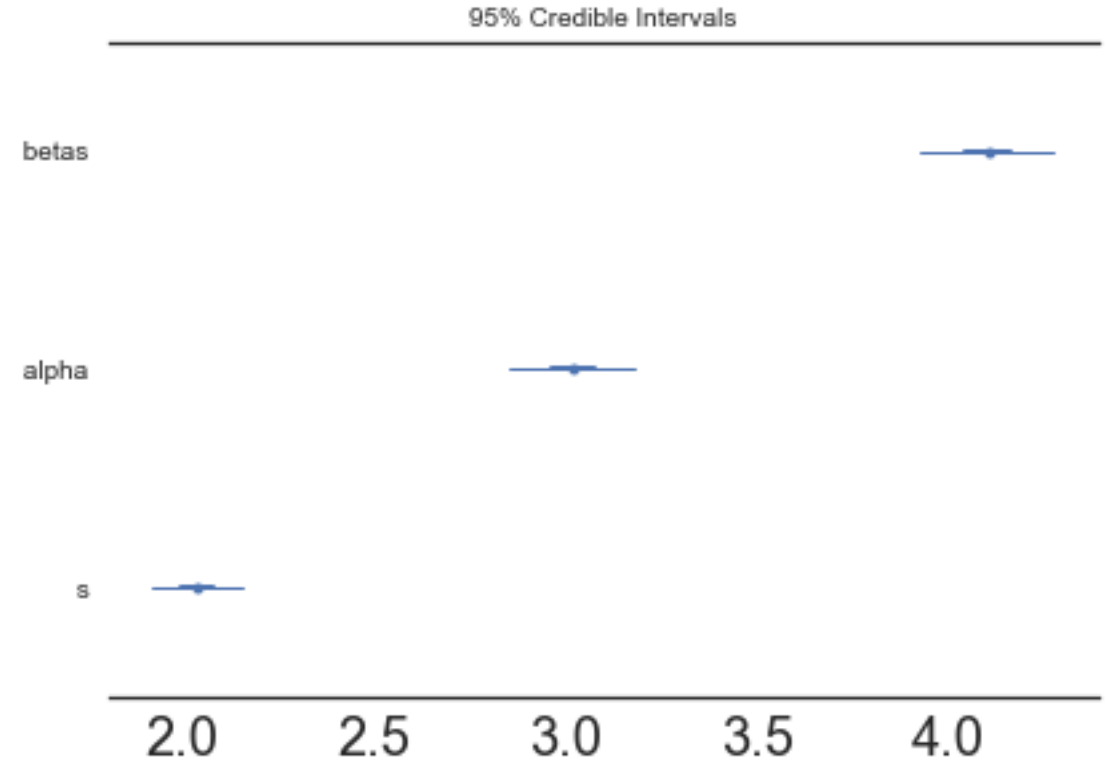
In [8]: pm.traceplot(model.trace);



In [9]: pm.traceplot(model.trace, lines = {"betas": slope,
"alpha": intercept,
"s": 2},
varnames=["betas", "alpha", "s"]);




```
In [10]: pm.forestplot(model.trace, varnames=["betas", "alpha", "s"]);
```



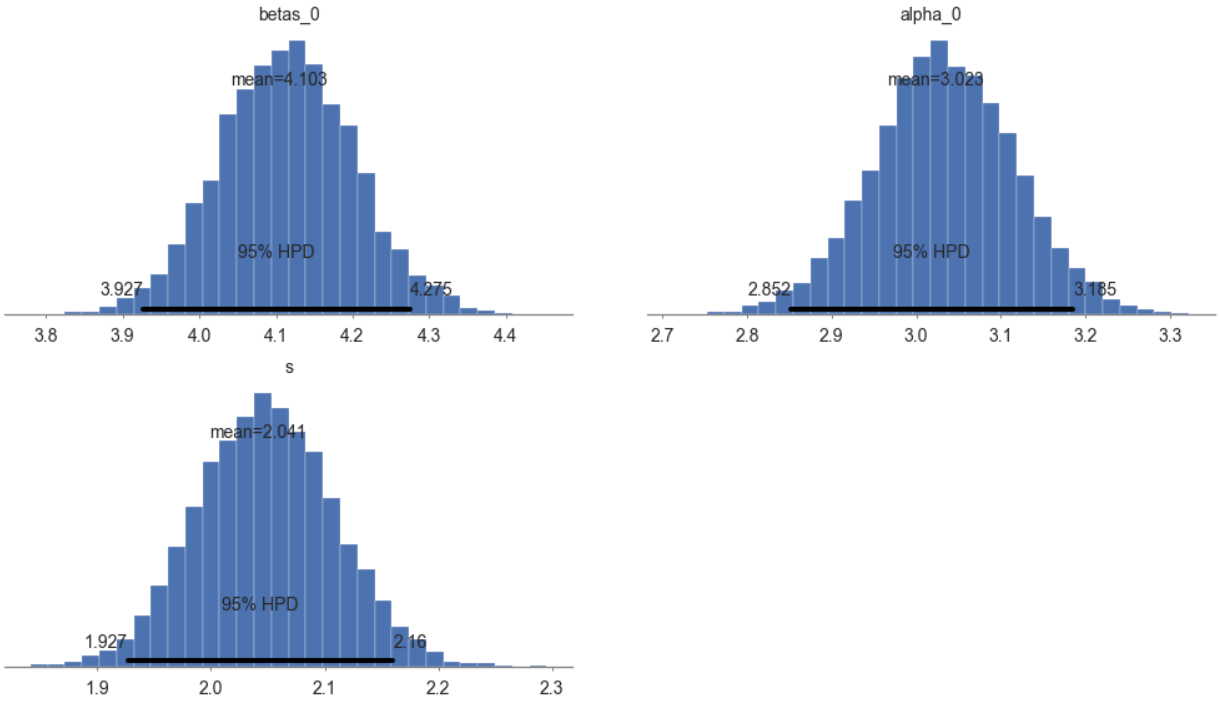
Step 5: Critize the model

```
In [11]: pm.summary(model.trace, varnames=["betas", "alpha", "s"])
```

```
Out[11]:                mean        sd  mc_error   hpd_2.5  hpd_97.5
         betas__0_0  4.103347  0.087682  0.000780  3.927048  4.274840
```

```
alpha__0    3.023007  0.084052  0.000876  2.851976  3.184953
s           2.041354  0.060310  0.000535  1.926967  2.160172
```

```
In [12]: pm.plot_posterior(model.trace, varnames=["betas", "alpha", "s"],
                           figsize = [14, 8]);
```




```
In [13]: # collect the results into a pandas dataframe to display
         # "mp" stands for marginal posterior
         pd.DataFrame({"Parameter": ["betas", "alpha", "s"],
                       "Parameter-Learned (Mean Value)": [float(model.trace["betas"].mean(axis=0)),
                                                          float(model.trace["alpha"].mean(axis=0)),
                                                          float(model.trace["s"].mean(axis=0))],
                       "True value": [slope, intercept, 2]})
```

```
Out[13]:   Parameter  Parameter-Learned (Mean Value)  True value
         0     betas                        4.103347           4
         1     alpha                        3.023007           3
         2         s                        2.041354           2
```

Step 6: Use the model for prediction

```
In [14]: y_predict = model.predict(X_test)
```

```
100%|| 2000/2000 [00:00<00:00, 2453.39it/s]
```

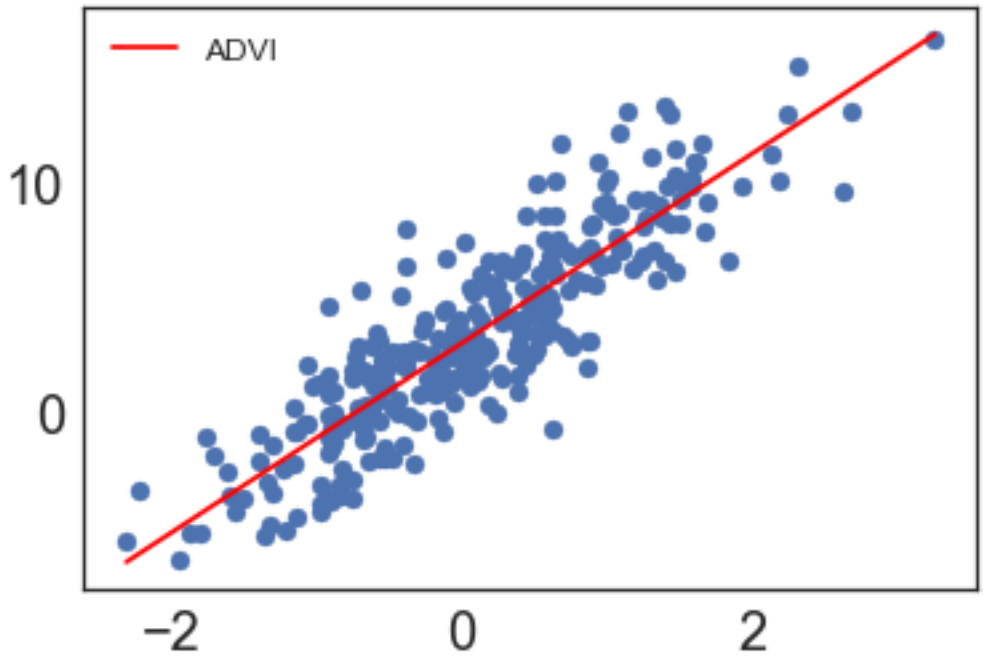
```
In [15]: model.score(X_test, y_test)
```

```
100%|| 2000/2000 [00:00<00:00, 2694.94it/s]
```

```
Out[15]: 0.77280797745735419
```

```
In [17]: max_x = max(X_test)
         min_x = min(X_test)

         slope_learned = model.summary['mean']['betas__0_0']
         intercept_learned = model.summary['mean']['alpha__0']
```

```
        fig1, ax1 = plt.subplots()
        ax1.scatter(X_test, y_test)
        ax1.plot([min_x, max_x], [slope_learned*min_x + intercept_learned, slope_learned*max_x + int
        ax1.legend();
```




```
In [18]: model.save('pickle_jar/linear_model')
```

Use already trained model for prediction

```
In [19]: model_new = LinearRegression()
```

```
In [20]: model_new.load('pickle_jar/linear_model')
```

```
In [21]: model_new.score(X_test, y_test)
```

```
100%|| 2000/2000 [00:00<00:00, 2407.83it/s]
```

```
Out[21]: 0.77374690833817472
```

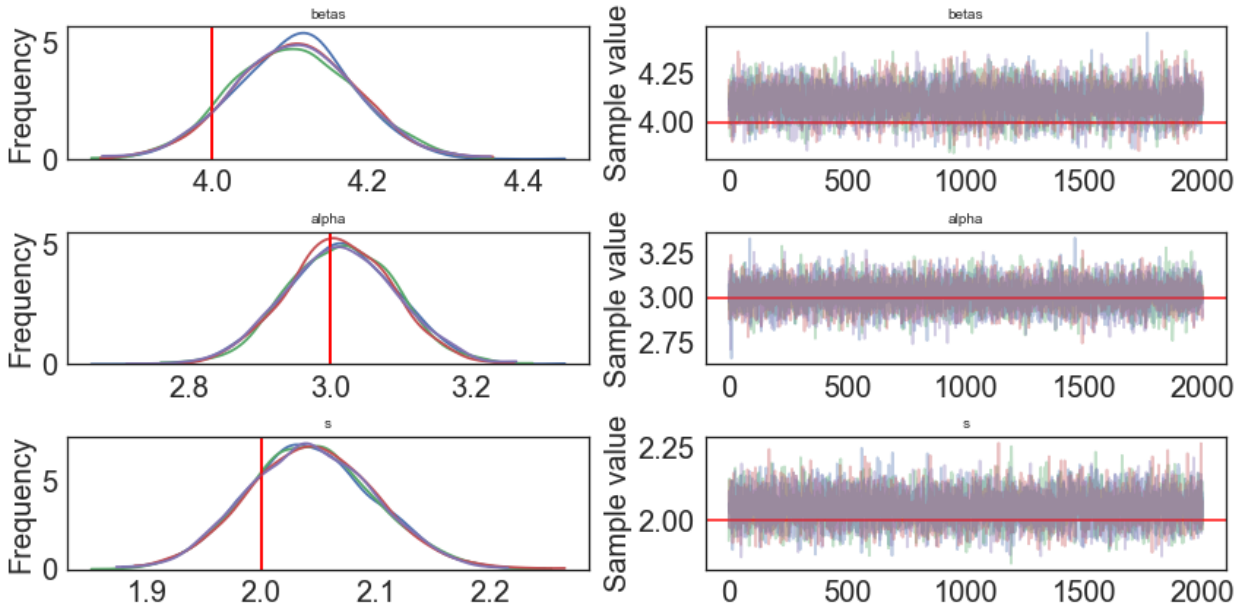
MCMC

```
In [22]: model2 = LinearRegression()
         model2.fit(X_train, y_train, inference_type='nuts')
```

```
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [s_log__, betas, alpha]
100%|| 2500/2500 [00:02<00:00, 1119.39it/s]
```

```
Out[22]: LinearRegression()
```

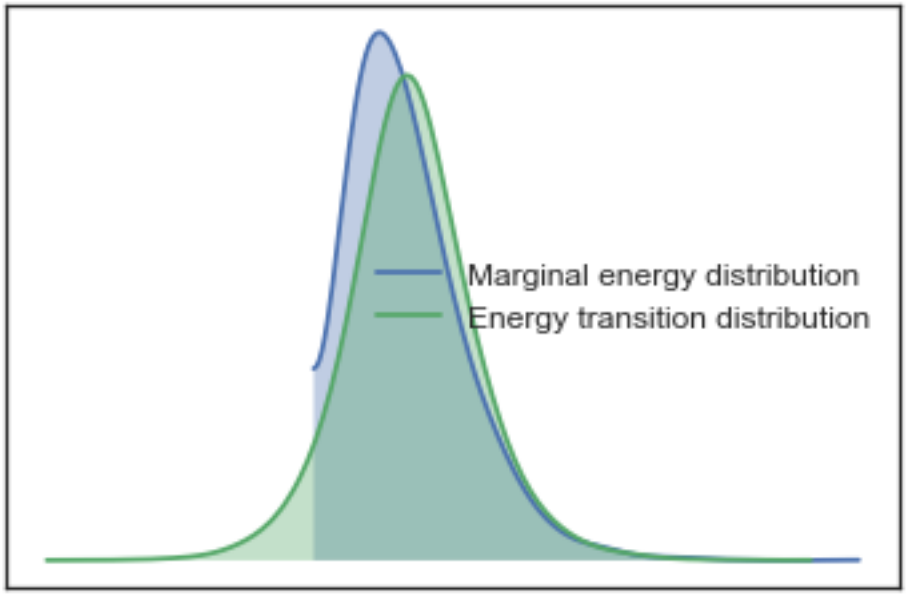
```
In [23]: pm.traceplot(model2.trace, lines = {"betas": slope,
                                              "alpha": intercept,
                                              "s": 2},
                      varnames=["betas", "alpha", "s"]);
```



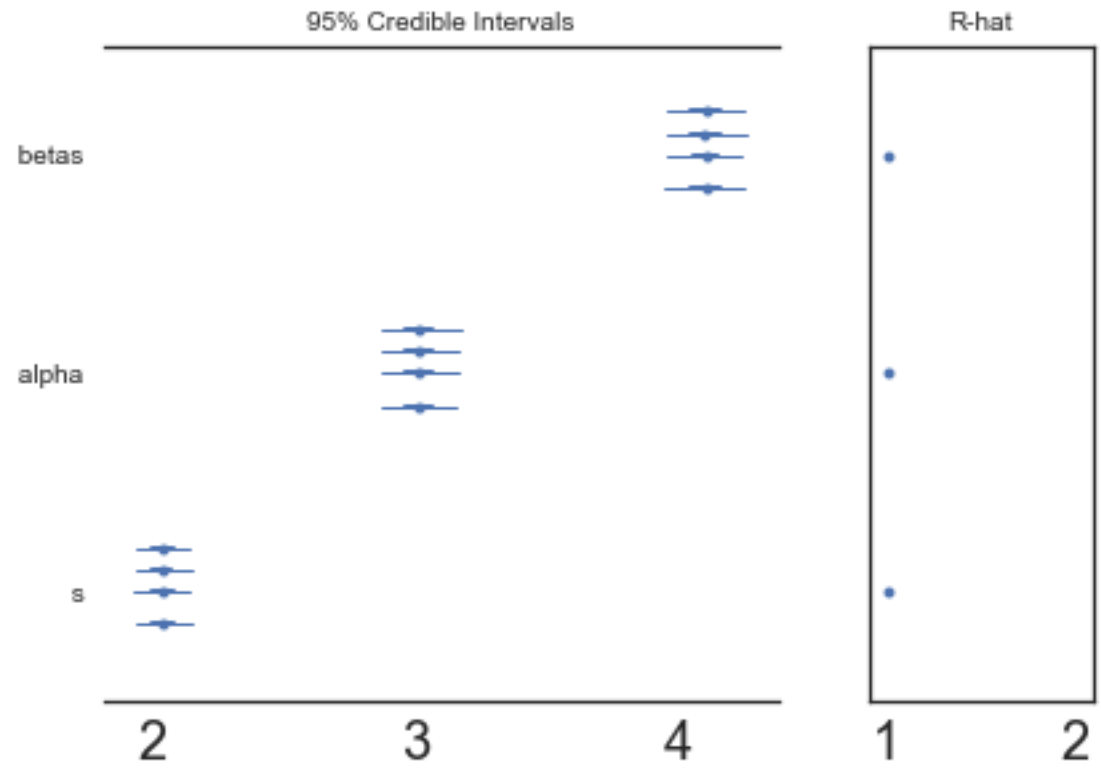

```
In [24]: pm.gelman_rubin(model2.trace, varnames=["betas", "alpha", "s"])
```

```
Out[24]: {'betas': array([[ 0.99983746]]),
          'alpha': array([ 0.99995935]),
          's': 0.99993398539611289}
```

```
In [25]: pm.energyplot(model2.trace);
```




```
In [26]: pm.forestplot(model2.trace, varnames=["betas", "alpha", "s"]);
```

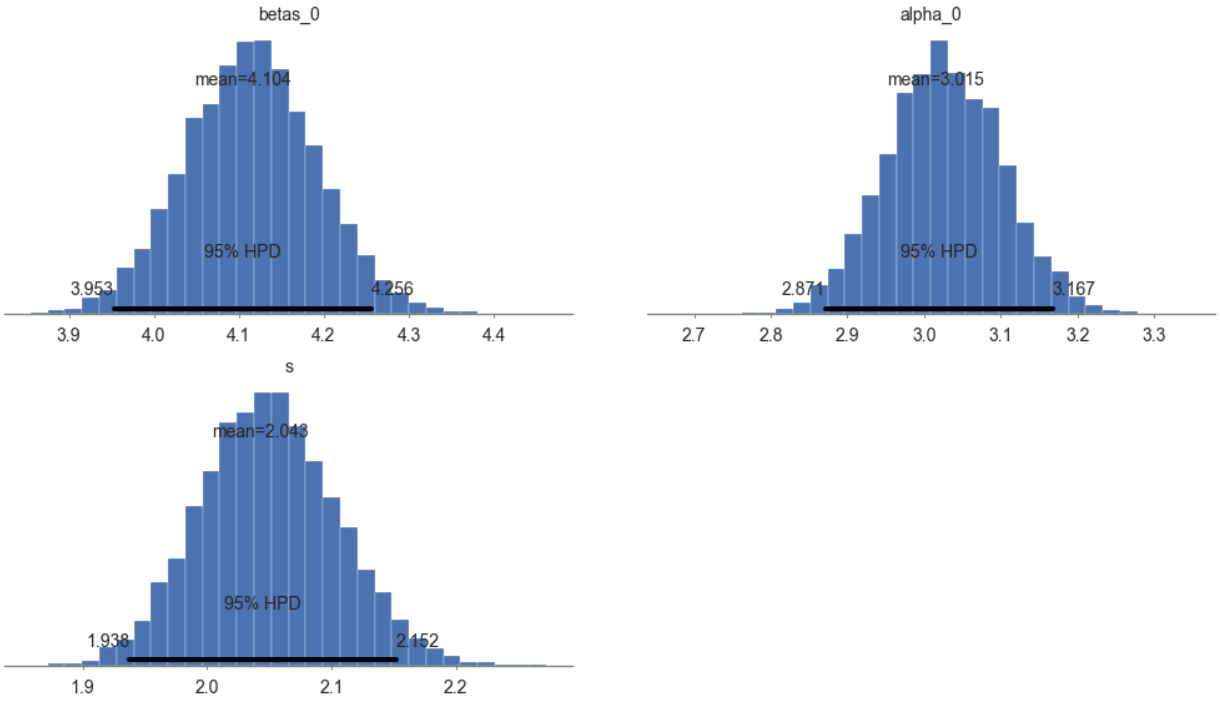



```
In [27]: pm.summary(model2.trace, varnames=["betas", "alpha", "s"])
```

Out[27]:

	mean	sd	mc_error	hpd_2.5	hpd_97.5	n_eff	Rhat
betas__0_0	4.104122	0.077358	0.000737	3.953261	4.255527	10544.412041	0.999837
alpha__0	3.014851	0.076018	0.000751	2.871447	3.167465	11005.925586	0.999959
s	2.043248	0.055253	0.000469	1.937634	2.152135	11918.920019	0.999934

```
In [28]: pm.plot_posterior(model2.trace, varnames=["betas", "alpha", "s"],
                          figsize = [14, 8]);
```




```
In [29]: y_predict2 = model2.predict(X_test)
```

```
100%|| 2000/2000 [00:00<00:00, 2430.09it/s]
```

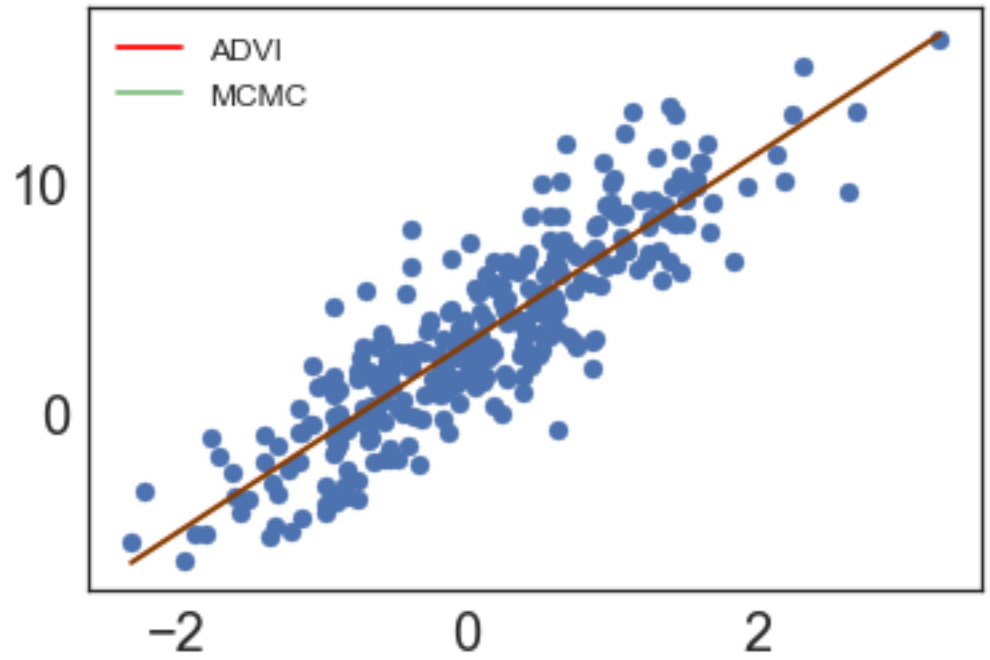
```
In [30]: model2.score(X_test, y_test)
```

```
100%|| 2000/2000 [00:00<00:00, 2517.14it/s]
```

```
Out[30]: 0.77203138547620576
```

Compare the two methods

```
In [31]: max_x = max(X_test)
         min_x = min(X_test)

         slope_learned = model.summary['mean']['betas__0_0']
         intercept_learned = model.summary['mean']['alpha__0']

         slope_learned2 = model2.summary['mean']['betas__0_0']
         intercept_learned2 = model2.summary['mean']['alpha__0']

         fig1, ax1 = plt.subplots()
         ax1.scatter(X_test, y_test)
         ax1.plot([min_x, max_x], [slope_learned*min_x + intercept_learned, slope_learned*max_x + int
         ax1.plot([min_x, max_x], [slope_learned2*min_x + intercept_learned2, slope_learned2*max_x +
         ax1.legend();
```




```
In [32]: model2.save('pickle_jar/linear_model2')
         model2_new = LinearRegression()
         model2_new.load('pickle_jar/linear_model2')
         model2_new.score(X_test, y_test)
```

```
100%|| 2000/2000 [00:00<00:00, 2510.66it/s]
```

```
Out[32]: 0.77268267975638316
```

Multiple predictors

```
In [33]: num_pred = 2
         X = np.random.randn(1000, num_pred)
         noise = 2 * np.random.randn(1000,)
         y = X.dot(np.array([4, 5])) + 3 + noise
         y = np.squeeze(y)
```

```
In [34]: model_big = LinearRegression()
```

```
In [35]: model_big.fit(X, y)
```

```
Average Loss = 2,101.7:  16%|        | 32715/200000 [00:16<01:27, 1920.43it/s]
Convergence archived at 32800
Interrupted at 32,799 [16%]: Average Loss = 7,323.3
```

```
Out[35]: LinearRegression()
```

```
In [36]: model_big.summary
```

```
Out[36]:                 mean        sd  mc_error   hpd_2.5  hpd_97.5
         alpha__0    2.907646  0.067021  0.000569  2.779519  3.038211
         betas__0_0  4.051214  0.066629  0.000735  3.921492  4.181938
         betas__0_1  5.038803  0.065906  0.000609  4.907503  5.166529
         s           1.929029  0.046299  0.000421  1.836476  2.016829
```

7.5.2 Logistic Regression

Let's set some setting for this Jupyter Notebook.

```
In [1]: %matplotlib inline
        from warnings import filterwarnings
        filterwarnings("ignore")
        import os
        os.environ['MKL_THREADING_LAYER'] = 'GNU'
        os.environ['THEANO_FLAGS'] = 'device=cpu'

        import numpy as np
        import pandas as pd
        import pymc3 as pm
        import seaborn as sns
        import matplotlib.pyplot as plt
        np.random.seed(12345)
        rc = {'xtick.labelsize': 20, 'ytick.labelsize': 20, 'axes.labelsize': 20, 'font.size': 20,
              'legend.fontsize': 12.0, 'axes.titlesize': 10, "figure.figsize": [12, 6]}
        sns.set(rc = rc)
        from IPython.core.interactiveshell import InteractiveShell
        InteractiveShell.ast_node_interactivity = "all"
```

Now, let's import the `LogisticRegression` model from the `pymc-learn` package.

```
In [2]: import pmlearn
        from pmlearn.linear_model import LogisticRegression
        print('Running on pymc-learn v{}'.format(pmlearn.__version__))
```

```
Running on pymc-learn v0.0.1.rc0
```

Step 1: Prepare the data

Generate synthetic data.

```
In [3]: num_pred = 2
        num_samples = 700000
        num_categories = 2
```

```
In [4]: alphas = 5 * np.random.randn(num_categories) + 5 # mu_alpha = sigma_alpha = 5
        betas = 10 * np.random.randn(num_categories, num_pred) + 10 # mu_beta = sigma_beta = 10
```

```
In [5]: alphas
```

```
Out[5]: array([ 3.9764617 ,  7.39471669])
```

```
In [6]: betas
```

```
Out[6]: array([[  4.80561285,   4.44269696],
               [ 29.65780573,  23.93405833]])
```

```
In [7]: def numpy_invlogit(x):
            return 1 / (1 + np.exp(-x))
```

```
In [8]: x_a = np.random.randn(num_samples, num_pred)
        y_a = np.random.binomial(1, numpy_invlogit(alphas[0] + np.sum(betas[0] * x_a, 1)))
        x_b = np.random.randn(num_samples, num_pred)
        y_b = np.random.binomial(1, numpy_invlogit(alphas[1] + np.sum(betas[1] * x_b, 1)))

        X = np.concatenate([x_a, x_b])
        y = np.concatenate([y_a, y_b])
        cats = np.concatenate([
            np.zeros(num_samples, dtype=np.int),
```

```
            np.ones(num_samples, dtype=np.int)
        ])
```

```
In [9]: from sklearn.model_selection import train_test_split
        X_train, X_test, y_train, y_test, cats_train, cats_test = train_test_split(X, y, cats, test_s
```

Step 2: Instantiate a model

```
In [10]: model = LogisticRegression()
```

Step 3: Perform Inference

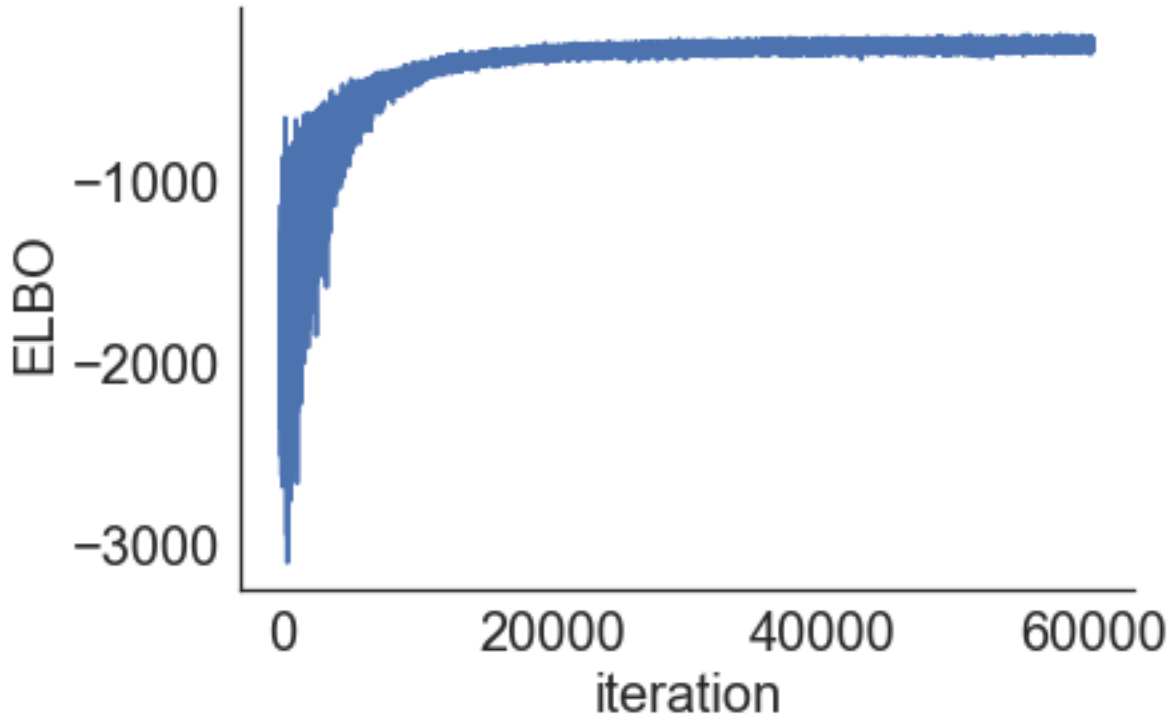
```
In [11]: model.fit(X_train, y_train, cats_train, minibatch_size=2000, inference_args={'n': 60000})
```

```
Average Loss = 249.45: 100%|| 60000/60000 [01:13<00:00, 814.48it/s]
Finished [100%]: Average Loss = 249.5
```

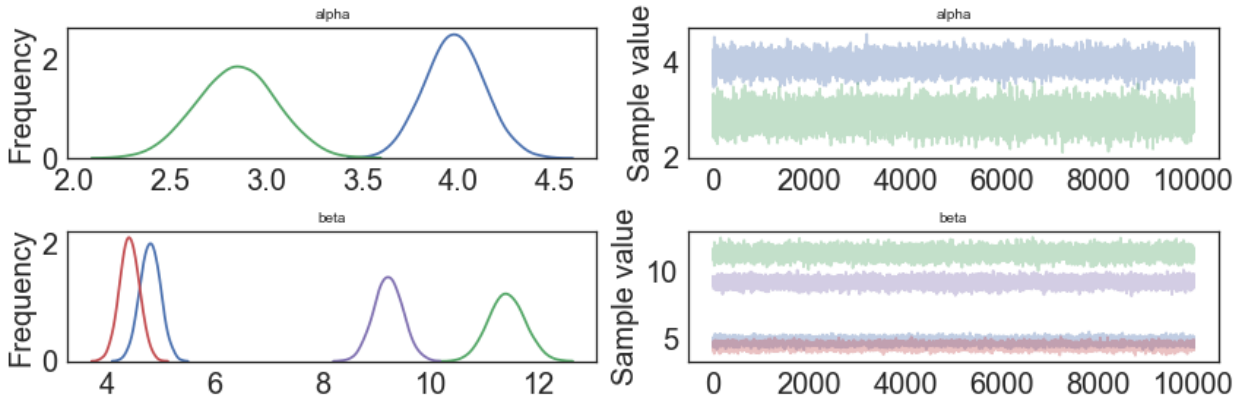
```
Out[11]: LogisticRegression()
```

Step 4: Diagnose convergence

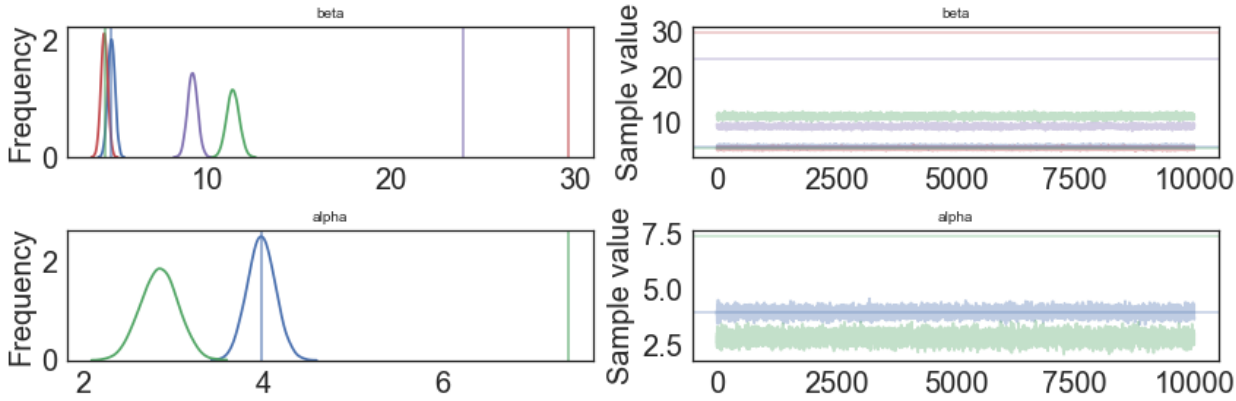
```
In [12]: model.plot_elbo()
```



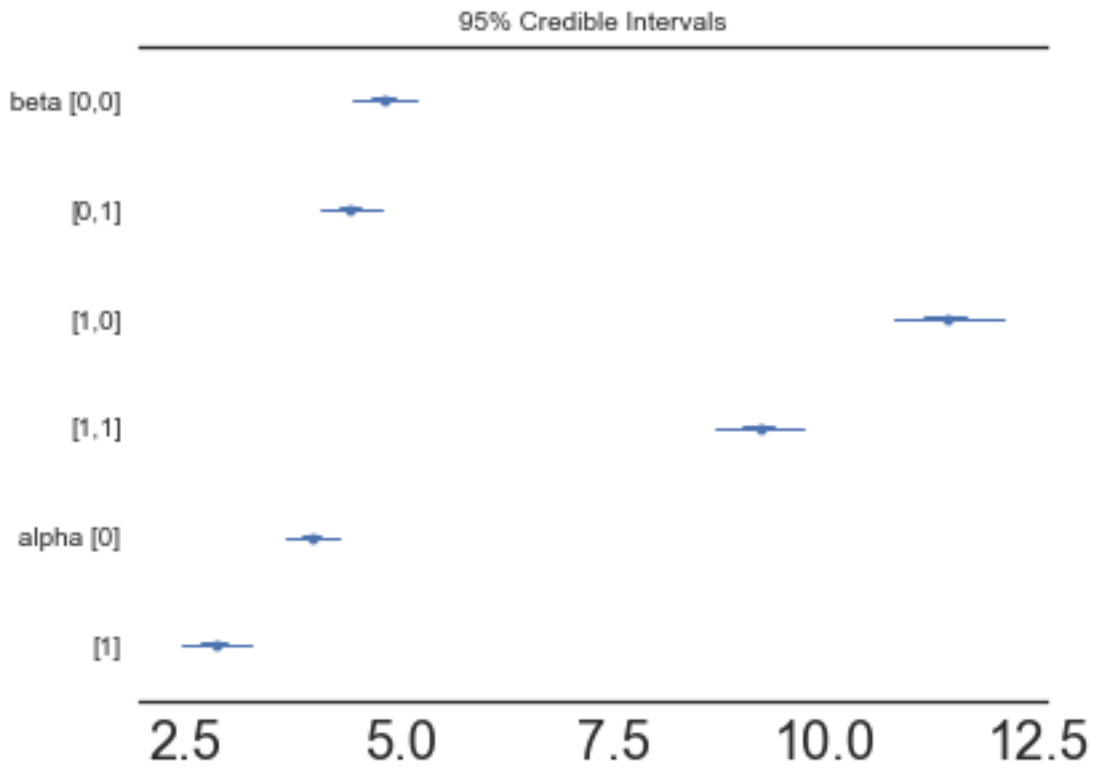

```
In [13]: pm.traceplot(model.trace);
```




```
In [14]: pm.traceplot(model.trace, lines = {"beta": betas,
                                            "alpha": alphas},
                      varnames=["beta", "alpha"]);
```




```
In [15]: pm.forestplot(model.trace, varnames=["beta", "alpha"]);
```



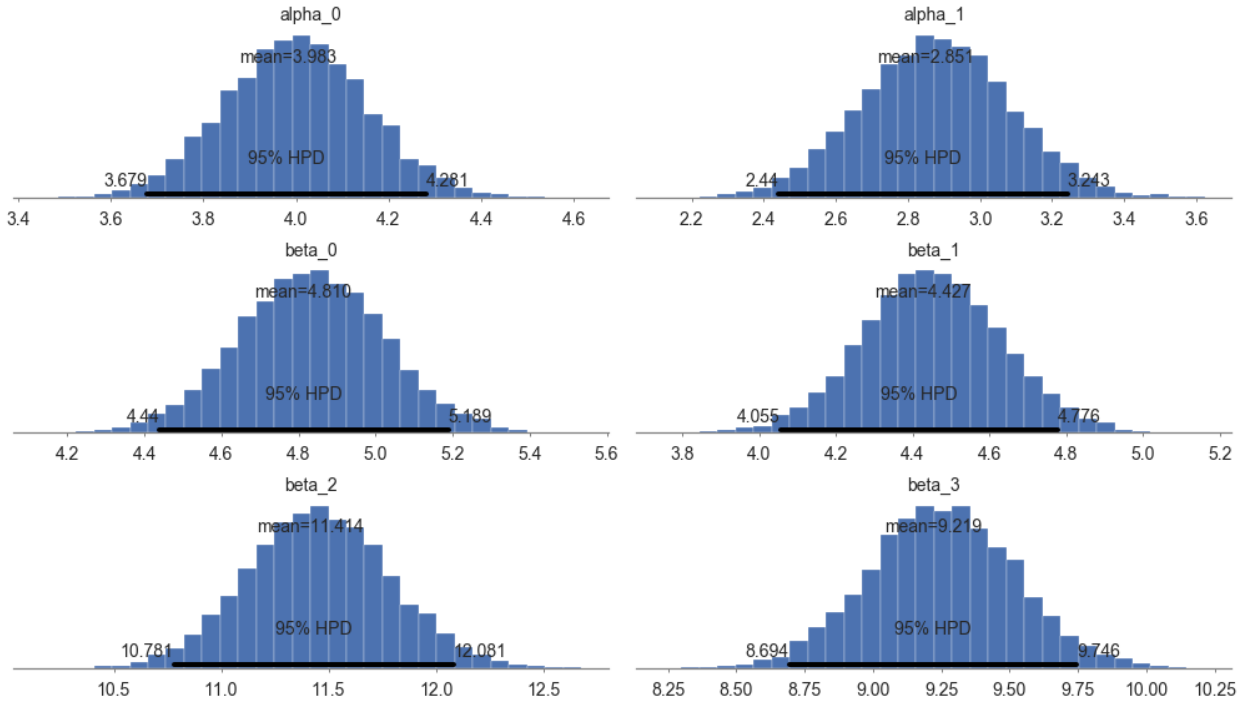
Step 5: Critize the model

```
In [16]: pm.summary(model.trace)
```

Out[16]:

	mean	sd	mc_error	hpd_2.5	hpd_97.5
alpha__0	3.982634	0.153671	0.001556	3.678890	4.280575
alpha__1	2.850619	0.206359	0.001756	2.440148	3.242568
beta__0_0	4.809822	0.189382	0.001727	4.439762	5.188622
beta__0_1	4.427498	0.183183	0.001855	4.055033	4.776228
beta__1_0	11.413951	0.333251	0.003194	10.781074	12.081359
beta__1_1	9.218845	0.267730	0.002730	8.693964	9.745963

```
In [17]: pm.plot_posterior(model.trace, figsize = [14, 8]);
```




```
In [18]: # collect the results into a pandas dataframe to display
         # "mp" stands for marginal posterior
         pd.DataFrame({"Parameter": ["beta", "alpha"],
                       "Parameter-Learned (Mean Value)": [model.trace["beta"].mean(axis=0),
                                                          model.trace["alpha"].mean(axis=0)],
                       "True value": [betas, alphas]})
```

```
Out[18]:   Parameter                      Parameter-Learned (Mean Value)  \
         0      beta  [[4.80982191646, 4.4274983607], [11.413950812,...
         1     alpha                      [3.98263424275, 2.85061932727]

                                                 True value
         0  [[4.80561284943, 4.44269695653], [29.657805725...
         1                    [3.97646170258, 7.39471669029]
```

Step 6: Use the model for prediction

```
In [19]: y_probs = model.predict_proba(X_test, cats_test)
100%|| 2000/2000 [01:24<00:00, 23.62it/s]
In [20]: y_predicted = model.predict(X_test, cats_test)
100%|| 2000/2000 [01:21<00:00, 24.65it/s]
In [21]: model.score(X_test, y_test, cats_test)
100%|| 2000/2000 [01:23<00:00, 23.97it/s]
Out[21]: 0.9580642857142857
In [22]: model.save('pickle_jar/logistic_model')
```

Use already trained model for prediction

```
In [23]: model_new = LogisticRegression()

In [25]: model_new.load('pickle_jar/logistic_model')

In [26]: model_new.score(X_test, y_test, cats_test)

100%|| 2000/2000 [01:23<00:00, 24.01it/s]

Out[26]: 0.9581952380952381
```

MCMC

```
In [ ]: model2 = LogisticRegression()
        model2.fit(X_train, y_train, cats_train, inference_type='nuts')

In [ ]: pm.traceplot(model2.trace, lines = {"beta": betas,
                                            "alpha": alphas},
                      varnames=["beta", "alpha"]);

In [ ]: pm.gelman_rubin(model2.trace)

In [ ]: pm.energyplot(model2.trace);

In [ ]: pm.summary(model2.trace)

In [ ]: pm.plot_posterior(model2.trace, figsize = [14, 8]);

In [ ]: y_predict2 = model2.predict(X_test)

In [ ]: model2.score(X_test, y_test)

In [ ]: model2.save('pickle_jar/logistic_model2')
        model2_new = LogisticRegression()
        model2_new.load('pickle_jar/logistic_model2')
        model2_new.score(X_test, y_test, cats_test)
```

7.5.3 Hierachical Logistic Regression

Let's set some setting for this Jupyter Notebook.

```
In [3]: %matplotlib inline
        from warnings import filterwarnings
        filterwarnings("ignore")
        import os
        os.environ['MKL_THREADING_LAYER'] = 'GNU'
        os.environ['THEANO_FLAGS'] = 'device=cpu'

        import numpy as np
        import pandas as pd
        import pymc3 as pm
        import seaborn as sns
        import matplotlib.pyplot as plt
        np.random.seed(12345)
        rc = {'xtick.labelsize': 20, 'ytick.labelsize': 20, 'axes.labelsize': 20, 'font.size': 20,
              'legend.fontsize': 12.0, 'axes.titlesize': 10, "figure.figsize": [12, 6]}
        sns.set(rc = rc)
        from IPython.core.interactiveshell import InteractiveShell
        InteractiveShell.ast_node_interactivity = "all"
```

Now, let's import the `HierarchicalLogisticRegression` model from the `pymc-learn` package.

```
In [4]: import pmlearn
        from pmlearn.linear_model import HierarchicalLogisticRegression
        print('Running on pymc-learn v{}'.format(pmlearn.__version__))
```

```
Running on pymc-learn v0.0.1.rc0
```

Step 1: Prepare the data

Generate synthetic data.

```
In [5]: num_pred = 2
        num_samples = 700000
        num_categories = 2
```

```
In [6]: alphas = 5 * np.random.randn(num_categories) + 5 # mu_alpha = sigma_alpha = 5
        betas = 10 * np.random.randn(num_categories, num_pred) + 10 # mu_beta = sigma_beta = 10
```

```
In [7]: alphas
```

```
Out[7]: array([ 3.9764617 ,  7.39471669])
```

```
In [8]: betas
```

```
Out[8]: array([[  4.80561285,   4.44269696],
               [ 29.65780573,  23.93405833]])
```

```
In [9]: def numpy_invlogit(x):
            return 1 / (1 + np.exp(-x))
```

```
In [10]: x_a = np.random.randn(num_samples, num_pred)
         y_a = np.random.binomial(1, numpy_invlogit(alphas[0] + np.sum(betas[0] * x_a, 1)))
         x_b = np.random.randn(num_samples, num_pred)
         y_b = np.random.binomial(1, numpy_invlogit(alphas[1] + np.sum(betas[1] * x_b, 1)))

         X = np.concatenate([x_a, x_b])
         y = np.concatenate([y_a, y_b])
         cats = np.concatenate([
             np.zeros(num_samples, dtype=np.int),
             np.ones(num_samples, dtype=np.int)
         ])
```

```
In [11]: from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test, cats_train, cats_test = train_test_split(X, y, cats, test_
```

Step 2: Instantiate a model

```
In [12]: model = HierarchicalLogisticRegression()
```

Step 3: Perform Inference

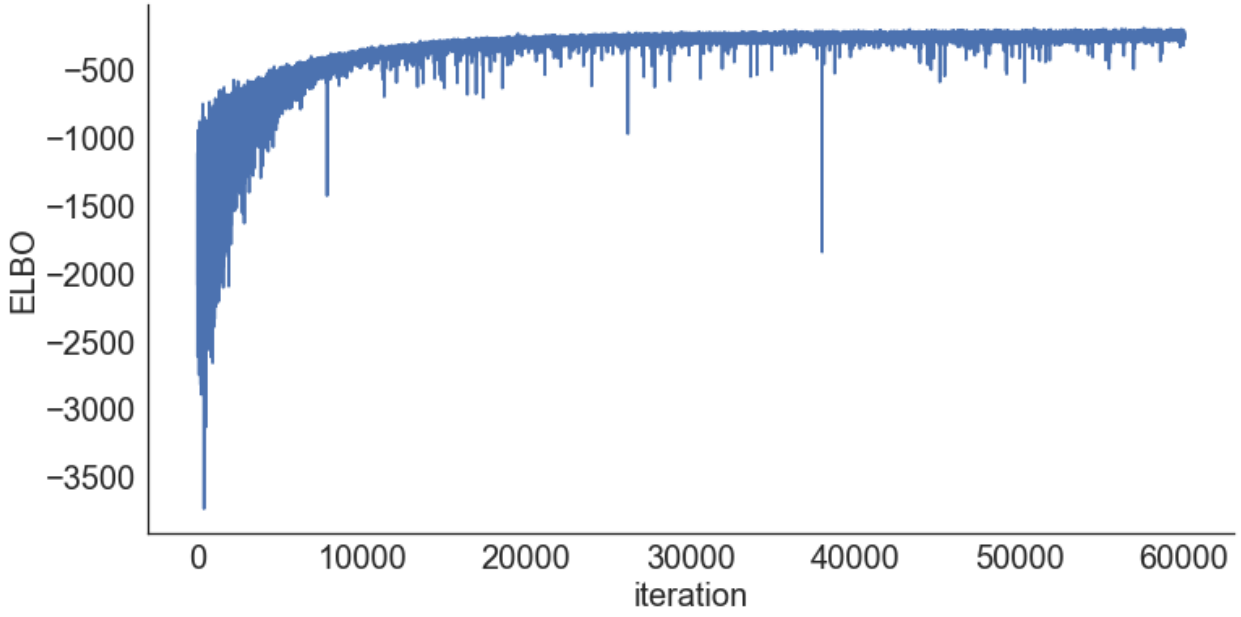
```
In [13]: model.fit(X_train, y_train, cats_train, minibatch_size=2000, inference_args={'n': 60000})
```

```
Average Loss = 246.46: 100%|| 60000/60000 [02:19<00:00, 429.21it/s]
Finished [100%]: Average Loss = 246.56
```

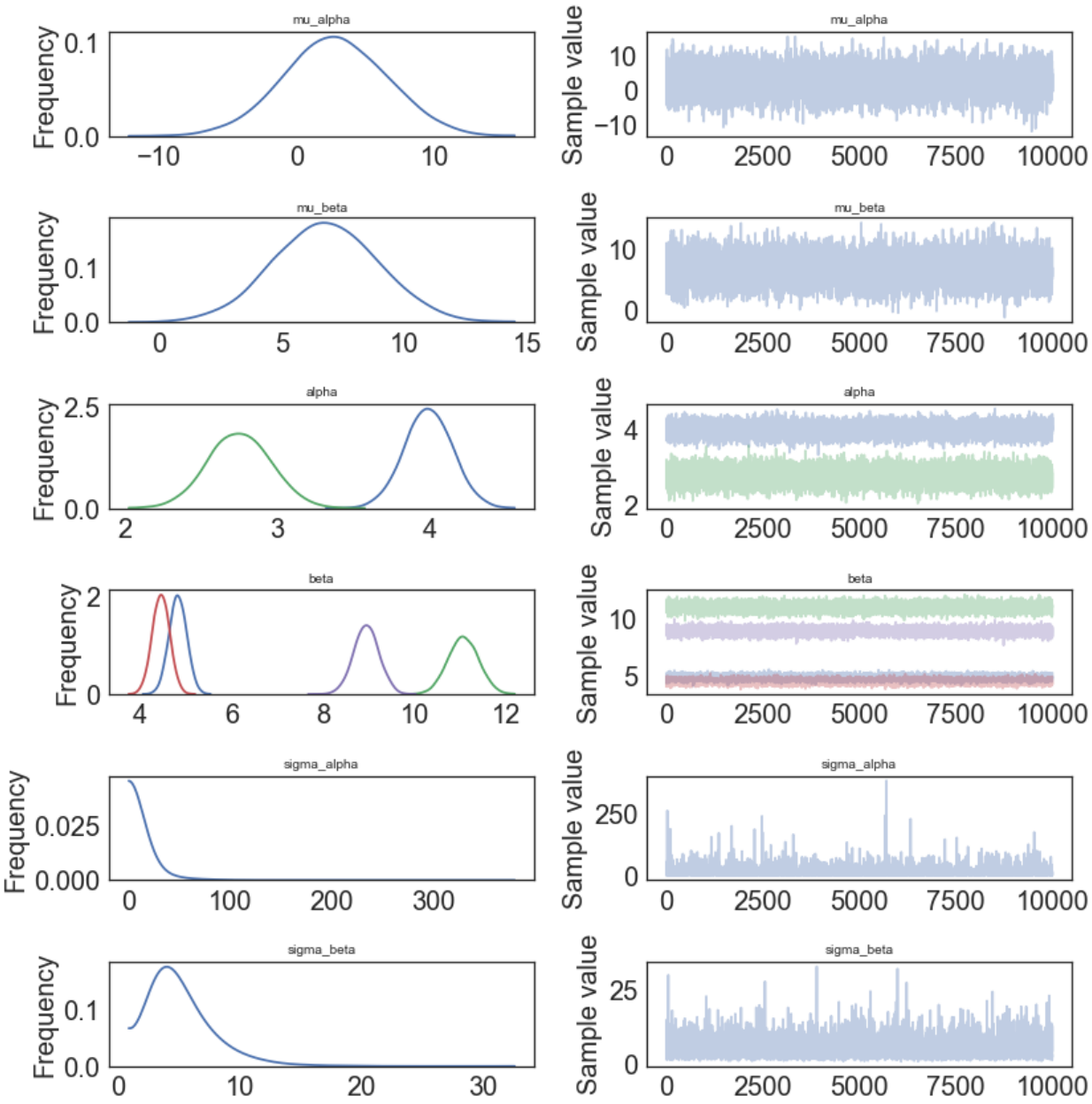
```
Out[13]: HierarchicalLogisticRegression()
```

Step 4: Diagnose convergence

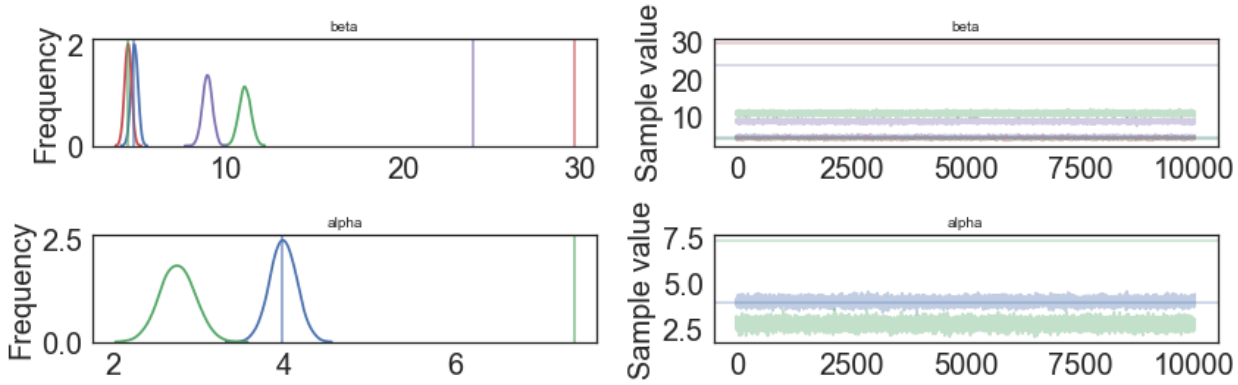
```
In [14]: model.plot_elbo()
```



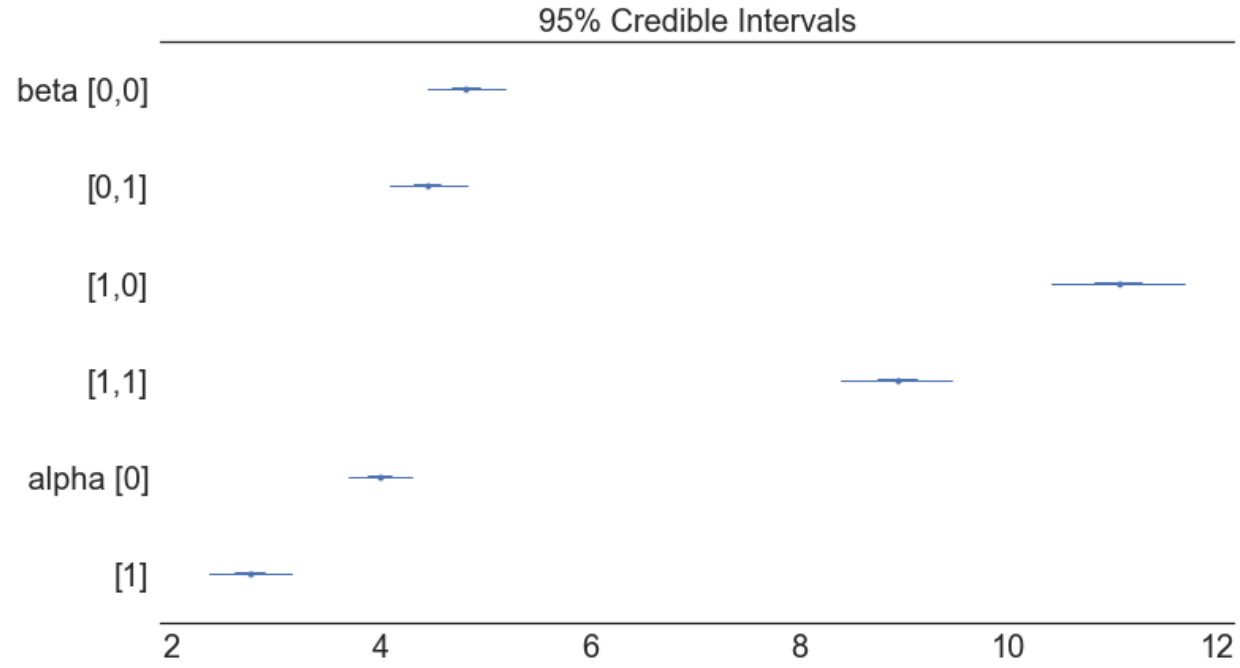

```
In [15]: pm.traceplot(model.trace);
```




```
In [16]: pm.traceplot(model.trace, lines = {"beta": betas,
                                            "alpha": alphas},
                       varnames=["beta", "alpha"]);
```




```
In [17]: pm.forestplot(model.trace, varnames=["beta", "alpha"]);
```



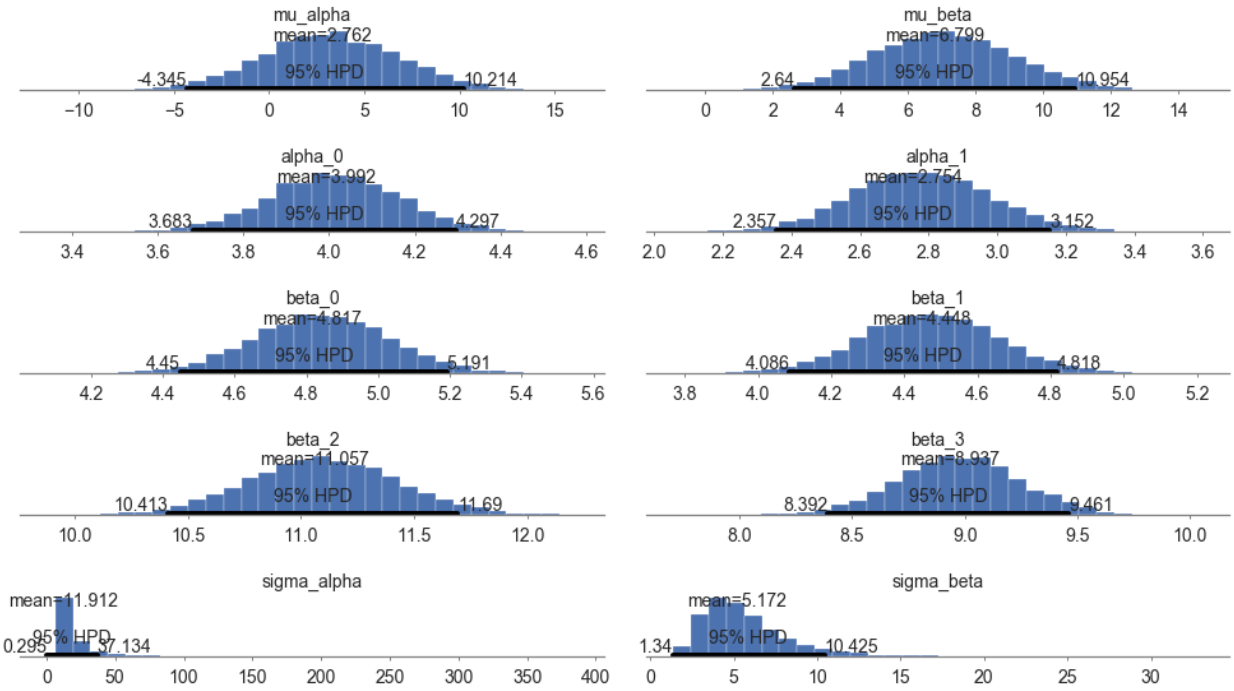
Step 5: Critize the model

```
In [18]: pm.summary(model.trace)
```

Out[18]:

	mean	sd	mc_error	hpd_2.5	hpd_97.5
mu_alpha	2.762447	3.702337	0.035507	-4.345352	10.213894
mu_beta	6.799099	2.119254	0.020894	2.640162	10.953644
alpha__0	3.992272	0.154989	0.001559	3.683080	4.296882
alpha__1	2.753978	0.203159	0.002084	2.356660	3.152206
beta__0_0	4.817129	0.190077	0.001997	4.449666	5.191381
beta__0_1	4.447704	0.186797	0.002043	4.086421	4.817914
beta__1_0	11.057359	0.327623	0.002926	10.413378	11.689952
beta__1_1	8.937158	0.271224	0.002630	8.392272	9.461213
sigma_alpha	11.912198	15.283716	0.165522	0.294809	37.133923
sigma_beta	5.171519	2.691029	0.028385	1.340010	10.425355

```
In [19]: pm.plot_posterior(model.trace, figsize = [14, 8]);
```




```
In [20]: # collect the results into a pandas dataframe to display
         # "mp" stands for marginal posterior
         pd.DataFrame({"Parameter": ["beta", "alpha"],
                       "Parameter-Learned (Mean Value)": [model.trace["beta"].mean(axis=0),
                                                          model.trace["alpha"].mean(axis=0)],
                       "True value": [betas, alphas]})
```

```
Out[20]:   Parameter                      Parameter-Learned (Mean Value)  \
         0      beta  [[4.81712892701, 4.44770360874], [11.057358919...
         1     alpha                     [3.99227159692, 2.75397848498]

                                                 True value
         0  [[4.80561284943, 4.44269695653], [29.657805725...
         1                    [3.97646170258, 7.39471669029]
```

Step 6: Use the model for prediction

```
In [21]: y_probs = model.predict_proba(X_test, cats_test)
```

```
100%|| 2000/2000 [02:07<00:00, 15.50it/s]
```

```
In [22]: y_predicted = model.predict(X_test, cats_test)
```

```
100%|| 2000/2000 [02:12<00:00, 15.66it/s]
```

```
In [23]: model.score(X_test, y_test, cats_test)
```

```
100%|| 2000/2000 [02:10<00:00, 15.18it/s]
```

```
Out[23]: 0.95806190476190478
```

```
In [24]: model.save('pickle_jar/hlogistic_model')
```

Use already trained model for prediction

```
In [25]: model_new = HierarchicalLogisticRegression()
```

```
In [26]: model_new.load('pickle_jar/hlogistic_model')

In [27]: model_new.score(X_test, y_test, cats_test)

100%|| 2000/2000 [01:25<00:00, 23.49it/s]

Out[27]: 0.95800952380952376
```

MCMC

```
In [ ]: model2 = HierarchicalLogisticRegression()
        model2.fit(X_train, y_train, cats_train, inference_type='nuts')

In [ ]: pm.traceplot(model2.trace, lines = {"beta": betas,
                                            "alpha": alphas},
                      varnames=["beta", "alpha"]);

In [ ]: pm.gelman_rubin(model2.trace)

In [ ]: pm.energyplot(model2.trace);

In [ ]: pm.summary(model2.trace)

In [ ]: pm.plot_posterior(model2.trace, figsize = [14, 8]);

In [ ]: y_predict2 = model2.predict(X_test)

In [ ]: model2.score(X_test, y_test)

In [ ]: model2.save('pickle_jar/hlogistic_model2')
        model2_new = LogisticRegression()
        model2_new.load('pickle_jar/hlogistic_model2')
        model2_new.score(X_test, y_test, cats_test)
```

7.5.4 Gaussian Process Regression

Let's set some setting for this Jupyter Notebook.

```
In [1]: %matplotlib inline
        from warnings import filterwarnings
        filterwarnings("ignore")
        import os
        os.environ['MKL_THREADING_LAYER'] = 'GNU'
        os.environ['THEANO_FLAGS'] = 'device=cpu'

        import numpy as np
        import pandas as pd
        import pymc3 as pm
        import seaborn as sns
        import matplotlib.pyplot as plt
        np.random.seed(12345)
        rc = {'xtick.labelsize': 20, 'ytick.labelsize': 20, 'axes.labelsize': 20, 'font.size': 20,
              'legend.fontsize': 12.0, 'axes.titlesize': 10, "figure.figsize": [12, 6]}
        sns.set(rc = rc)
        from IPython.core.interactiveshell import InteractiveShell
        InteractiveShell.ast_node_interactivity = "all"
```

Now, let's import the `GaussianProcessRegression` algorithm from the `pymc-learn` package.

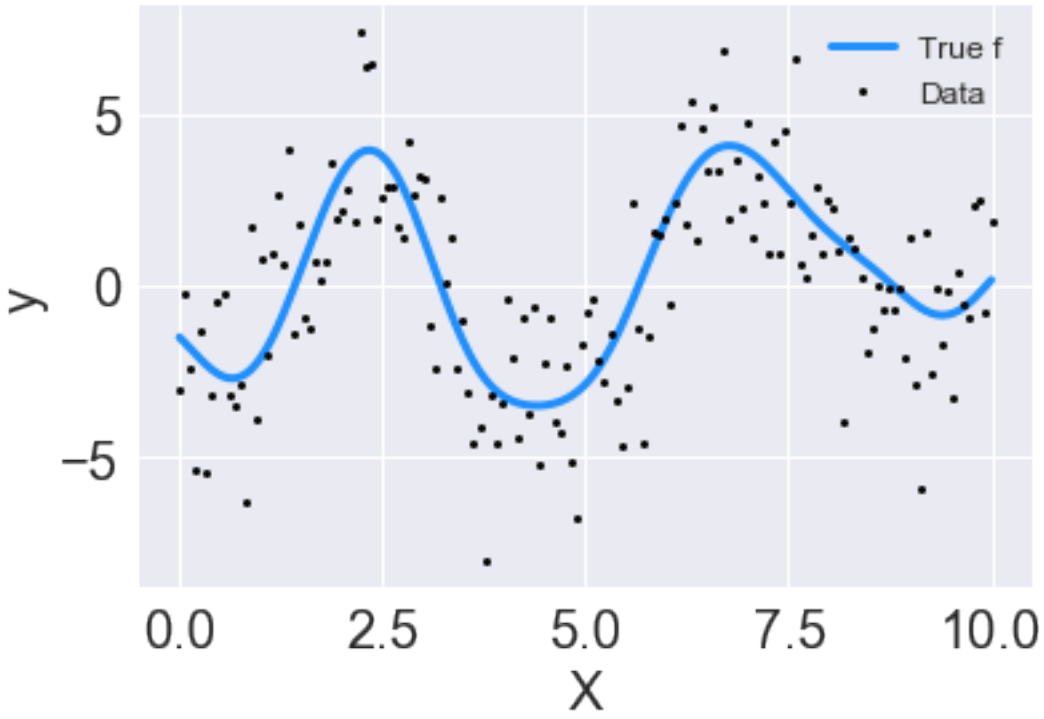
```
In [2]: import pmlearn
        from pmlearn.gaussian_process import GaussianProcessRegressor
        print('Running on pymc-learn v{}'.format(pmlearn.__version__))
```

```
Running on pymc-learn v0.0.1.rc0
```

Step 1: Prepare the data

Generate synthetic data.

```
In [3]: n = 150 # The number of data points
        X = np.linspace(start = 0, stop = 10, num = n)[:, None] # The inputs to the GP, they must be

        # Define the true covariance function and its parameters
        length_scale_true = 1.0
        signal_variance_true = 3.0
        cov_func = signal_variance_true**2 * pm.gp.cov.ExpQuad(1, length_scale_true)

        # A mean function that is zero everywhere
        mean_func = pm.gp.mean.Zero()

        # The latent function values are one sample from a multivariate normal
        # Note that we have to call `eval()` because PyMC3 built on top of Theano
        f_true = np.random.multivariate_normal(mean_func(X).eval(),
                                               cov_func(X).eval() + 1e-8*np.eye(n), 1).flatten()

        # The observed data is the latent function plus a small amount of Gaussian distributed noise
        # The standard deviation of the noise is `sigma`
        noise_variance_true = 2.0
        y = f_true + noise_variance_true * np.random.randn(n)

        ## Plot the data and the unobserved latent function
        fig = plt.figure()
        ax = fig.gca()
        ax.plot(X, f_true, "dodgerblue", lw=3, label="True f");
        ax.plot(X, y, 'ok', ms=3, label="Data");
        ax.set_xlabel("X"); ax.set_ylabel("y"); plt.legend();
```




```
In [4]: from sklearn.model_selection import train_test_split
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

Step 2: Instantiate a model

```
In [5]: model = GaussianProcessRegressor()
```

```
In [6]: model?
```

```
[0;31mType:[0m        GaussianProcessRegressor
[0;31mString form:[0m GaussianProcessRegressor(kernel=None, prior_mean=None)
[0;31mFile:[0m        ~/pymc-learn/pmlearn/gaussian_process/gpr.py
[0;31mDocstring:[0m
Gaussian Process Regression built using PyMC3.

Fit a Gaussian process model and estimate model parameters using
MCMC algorithms or Variational Inference algorithms

Parameters
----------
prior_mean : mean object
    The mean specifying the mean function of the GP. If None is passed,
    the mean "pm.gp.mean.Zero()" is used as default.

Examples
--------
>>> from sklearn.datasets import make_friedman2
>>> from pmlearn.gaussian_process import GaussianProcessRegressor
>>> from pmlearn.gaussian_process.kernels import DotProduct, WhiteKernel
>>> X, y = make_friedman2(n_samples=500, noise=0, random_state=0)
>>> kernel = DotProduct() + WhiteKernel()
>>> gpr = GaussianProcessRegressor(kernel=kernel).fit(X, y)
```

```
>>> gpr.score(X, y) # doctest: +ELLIPSIS
0.3680...
>>> gpr.predict(X[:2,:], return_std=True) # doctest: +ELLIPSIS
(array([653.0..., 592.1...]), array([316.6..., 316.6...]))


Reference
----------
Rasmussen and Williams (2006). Gaussian Processes for Machine Learning.
```

Step 3: Perform Inference

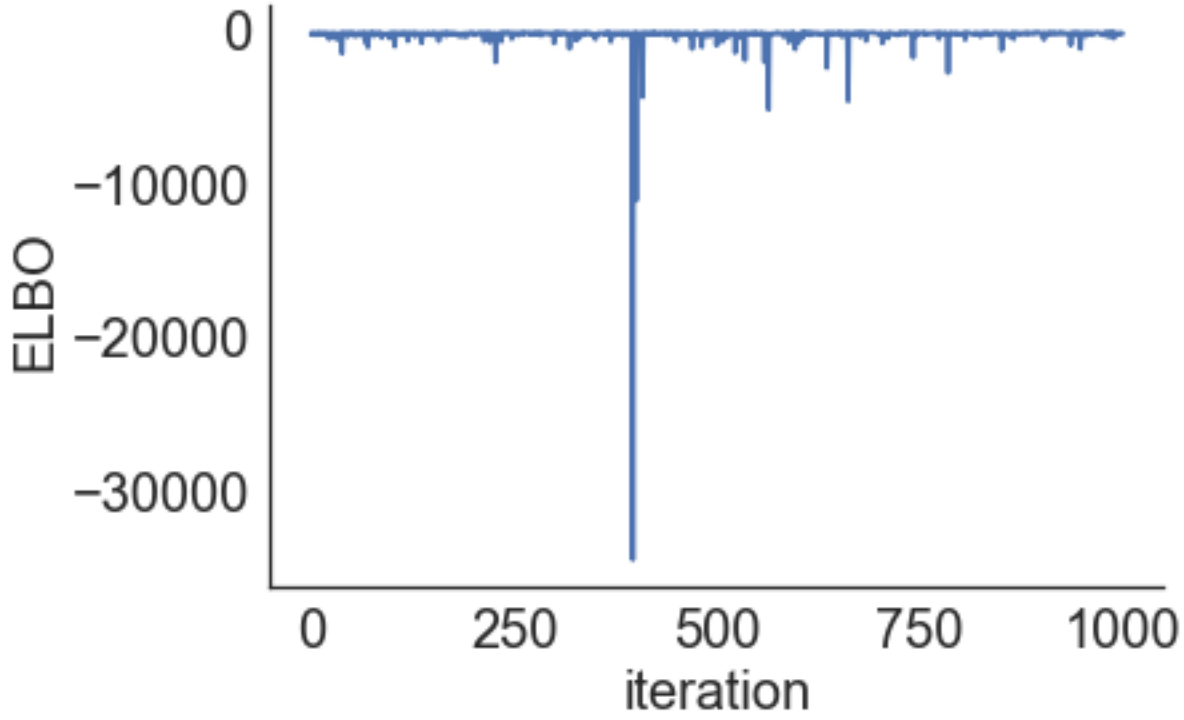
```
In [7]: model.fit(X_train, y_train, inference_args={"n": 1000})
```

```
Average Loss = 416.16: 100%|| 1000/1000 [00:02<00:00, 474.85it/s]
Finished [100%]: Average Loss = 415.55
```

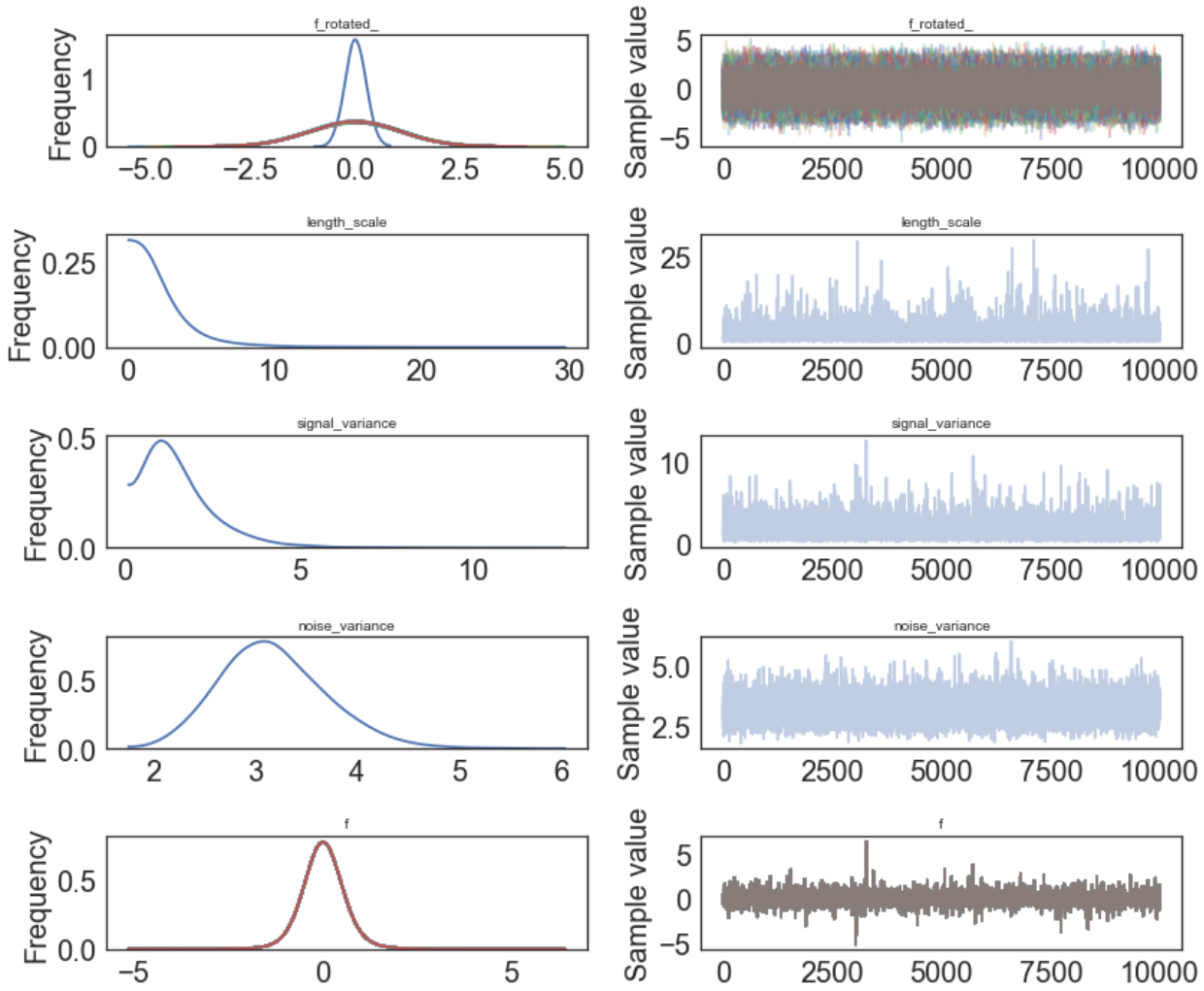
```
Out[7]: GaussianProcessRegressor(kernel=None, prior_mean=None)
```

Step 4: Diagnose convergence

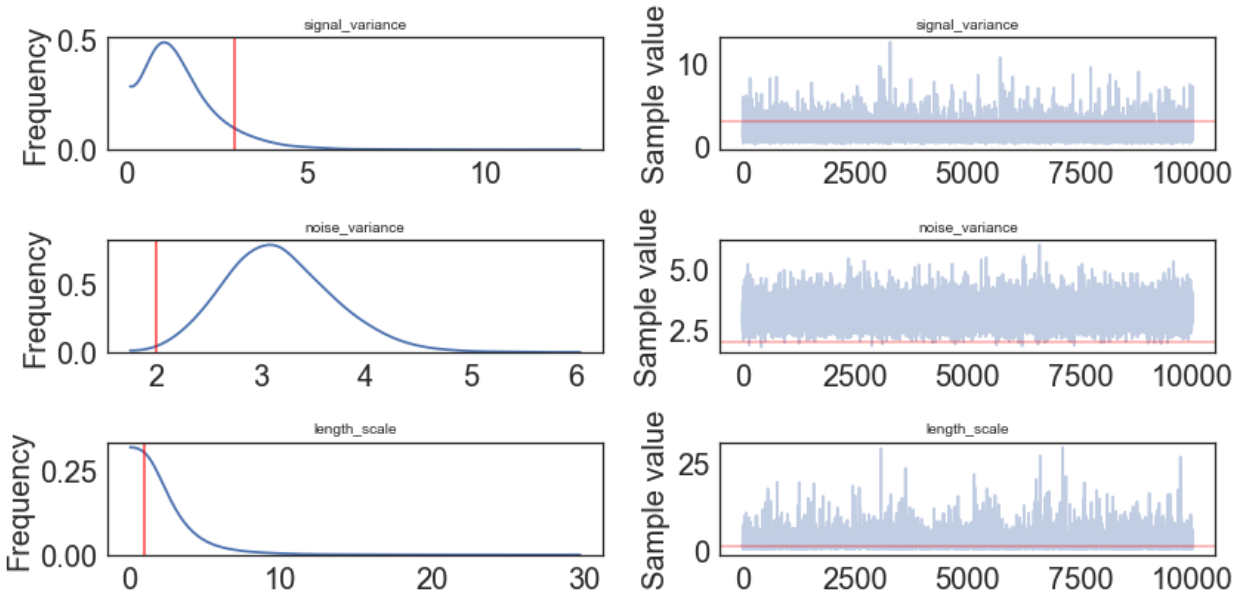
```
In [8]: model.plot_elbo()
```



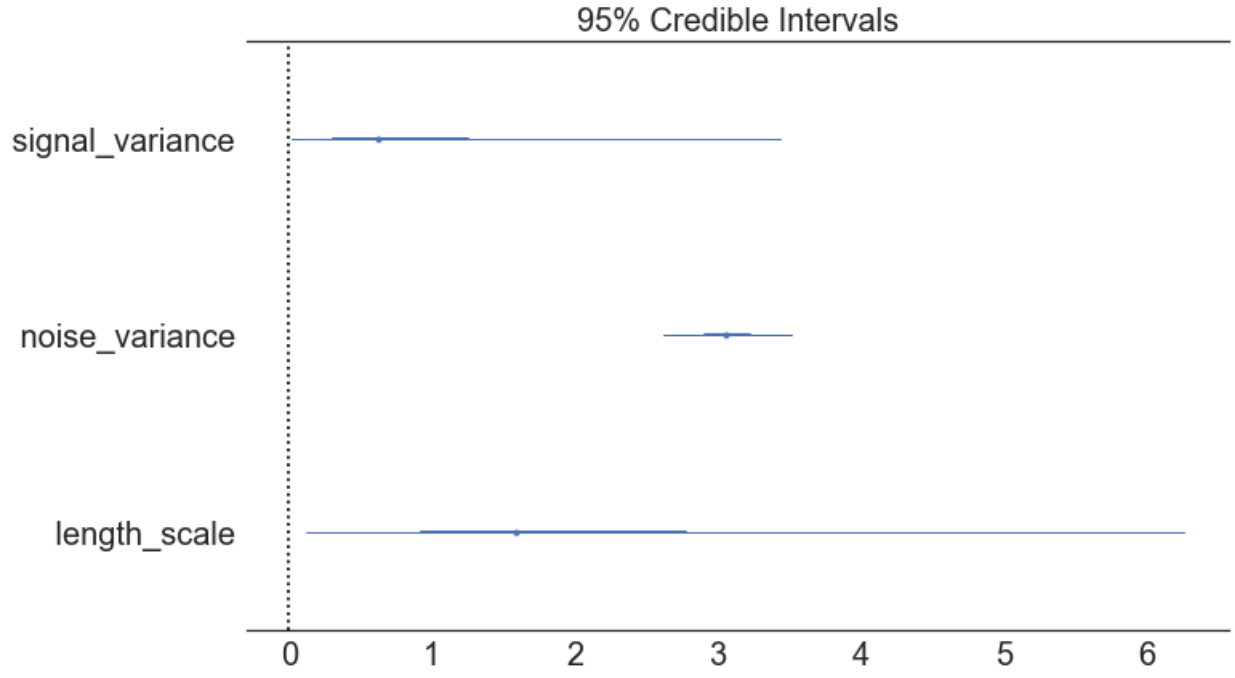

```
In [11]: pm.traceplot(model.trace);
```




```
In [12]: pm.traceplot(model.trace, lines = {"signal_variance": signal_variance_true,
                                            "noise_variance": noise_variance_true,
                                            "length_scale": length_scale_true},
                      varnames=["signal_variance", "noise_variance", "length_scale"]);
```




```
In [12]: pm.forestplot(model.trace, varnames=["signal_variance", "noise_variance", "length_scale"]);
```



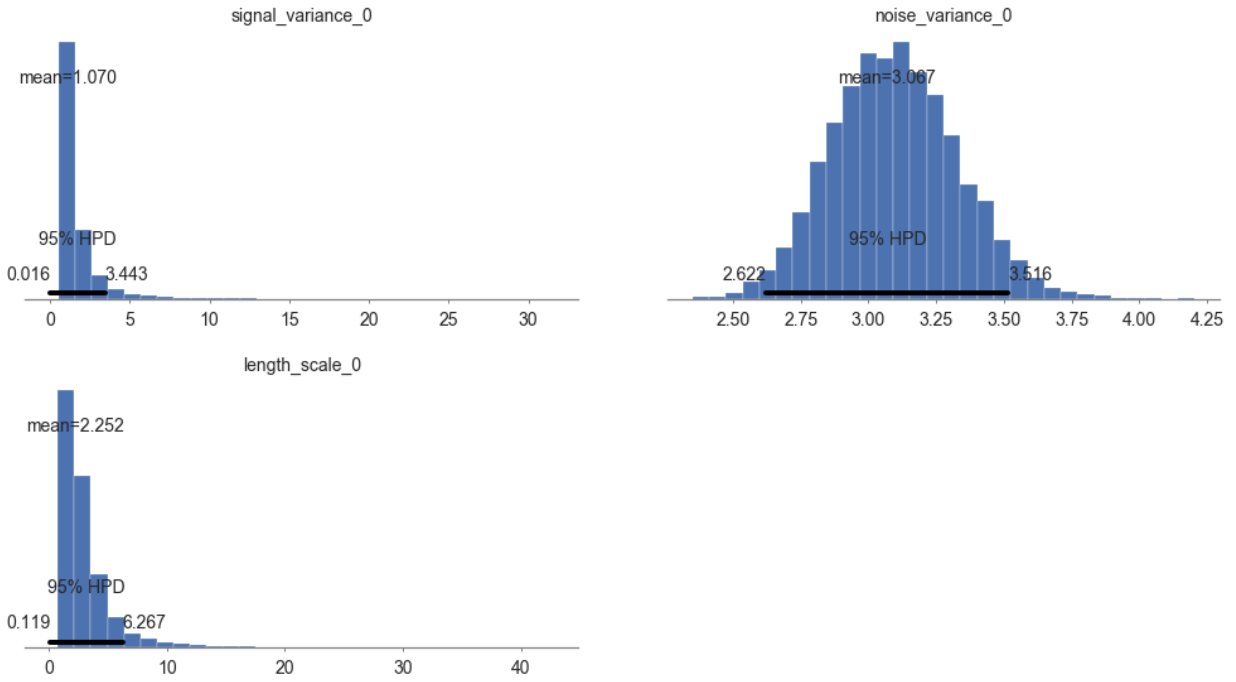
Step 5: Critize the model

```
In [13]: pm.summary(model.trace, varnames=["signal_variance", "length_scale", "noise_variance"])
```

Out[13]:

	mean	sd	mc_error	hpd_2.5	hpd_97.5
signal_variance__0	1.069652	1.472790	0.014072	0.016441	3.442904
length_scale__0_0	2.252174	2.231719	0.025261	0.119457	6.266875
noise_variance__0	3.066997	0.231325	0.002249	2.622387	3.516474

```
In [14]: pm.plot_posterior(model.trace, varnames=["signal_variance", "noise_variance", "length_scale"
                         figsize = [14, 8]);
```




```
In [15]: # collect the results into a pandas dataframe to display
         # "mp" stands for marginal posterior
         pd.DataFrame({"Parameter": ["length_scale", "signal_variance", "noise_variance"],
                       "Predicted Mean Value": [float(model.trace["length_scale"].mean(axis=0)),
                                                float(model.trace["signal_variance"].mean(axis=0)),
                                                float(model.trace["noise_variance"].mean(axis=0))],
                       "True value": [length_scale_true, signal_variance_true, noise_variance_true]})
```

Out[15]:

	Parameter	Predicted Mean Value	True value
0	length_scale	2.252174	1.0
1	signal_variance	1.069652	3.0
2	noise_variance	3.066997	2.0

Step 6: Use the model for prediction

```
In [9]: y_predict1 = model.predict(X_test)
```

```
100%|| 2000/2000 [00:14<00:00, 135.47it/s]
```

```
In [10]: y_predict1
```

```
Out[10]: array([ 0.00166453,  0.07415753,  0.07185864,  0.01505948,  0.02280044,
                -0.00041549, -0.02338406,  0.01753743,  0.02065263,  0.00825294,
                 0.02449021,  0.06761137,  0.04990807,  0.01614856, -0.03135927,
                -0.00813461,  0.04545187, -0.03770688,  0.06116857,  0.06864128,
                 0.04164327, -0.01700696,  0.01389948, -0.02395358, -0.01853882,
                -0.02147422,  0.05869176, -0.02825002,  0.01058576,  0.04180517,
                 0.01563565, -0.0086748 ,  0.01048786, -0.02464047,  0.0639958 ,
                -0.02110329, -0.03658159,  0.0552832 , -0.00030839,  0.03097778,
                 0.00415975,  0.05252889,  0.00894602,  0.06400553, -0.05004306])
```

```
In [ ]: model.score(X_test, y_test)
```

```
In [12]: model.save('pickle_jar/gpr')
```

Use already trained model for prediction

```
In [13]: model_new = GaussianProcessRegressor()
         model_new.load('pickle_jar/gpr')
         model_new.score(X_test, y_test)
```

```
100%|| 2000/2000 [00:14<00:00, 136.18it/s]
```

```
Out[13]: -0.0049724872177634438
```

Multiple Features

```
In [14]: num_pred = 2
         X = np.random.randn(1000, num_pred)
         noise = 2 * np.random.randn(1000,)
         Y = X.dot(np.array([4, 5])) + 3 + noise
```

```
In [15]: y = np.squeeze(Y)
```

```
In [16]: model_big = GaussianProcessRegressor()
```

```
In [17]: model_big.fit(X, y, inference_args={"n" : 1000})
```

```
Average Loss = 6,077.1: 100%|| 1000/1000 [02:17<00:00,  7.11it/s]
Finished [100%]: Average Loss = 6,056.9
```

```
Out[17]: GaussianProcessRegressor(prior_mean=0.0)
```

```
In [18]: pm.summary(model_big.trace, varnames=["signal_variance", "length_scale", "noise_variance"])
```

```
Out[18]:                          mean        sd  mc_error   hpd_2.5   hpd_97.5
         signal_variance__0   5.420972  4.049228  0.041386  0.635510  12.967287
         length_scale__0_0    2.460546  2.034025  0.021939  0.192283   6.279051
         length_scale__0_1    2.437830  1.994458  0.018703  0.267447   6.202378
         noise_variance__0    7.173519  4.732447  0.042548  0.936711  16.368718
```

MCMC

```
In [ ]: model2 = GaussianProcessRegressor()
        model2.fit(X_train, y_train, inference_type='nuts')
```

```
In [ ]: pm.traceplot(model2.trace, lines = {"signal_variance": signal_variance_true,
                                           "noise_variance": noise_variance_true,
                                           "length_scale": length_scale_true},
                      varnames=["signal_variance", "noise_variance", "length_scale"]);
```

```
In [ ]: pm.gelman_rubin(model2.trace, varnames=["signal_variance", "noise_variance", "length_scale"])
```

```
In [ ]: pm.energyplot(model2.trace);
```

```
In [ ]: pm.forestplot(model2.trace, varnames=["signal_variance", "noise_variance", "length_scale"]);
```

```
In [ ]: pm.summary(model2.trace, varnames=["signal_variance", "length_scale", "noise_variance"])
```

```
In [ ]: # collect the results into a pandas dataframe to display
        # "mp" stands for marginal posterior
        pd.DataFrame({"Parameter": ["length_scale", "signal_variance", "noise_variance"],
                      "Predicted Mean Value": [float(model2.trace["length_scale"].mean(axis=0)),
                                               float(model2.trace["signal_variance"].mean(axis=0)),
                                               float(model2.trace["noise_variance"].mean(axis=0))],
                      "True value": [length_scale_true, signal_variance_true, noise_variance_true]})
```

```
In [ ]: pm.plot_posterior(model2.trace, varnames=["signal_variance", "noise_variance", "length_scale"
                          figsize = [14, 8]);
```

```
In [ ]: y_predict2 = model2.predict(X_test)

In [ ]: y_predict2

In [ ]: model2.score(X_test, y_test)

In [ ]: model2.save('pickle_jar/gpr2')
        model2_new = GaussianProcessRegressor
        model2_new.load('pickle_jar//gpr2')
        model2_new.score(X_test, y_test)
```

7.5.5 Student's T Process Regression

Let's set some setting for this Jupyter Notebook.

```
In [2]: %matplotlib inline
        from warnings import filterwarnings
        filterwarnings("ignore")
        import os
        os.environ['MKL_THREADING_LAYER'] = 'GNU'
        os.environ['THEANO_FLAGS'] = 'device=cpu'

        import numpy as np
        import pandas as pd
        import pymc3 as pm
        import seaborn as sns
        import matplotlib.pyplot as plt
        np.random.seed(12345)
        rc = {'xtick.labelsize': 20, 'ytick.labelsize': 20, 'axes.labelsize': 20, 'font.size': 20,
              'legend.fontsize': 12.0, 'axes.titlesize': 10, "figure.figsize": [12, 6]}
        sns.set(rc = rc)
        from IPython.core.interactiveshell import InteractiveShell
        InteractiveShell.ast_node_interactivity = "all"
```

Now, let's import the `StudentsTProcessRegression` algorithm from the `pymc-learn` package.

```
In [3]: import pmlearn
        from pmlearn.gaussian_process import StudentsTProcessRegressor
        print('Running on pymc-learn v{}'.format(pmlearn.__version__))
```

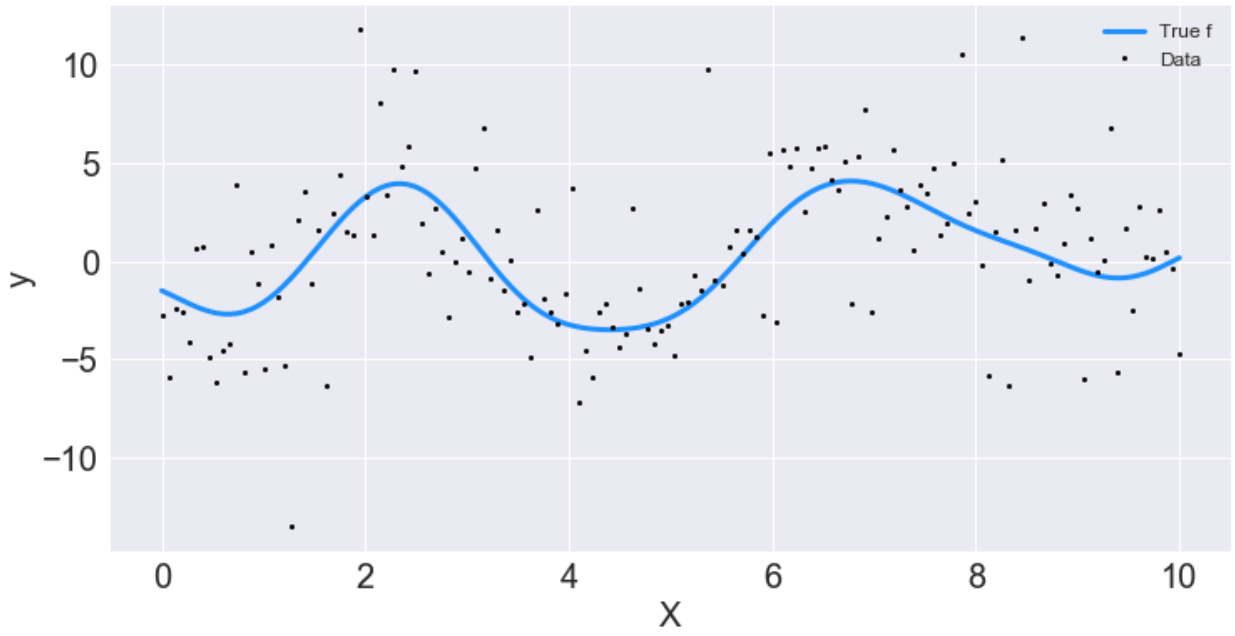
```
Running on pymc-learn v0.0.1.rc0
```

Step 1: Prepare the data

Generate synthetic data.

```
In [4]: n = 150 # The number of data points
        X = np.linspace(start = 0, stop = 10, num = n)[:, None] # The inputs to the GP, they must be

        # Define the true covariance function and its parameters
        length_scale_true = 1.0
        signal_variance_true = 3.0
        cov_func = signal_variance_true**2 * pm.gp.cov.ExpQuad(1, length_scale_true)

        # A mean function that is zero everywhere
        mean_func = pm.gp.mean.Zero()

        # The latent function values are one sample from a multivariate normal
        # Note that we have to call `eval()` because PyMC3 built on top of Theano
        f_true = np.random.multivariate_normal(mean_func(X).eval(),
```

```
                                                    cov_func(X).eval() + 1e-8*np.eye(n), 1).flatten()

    # The observed data is the latent function plus a small amount of T distributed noise
    # The standard deviation of the noise is `sigma`, and the degrees of freedom is `nu`
    noise_variance_true = 2.0
    degrees_of_freedom_true = 3.0
    y = f_true + noise_variance_true * np.random.standard_t(degrees_of_freedom_true, size=n)

    ## Plot the data and the unobserved latent function
    fig, ax = plt.subplots()
    ax.plot(X, f_true, "dodgerblue", lw=3, label="True f");
    ax.plot(X, y, 'ok', ms=3, label="Data");
    ax.set_xlabel("X"); ax.set_ylabel("y"); plt.legend();
```




```
In [5]: from sklearn.model_selection import train_test_split
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

Step 2: Instantiate a model

```
In [6]: model = StudentsTProcessRegressor()
```

Step 3: Perform Inference

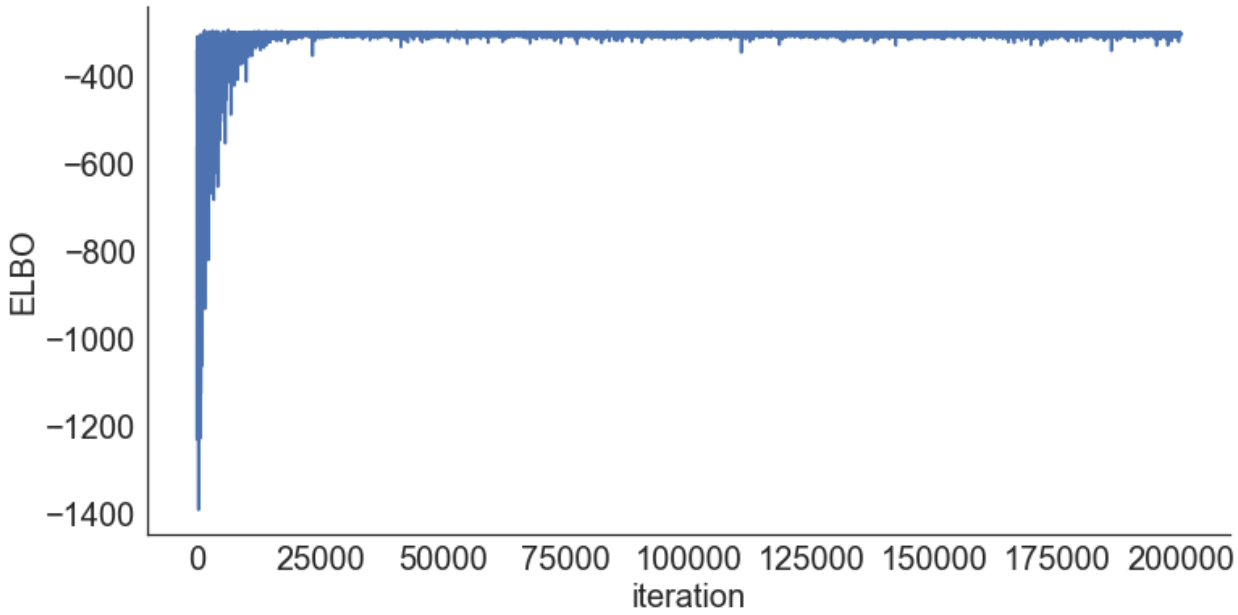
```
In [7]: model.fit(X_train, y_train)
```

```
Average Loss = 303.15: 100%|| 200000/200000 [06:37<00:00, 503.33it/s]
Finished [100%]: Average Loss = 303.15
```

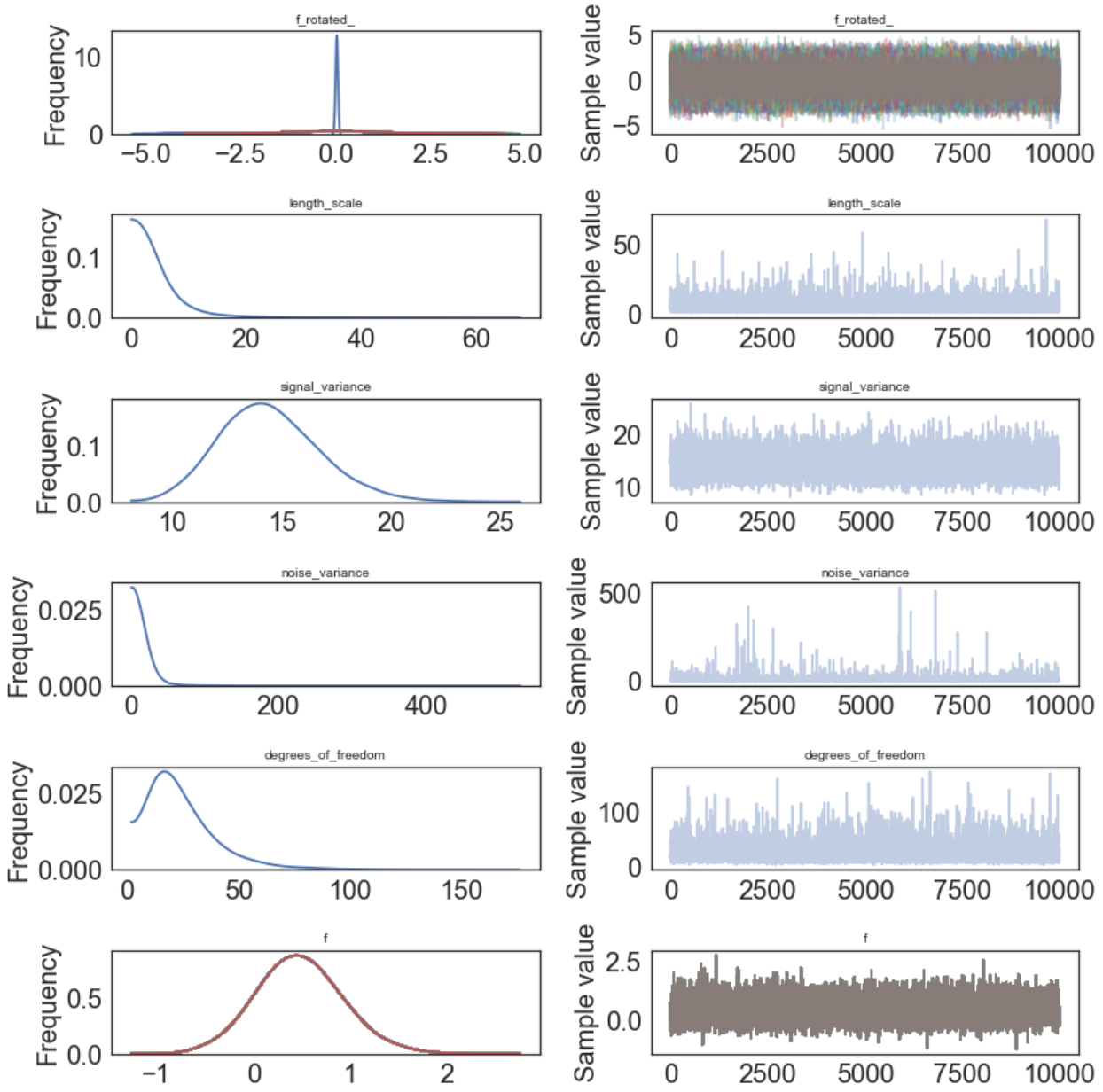
```
Out[7]: StudentsTProcessRegressor(prior_mean=0.0)
```

Step 4: Diagnose convergence

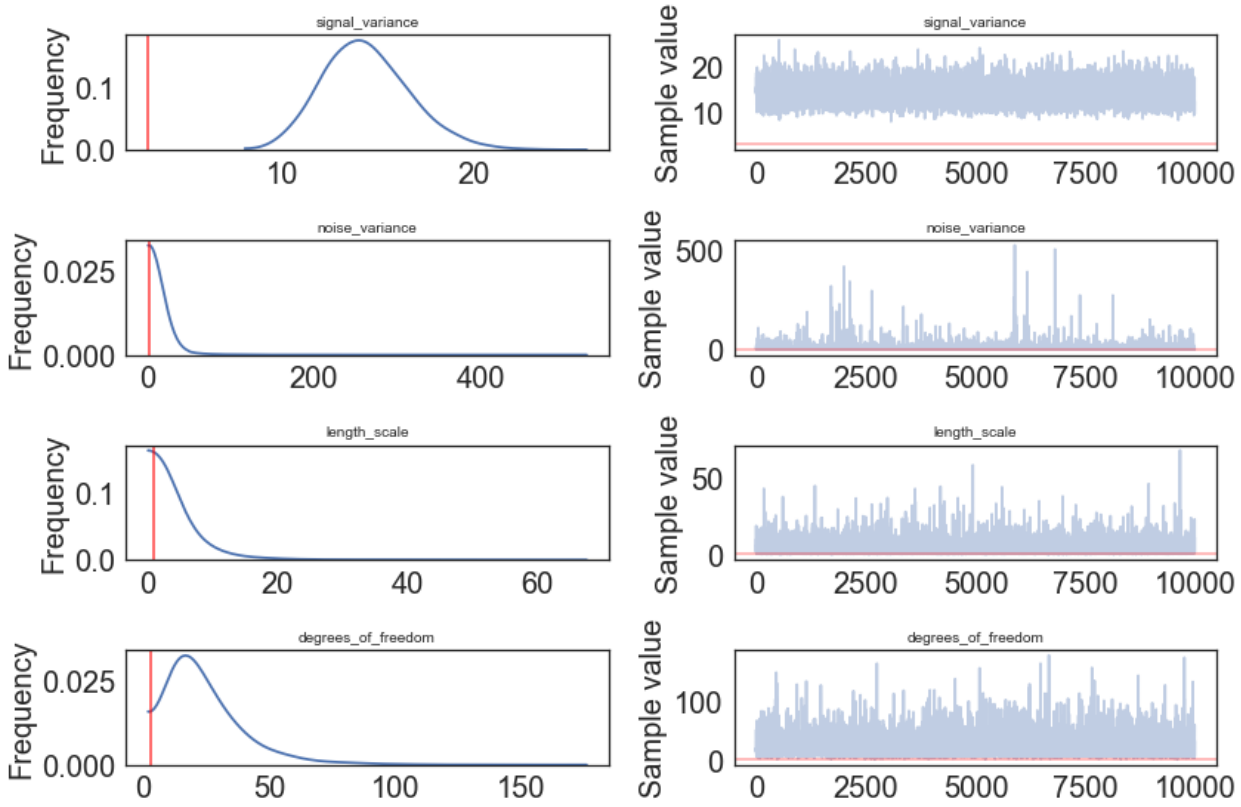
```
In [8]: model.plot_elbo()
```



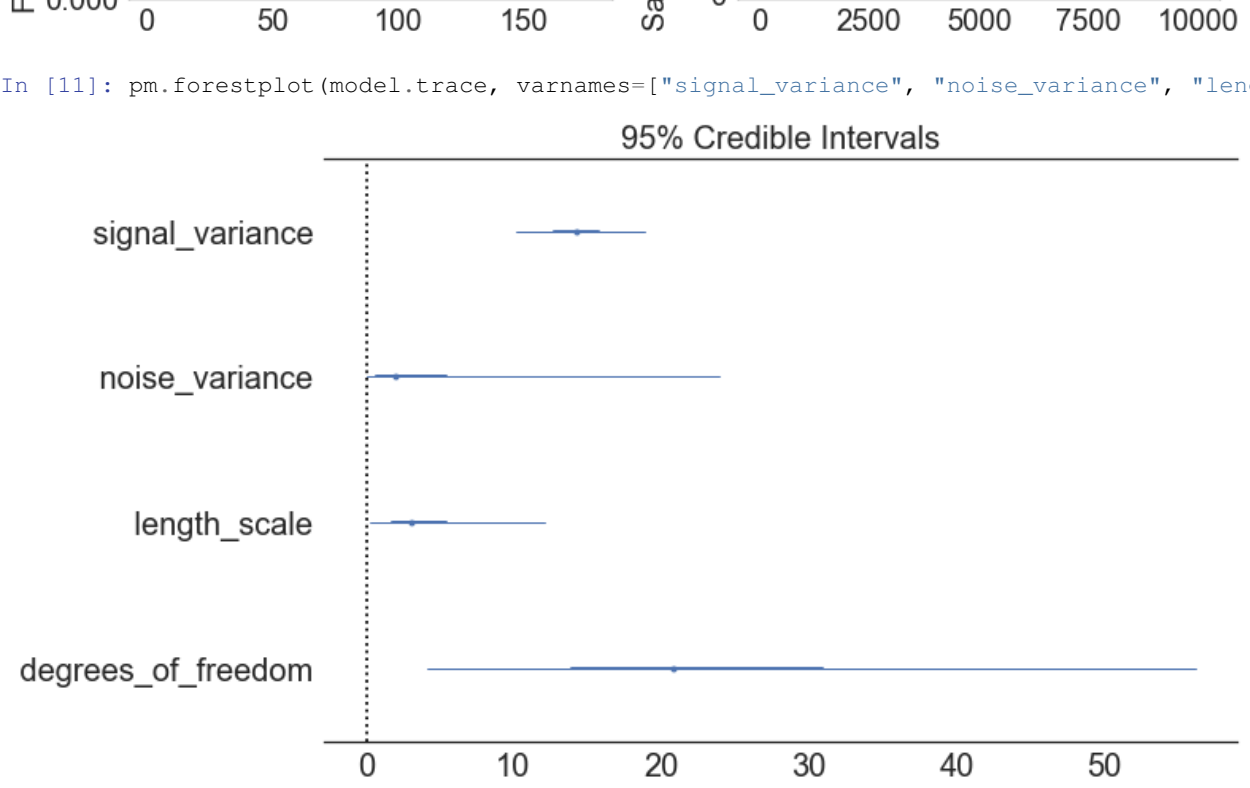
In [9]: pm.traceplot(model.trace);




```
In [10]: pm.traceplot(model.trace, lines = {"signal_variance": signal_variance_true,
                                            "noise_variance": noise_variance_true,
                                            "length_scale": length_scale_true,
                                            "degrees_of_freedom": degrees_of_freedom_true},
                      varnames=["signal_variance", "noise_variance", "length_scale", "degrees_of_free
```




```
In [11]: pm.forestplot(model.trace, varnames=["signal_variance", "noise_variance", "length_scale", "
```



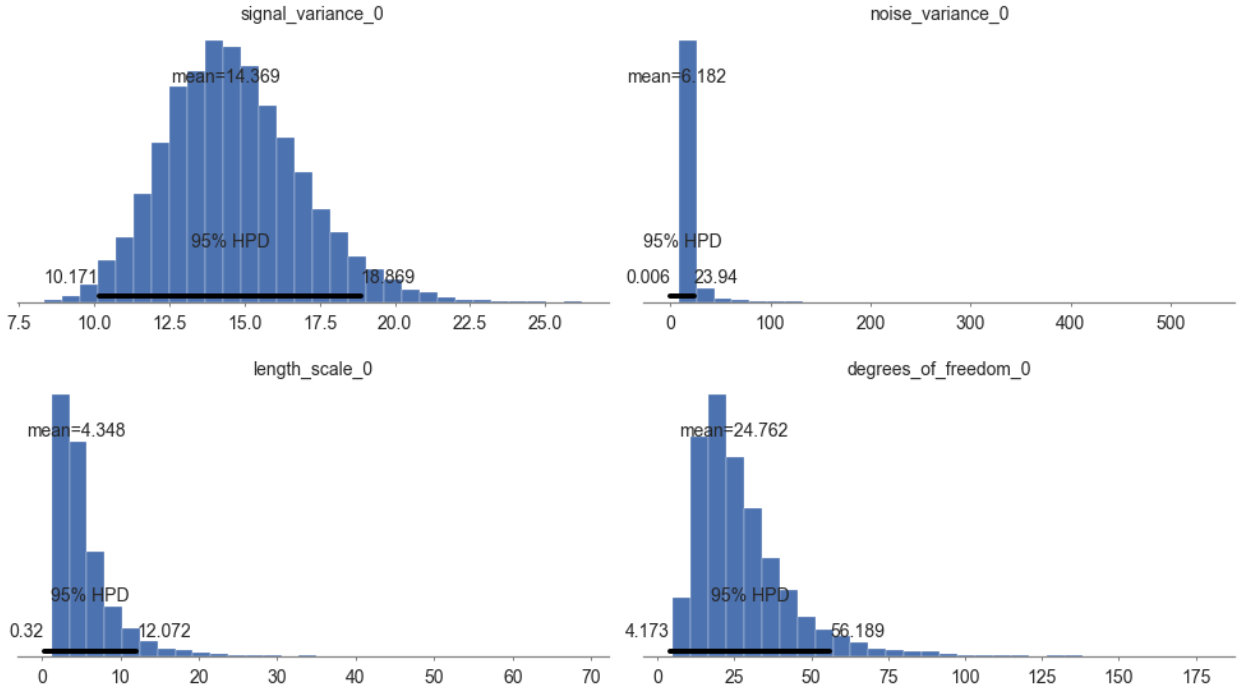
Step 5: Criticize the model

```
In [12]: pm.summary(model.trace, varnames=["signal_variance", "noise_variance", "length_scale", "degr
```

Out[12]:

	mean	sd	mc_error	hpd_2.5	hpd_97.5
signal_variance__0	14.369054	2.254097	0.021957	10.170910	18.868740
noise_variance__0	6.182008	17.330061	0.187540	0.006105	23.940172
length_scale__0_0	4.348170	4.162548	0.042757	0.319785	12.072470
degrees_of_freedom__0	24.762167	16.178113	0.149632	4.172859	56.188869

In [13]:
```
pm.plot_posterior(model.trace, varnames=["signal_variance", "noise_variance", "length_scale"
                  figsize = [14, 8]);
```



In [14]:
```
# collect the results into a pandas dataframe to display
# "mp" stands for marginal posterior
pd.DataFrame({"Parameter": ["signal_variance", "noise_variance", "length_scale", "degrees_o
              "Predicted Mean Value": [float(model.trace["length_scale"].mean(axis=0)),
                                       float(model.trace["signal_variance"].mean(axis=0)),
                                       float(model.trace["noise_variance"].mean(axis=0)),
                                       float(model.trace["degrees_of_freedom"].mean(axis=0))
              "True value": [length_scale_true, signal_variance_true,
                             noise_variance_true, degrees_of_freedom_true]})
```

Out[14]:

	Parameter	Predicted Mean Value	True value
0	signal_variance	4.348170	1.0
1	noise_variance	14.369054	3.0
2	length_scale	6.182008	2.0
3	degrees_of_freedom	24.762167	3.0

Step 6: Use the model for prediction

In [15]:
```
y_predict1 = model.predict(X_test)
```

```
100%|| 2000/2000 [00:08<00:00, 247.33it/s]
```

In [16]:
```
y_predict1
```

Out[16]:
```
array([0.52060618, 0.20859641, 0.32341845, 0.71517795, 0.12535947,
       0.10130519, 0.13356278, 0.48476055, 0.33239652, 0.05354277,
       0.3221012 , 0.27747592, 0.33224296, 0.16754793, 0.70514462,
```

```
                0.37293254, 0.38020924, 0.65038549, 0.34252208, 0.38382534,
                0.15502318, 0.37618247, 0.58213956, 0.63244638, 0.27682323,
                0.17309081, 0.11088147, 0.38385589, 0.05206571, 0.33370627,
                0.0590494 , 0.21805391, 0.24068462, 0.14248978, 0.16113507,
                0.6395228 , 0.13902426, 0.29770677, 0.24498306, 0.18377858,
                0.12288624, 0.35066241, 0.25833606, 0.70100999, 0.66802676])
```

```
In [24]: model.score(X_test, y_test)
```

```
In [26]: model.save('pickle_jar/spr')
```

Use already trained model for prediction

```
In [27]: model_new = StudentsTProcessRegressor()
         model_new.load('pickle_jar/spr')
         model_new.score(X_test, y_test)
```

```
100%|| 2000/2000 [00:01<00:00, 1201.17it/s]
```

```
Out[27]: -0.09713232621579238
```

Multiple Features

```
In [34]: num_pred = 2
         X = np.random.randn(1000, num_pred)
         noise = 2 * np.random.randn(1000,)
         Y = X.dot(np.array([4, 5])) + 3 + noise
```

```
In [35]: y = np.squeeze(Y)
```

```
In [36]: model_big = StudentsTProcessRegressor()
```

```
In [37]: model_big.fit(X, y, inference_args={"n" : 1000})
```

```
Average Loss = 6,129.9: 100%|| 1000/1000 [03:13<00:00,  5.16it/s]
Finished [100%]: Average Loss = 6,118.9
```

```
Out[37]: StudentsTProcessRegression()
```

```
In [38]: pm.summary(model_big.trace, varnames=["signal_variance", "noise_variance", "length_scale", "
```

Out[38]:

	mean	sd	mc_error	hpd_2.5	hpd_97.5
signal_variance__0	7.029373	4.739398	0.049285	0.968977	16.254415
noise_variance__0	7.163999	7.395869	0.075956	0.337118	20.242566
length_scale__0_0	2.451322	1.983714	0.022889	0.256500	6.175940
length_scale__0_1	2.466894	2.009942	0.021930	0.196610	6.184087
degrees_of_freedom__0	19.622088	15.718934	0.147647	2.390572	49.401395

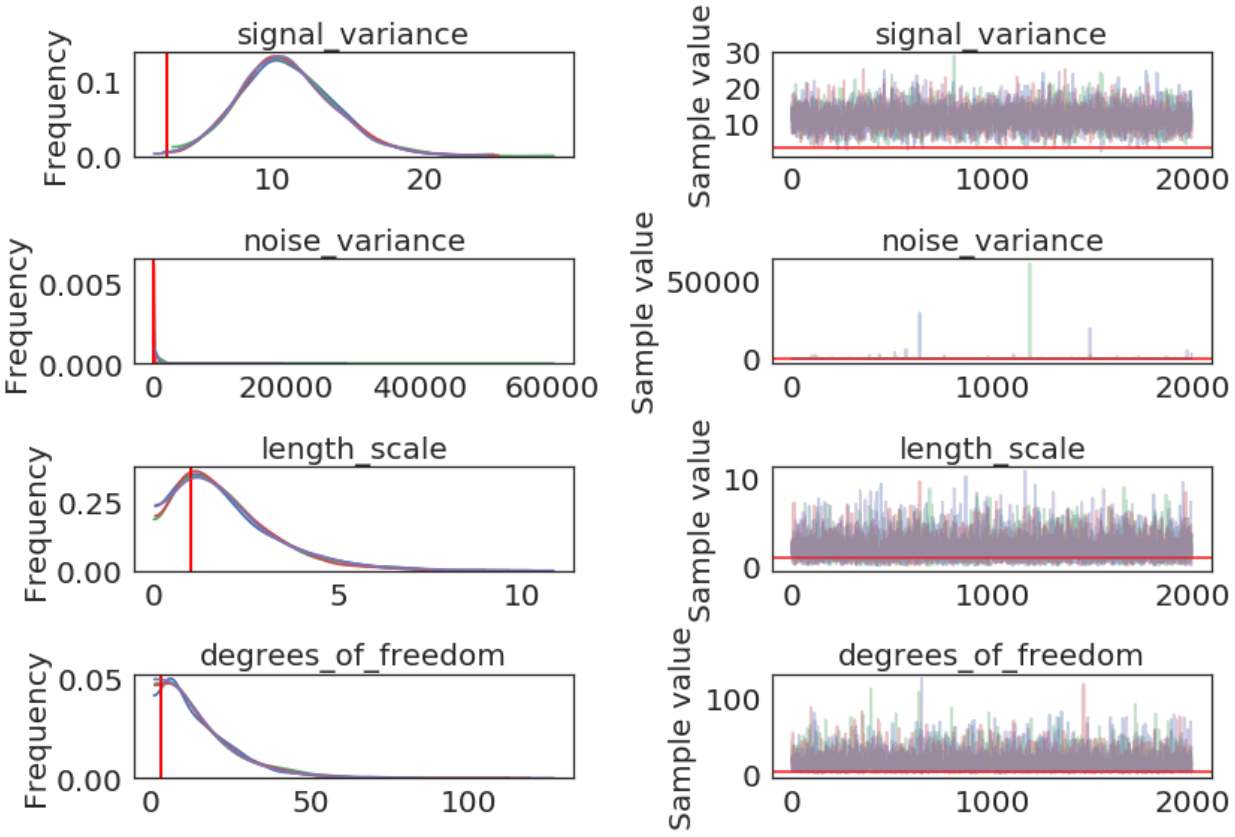
MCMC

```
In [8]: model2 = StudentsTProcessRegressor()
        model2.fit(X_train, y_train, inference_type='nuts')
```

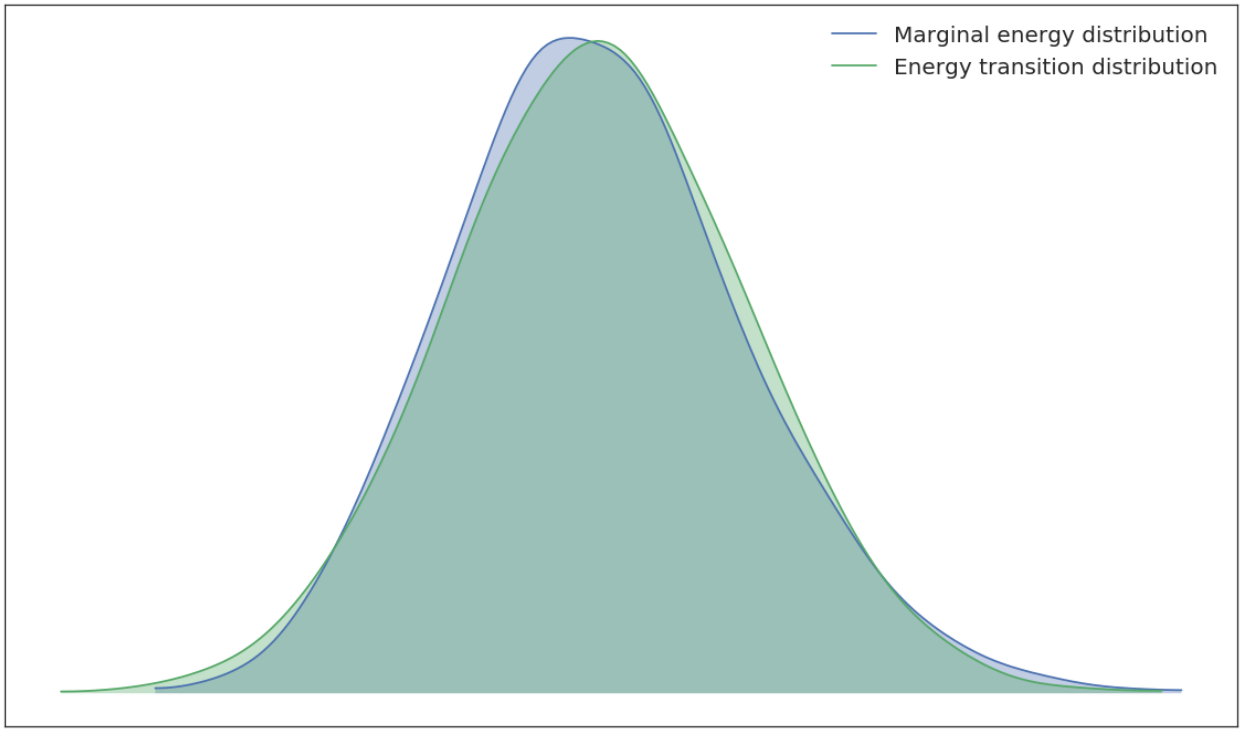
```
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [f_rotated_, degrees_of_freedom_log__, noise_variance_log__, signal_variance_log__, length_scal
100%|| 2500/2500 [03:33<00:00, 11.70it/s]
```

```
Out[8]: StudentsTProcessRegression()
```

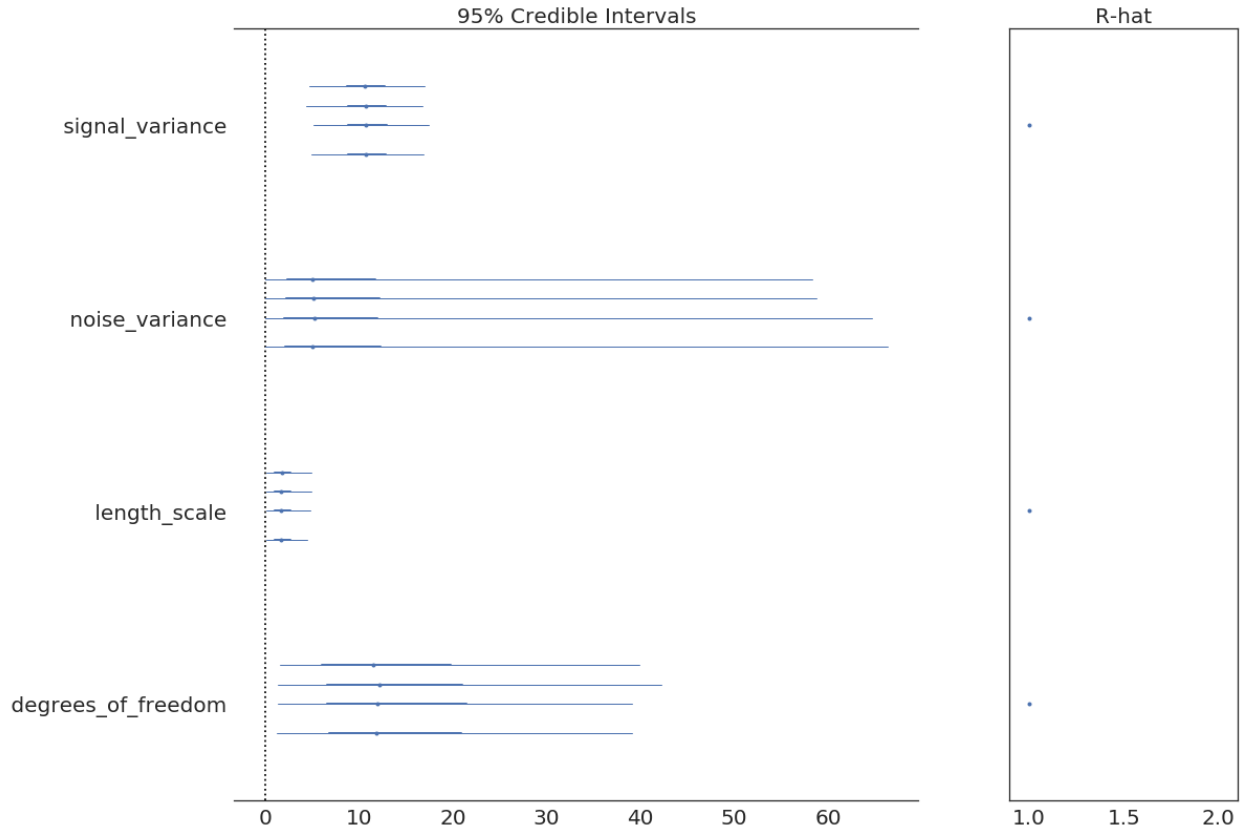
```
In [13]: pm.traceplot(model2.trace, lines = {"signal_variance": signal_variance_true,
                                             "noise_variance": noise_variance_true,
                                             "length_scale": length_scale_true,
                                             "degrees_of_freedom": degrees_of_freedom_true},
                      varnames=["signal_variance", "noise_variance", "length_scale", "degrees_of_free
```




```
In [14]: pm.gelman_rubin(model2.trace, varnames=["signal_variance", "noise_variance", "length_scale",

Out[14]: {'degrees_of_freedom': array([1.00019487]),
          'length_scale': array([[1.00008203]]),
          'noise_variance': array([0.99986753]),
          'signal_variance': array([0.99999439])}

In [15]: pm.energyplot(SPR2.trace);
```




```
In [16]: pm.forestplot(model2.trace, varnames=["signal_variance", "noise_variance", "length_scale", "
```




```
In [20]: pm.summary(model2.trace, varnames=["signal_variance", "length_scale", "noise_variance"])
Out[20]:                          mean          sd  mc_error   hpd_2.5   hpd_97.5  \
```

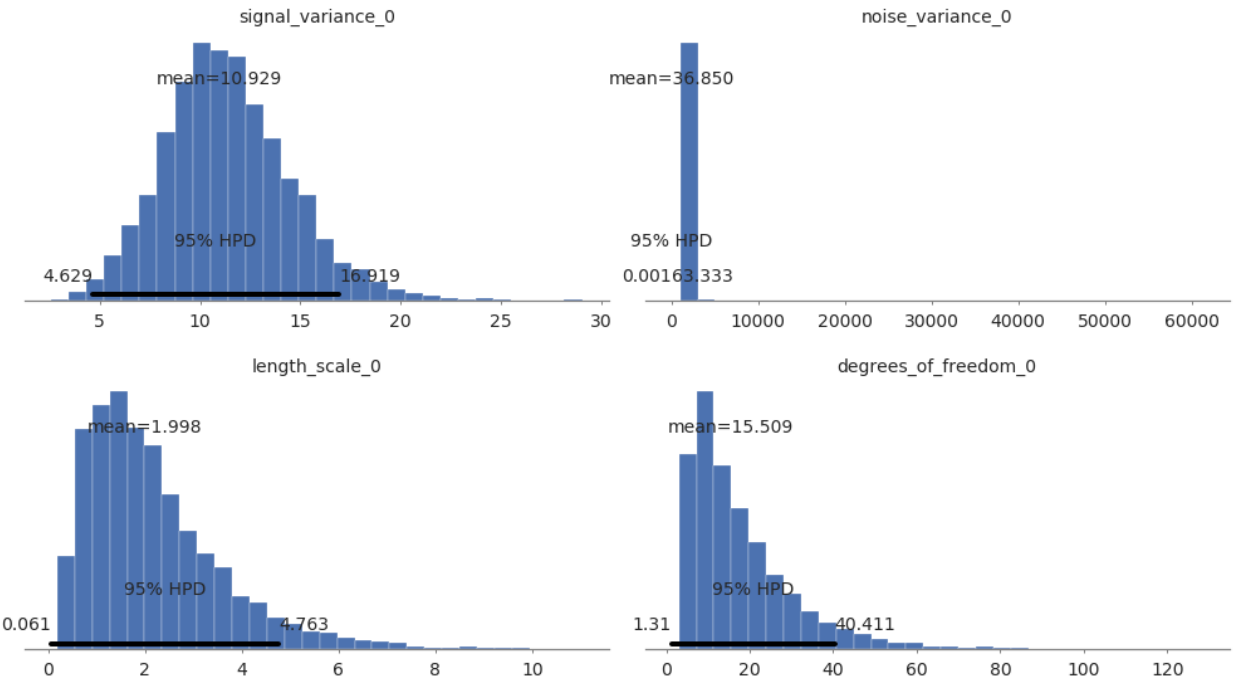
```
signal_variance__0  10.929238    3.144499  0.043256  4.628908  16.918602
length_scale__0_0    1.997809    1.397544  0.012827  0.061045   4.763012
noise_variance__0   36.850186  793.367946  9.474022  0.000534  63.332892

                                n_eff      Rhat
signal_variance__0   4343.913263  0.999994
length_scale__0_0   11392.919787  1.000082
noise_variance__0    7491.212453  0.999868
```

In [21]:
```
# collect the results into a pandas dataframe to display
# "mp" stands for marginal posterior
pd.DataFrame({"Parameter": ["signal_variance", "noise_variance", "length_scale", "degrees_of
              "Predicted Mean Value": [float(model2.trace["length_scale"].mean(axis=0)),
                                       float(model2.trace["signal_variance"].mean(axis=0)),
                                       float(model2.trace["noise_variance"].mean(axis=0)),
                                       float(model2.trace["degrees_of_freedom"].mean(axis=0)
              "True value": [length_scale_true, signal_variance_true,
                             noise_variance_true, degrees_of_freedom_true]})
```

Out[21]:

	Parameter	Predicted Mean Value	True value
0	signal_variance	1.997809	1.0
1	noise_variance	10.929238	3.0
2	length_scale	36.850186	2.0
3	degrees_of_freedom	15.509439	3.0

In [22]:
```
pm.plot_posterior(model2.trace, varnames=["signal_variance", "noise_variance", "length_scale
                  figsize = [14, 8]);
```



In [28]:
```
y_predict2 = model2.predict(X_test)
```

```
100%|| 2000/2000 [00:01<00:00, 1174.91it/s]
```

In [29]:
```
y_predict2
```

Out[29]:
```
array([1.67834026, 1.64158368, 1.53728732, 1.56489496, 1.48686425,
       1.48626043, 1.57801849, 1.5609818 , 1.57435388, 1.76800657,
       1.56198154, 1.49355969, 1.72304612, 1.53818178, 1.68932836,
       1.6059991 , 1.62152421, 1.50726857, 1.92453348, 1.61906672,
```

```
                1.4703559 , 1.49874483, 1.63398678, 1.72795675, 1.62348916,
                1.65877512, 1.78012082, 1.65401634, 1.47100635, 1.51878226,
                1.53634253, 1.66642193, 1.5899548 , 1.62872435, 1.66256587,
                1.67191658, 1.45945213, 1.43421284, 1.52586924, 1.56299994,
                1.79883016, 1.6769178 , 1.52190602, 1.58302155, 1.44959024,
                1.66465733, 1.5804623 , 1.62288222, 1.53714604, 1.80406125])
```

```
In [30]: model2.score(X_test, y_test)
```

```
100%|| 2000/2000 [00:01<00:00, 1254.66it/s]
```

```
Out[30]: -0.0069721816446493
```

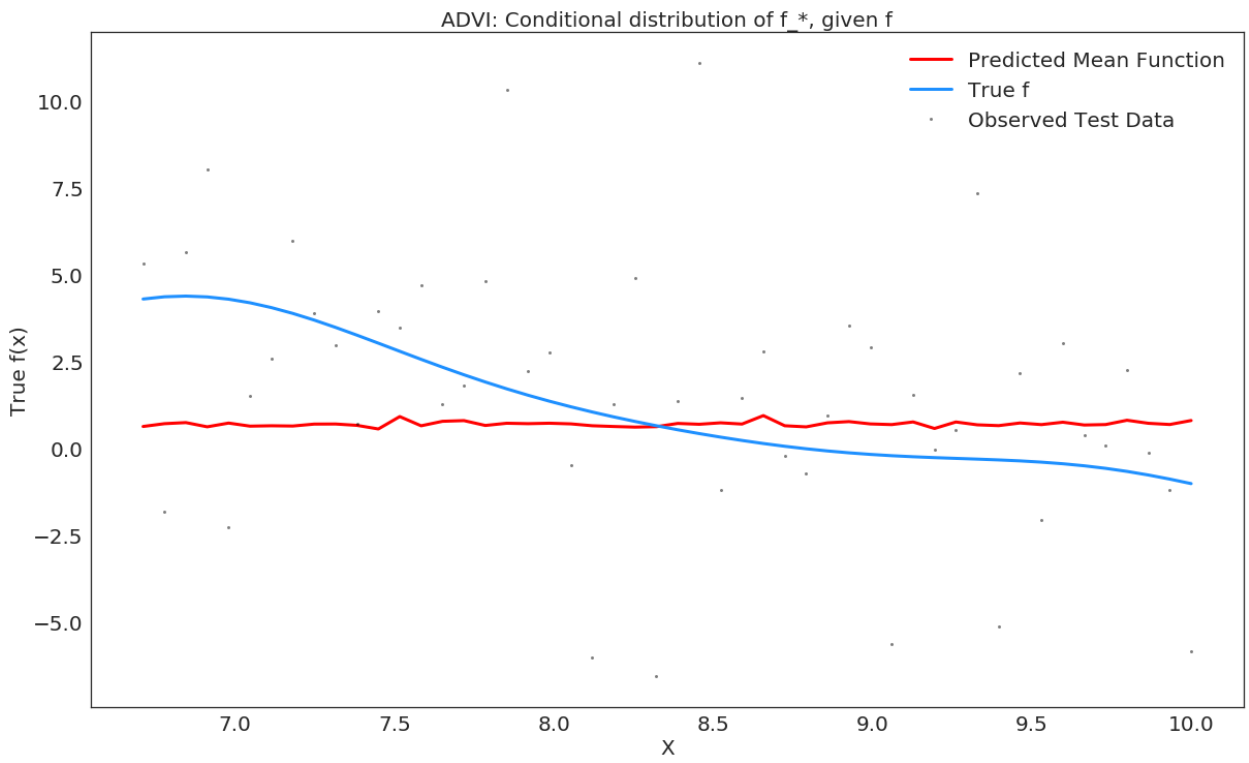
```
In [31]: model2.save('pickle_jar/spr2')
         model2_new = StudentsTProcessRegressor()
         model2_new.load('pickle_jar/spr2')
         model2_new.score(X_test, y_test)
```

```
100%|| 2000/2000 [00:01<00:00, 1104.45it/s]
```

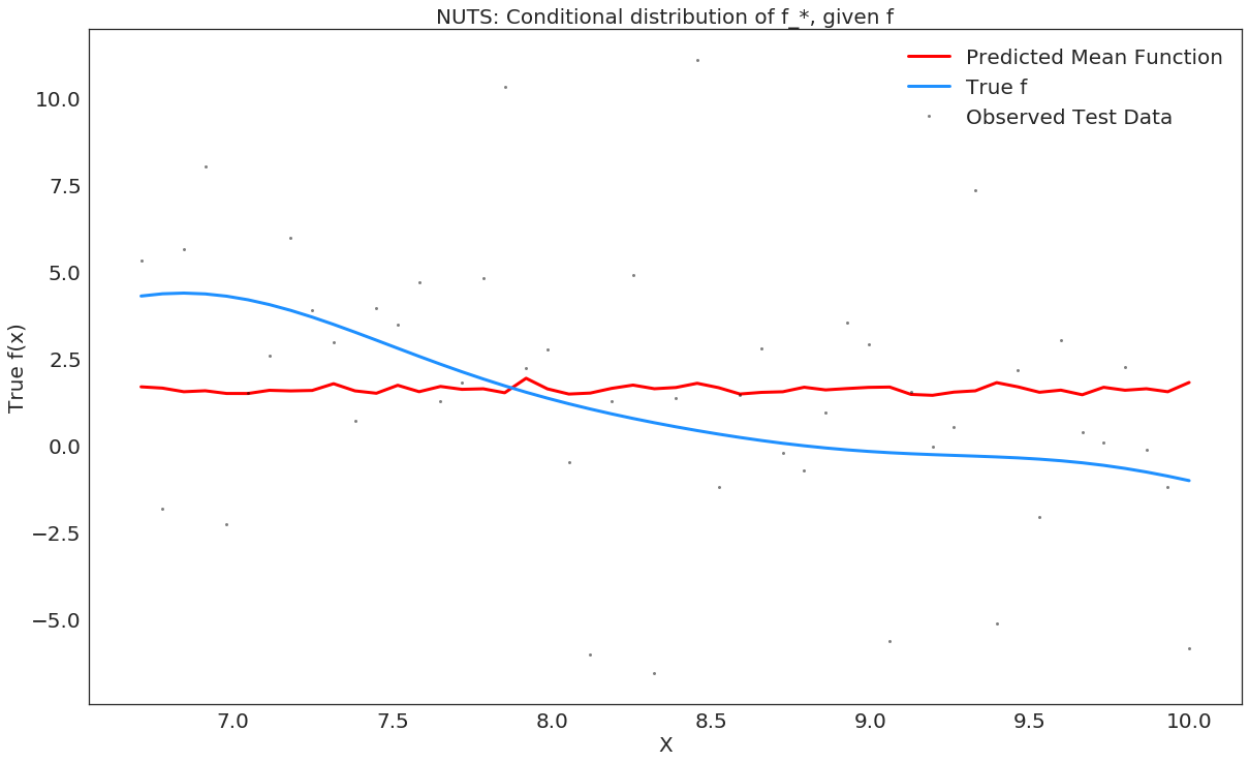
```
Out[31]: 0.0038373353227000306
```

Compare models

```
In [32]: # plot the results
         fig, ax = plt.subplots()

         # plot the samples of the gp posterior
         plt.plot(X_test, y_predict1, "r", lw=3, label="Predicted Mean Function")

         plt.plot(X_train, f_true[100:], "dodgerblue", lw=3, label="True f");
         plt.plot(X_test, y_test, 'ok', ms=3, alpha=0.5, label="Observed Test Data");
         plt.xlabel("X")
         plt.ylabel("True f(x)");
         plt.title("ADVI: Conditional distribution of f_*, given f");
         plt.legend();
```



In [33]:
```
# plot the results
fig, ax = plt.subplots()

# plot the samples of the gp posterior
plt.plot(X_test, y_predict2, "r", lw=3, label="Predicted Mean Function")

plt.plot(X_train, f_true[100:], "dodgerblue", lw=3, label="True f");
plt.plot(X_test, y_test, 'ok', ms=3, alpha=0.5, label="Observed Test Data");
plt.xlabel("X")
plt.ylabel("True f(x)");
plt.title("NUTS: Conditional distribution of f_*, given f");
plt.legend();
```



7.5.6 Sparse Gaussian Process Regression

Let's set some setting for this Jupyter Notebook.

```
In [2]: %matplotlib inline
        from warnings import filterwarnings
        filterwarnings("ignore")
        import os
        os.environ['MKL_THREADING_LAYER'] = 'GNU'
        os.environ['THEANO_FLAGS'] = 'device=cpu'

        import numpy as np
        import pandas as pd
        import pymc3 as pm
        import seaborn as sns
        import matplotlib.pyplot as plt
        np.random.seed(12345)
        rc = {'xtick.labelsize': 20, 'ytick.labelsize': 20, 'axes.labelsize': 20, 'font.size': 20,
              'legend.fontsize': 12.0, 'axes.titlesize': 10, "figure.figsize": [12, 6]}
        sns.set(rc = rc)
        from IPython.core.interactiveshell import InteractiveShell
        InteractiveShell.ast_node_interactivity = "all"
```

Now, let's import the `SparseGaussianProcessRegression` algorithm from the `pymc-learn` package.

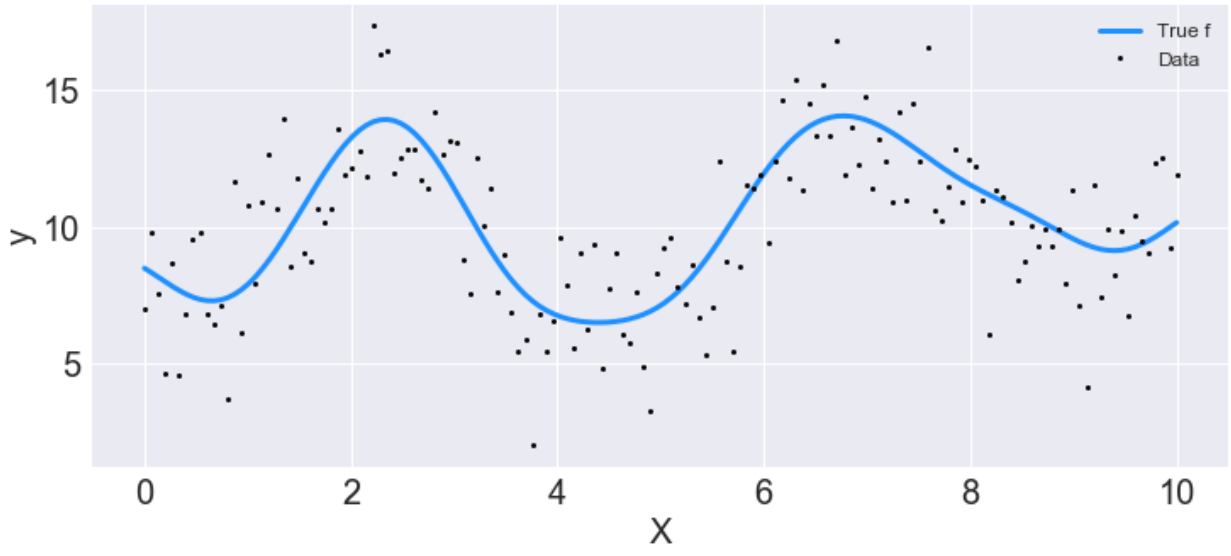
```
In [3]: import pmlearn
        from pmlearn.gaussian_process import SparseGaussianProcessRegressor
        print('Running on pymc-learn v{}'.format(pmlearn.__version__))
```

```
Running on pymc-learn v0.0.1.rc0
```

Step 1: Prepare the data

Generate synthetic data.

```
In [4]: n = 150 # The number of data points
        X = np.linspace(start = 0, stop = 10, num = n)[:, None] # The inputs to the GP, they must be

        # Define the true covariance function and its parameters
        length_scale_true = 1.0
        signal_variance_true = 3.0
        cov_func = signal_variance_true**2 * pm.gp.cov.ExpQuad(1, length_scale_true)

        # A mean function that is zero everywhere
        mean_func = pm.gp.mean.Constant(10)

        # The latent function values are one sample from a multivariate normal
        # Note that we have to call `eval()` because PyMC3 built on top of Theano
        f_true = np.random.multivariate_normal(mean_func(X).eval(),
                                               cov_func(X).eval() + 1e-8*np.eye(n), 1).flatten()

        # The observed data is the latent function plus a small amount of Gaussian distributed noise
        # The standard deviation of the noise is `sigma`
        noise_variance_true = 2.0
        y = f_true + noise_variance_true * np.random.randn(n)

        ## Plot the data and the unobserved latent function
        fig = plt.figure(figsize=(12,5))
        ax = fig.gca()
        ax.plot(X, f_true, "dodgerblue", lw=3, label="True f");
        ax.plot(X, y, 'ok', ms=3, label="Data");
        ax.set_xlabel("X"); ax.set_ylabel("y"); plt.legend();
```




```
In [5]: from sklearn.model_selection import train_test_split
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

Step 2: Instantiate a model

```
In [6]: model = SparseGaussianProcessRegressor()
```

Step 3: Perform Inference

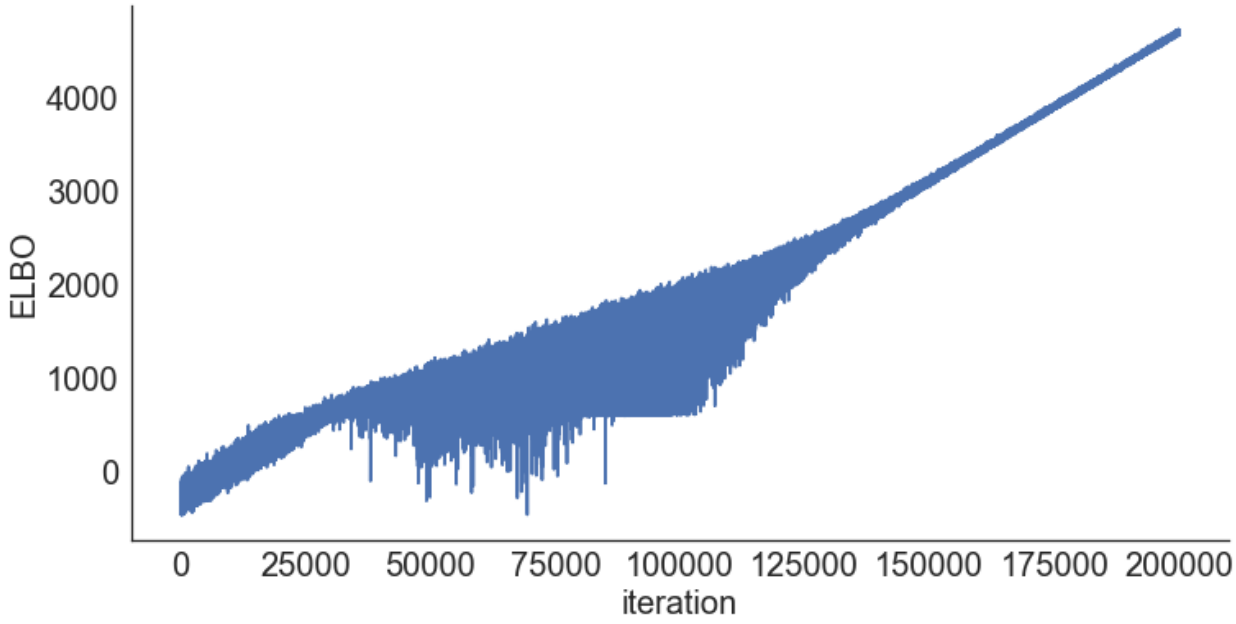
```
In [7]: model.fit(X_train, y_train)
```

```
Average Loss = -4,669.3: 100%|| 200000/200000 [06:12<00:00, 536.41it/s]
Finished [100%]: Average Loss = -4,669.6
```

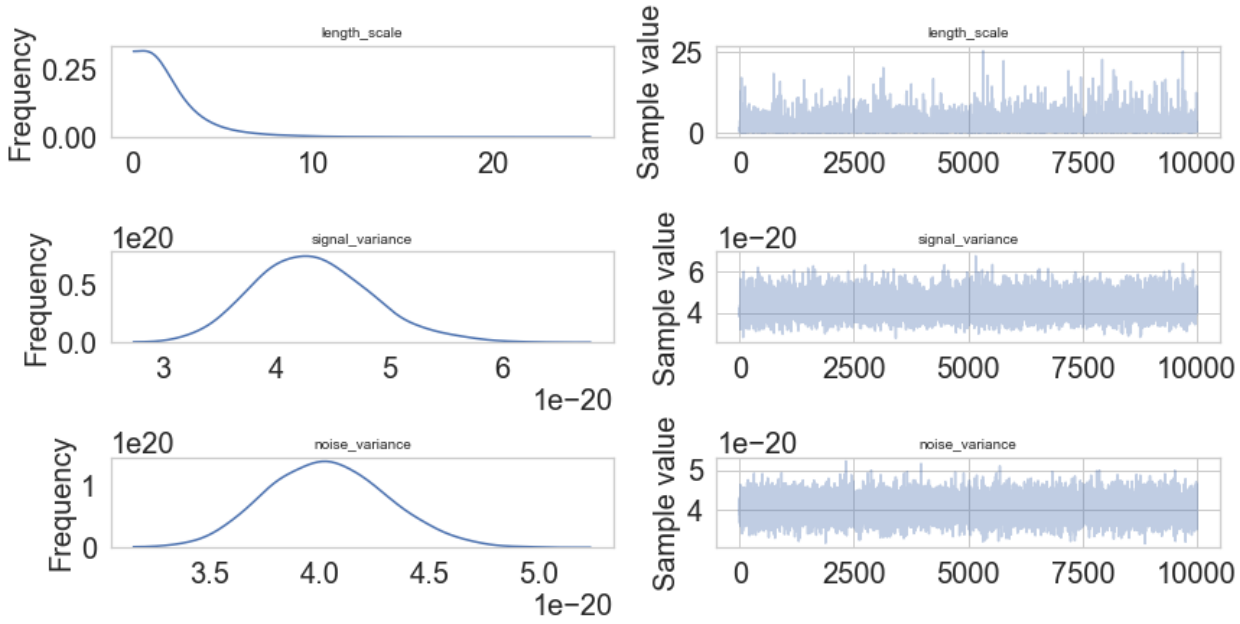
```
Out[7]: SparseGaussianProcessRegressor(prior_mean=0.0)
```

Step 4: Diagnose convergence

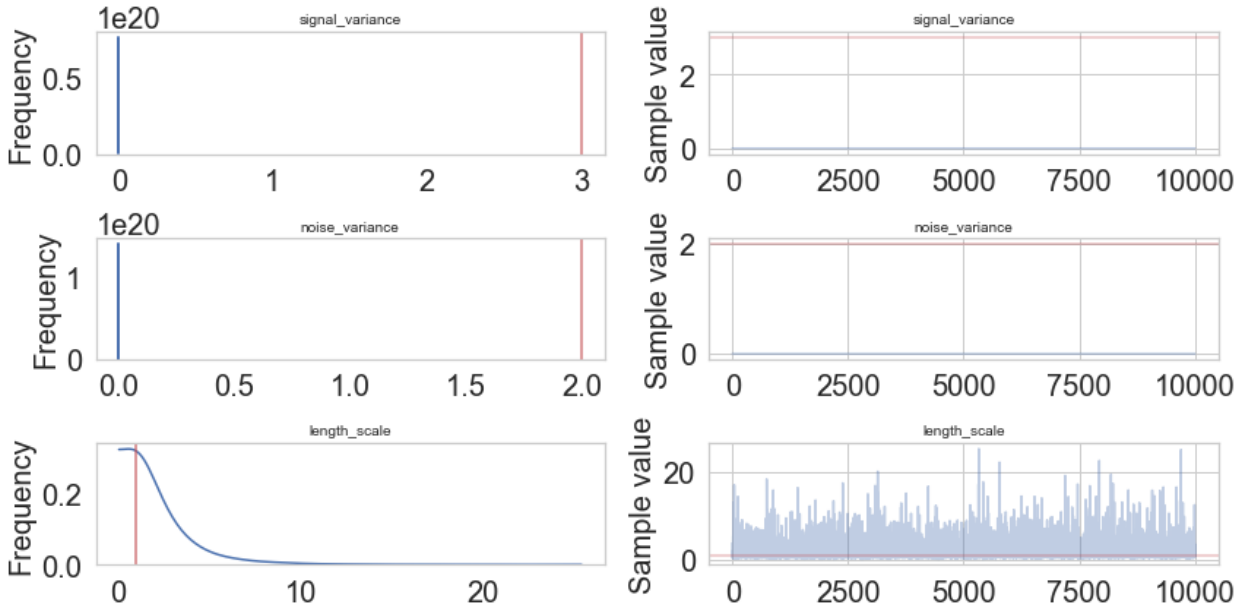
```
In [8]: model.plot_elbo()
```



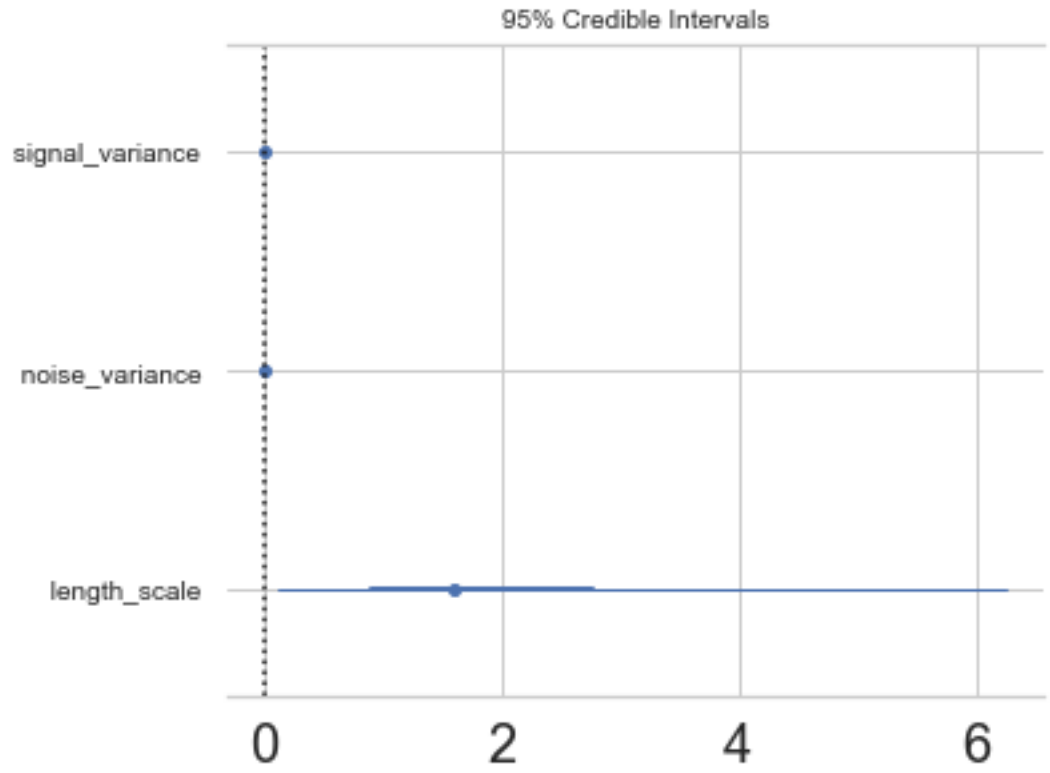

```
In [8]: pm.traceplot(model.trace);
```



```
In [9]: pm.traceplot(model.trace, lines = {"signal_variance": signal_variance_true,
                                           "noise_variance": noise_variance_true,
                                           "length_scale": length_scale_true},
                     varnames=["signal_variance", "noise_variance", "length_scale"]);
```




```
In [11]: pm.forestplot(model.trace, varnames=["signal_variance", "noise_variance", "length_scale"]);
```



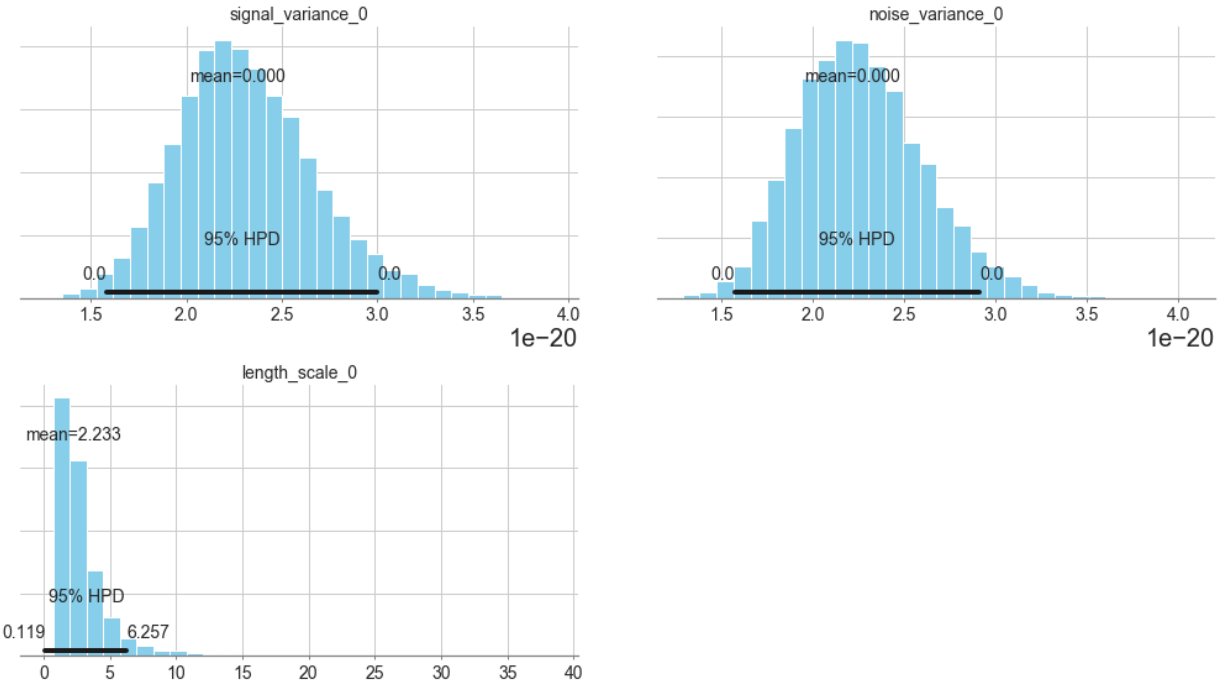
Step 5: Critize the model

```
In [12]: pm.summary(model.trace, varnames=["signal_variance", "length_scale", "noise_variance"])
```

Out[12]:

	mean	sd	mc_error	hpd_2.5	hpd_97.5
signal_variance__0	2.266128e-20	3.632205e-21	3.731201e-23	1.578536e-20	2.999736e-20
length_scale__0_0	2.233311e+00	2.182089e+00	2.042519e-02	1.187429e-01	6.257235e+00
noise_variance__0	2.216509e-20	3.491209e-21	3.041214e-23	1.571128e-20	2.914439e-20

```
In [13]: pm.plot_posterior(model.trace, varnames=["signal_variance", "noise_variance", "length_scale"
                         figsize = [14, 8]);
```




```
In [8]: # collect the results into a pandas dataframe to display
        # "mp" stands for marginal posterior
        pd.DataFrame({"Parameter": ["length_scale", "signal_variance", "noise_variance"],
                      "Predicted Mean Value": [float(model.trace["length_scale"].mean(axis=0)),
                                               float(model.trace["signal_variance"].mean(axis=0)),
                                               float(model.trace["noise_variance"].mean(axis=0))],
                      "True value": [length_scale_true, signal_variance_true, noise_variance_true]})
```

Out[8]:

	Parameter	Predicted Mean Value	True value
0	length_scale	2.182521	1.0
1	signal_variance	9.261435	3.0
2	noise_variance	0.002241	2.0

Step 6: Use the model for prediction

```
In [ ]: y_predict1 = model.predict(X_test)
```

```
In [ ]: y_predict1
```

```
In [ ]: model.score(X_test, y_test)

In [ ]: model.save('pickle_jar/sgpr')
```

Use already trained model for prediction

```
In [ ]: model_new = SparseGaussianProcessRegressor()
        model_new.load('pickle_jar/sgpr')
        model_new.score(X_test, y_test)
```

Multiple Features

```
In [ ]: num_pred = 2
        X = np.random.randn(1000, num_pred)
        noise = 2 * np.random.randn(1000,)
        Y = X.dot(np.array([4, 5])) + 3 + noise

In [ ]: y = np.squeeze(Y)

In [ ]: model_big = SparseGaussianProcessRegressor()

In [ ]: model_big.fit(X, y, inference_args={"n" : 1000})

In [ ]: pm.summary(model_big.trace, varnames=["signal_variance", "length_scale", "noise_variance"])
```

MCMC

```
In [8]: model2 = SparseGaussianProcessRegressor()
        model2.fit(X_train, y_train, inference_type='nuts')
```

```
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [f_rotated_, noise_variance_log__, signal_variance_log__, length_scale_log__]
100%|| 1500/1500 [00:24<00:00, 60.20it/s]
There were 92 divergences after tuning. Increase `target_accept` or reparameterize.
There were 88 divergences after tuning. Increase `target_accept` or reparameterize.
There were 87 divergences after tuning. Increase `target_accept` or reparameterize.
There were 40 divergences after tuning. Increase `target_accept` or reparameterize.
The number of effective samples is smaller than 10% for some parameters.
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [f_rotated_, noise_variance_log__, signal_variance_log__, length_scale_log__]
100%|| 4000/4000 [11:41<00:00,  5.70it/s]
There were 4 divergences after tuning. Increase `target_accept` or reparameterize.
There were 2 divergences after tuning. Increase `target_accept` or reparameterize.
There were 1 divergences after tuning. Increase `target_accept` or reparameterize.
There were 1 divergences after tuning. Increase `target_accept` or reparameterize.
```

```
Out[8]: GaussianProcessRegression()
```

```
In [18]: pm.traceplot(model2.trace, lines = {"signal_variance": signal_variance_true,
                                              "noise_variance": noise_variance_true,
                                              "length_scale": length_scale_true},
                       varnames=["signal_variance", "noise_variance", "length_scale"]);
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
In [19]: pm.gelman_rubin(model2.trace, varnames=["signal_variance", "noise_variance", "length_scale"]
```

```
Out[19]: {'signal_variance': array([ 1.00134827]),
          'noise_variance': array([ 0.99982997]),
          'length_scale': array([[ 0.9997668]])}

In [22]: pm.energyplot(model2.trace);

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

In [21]: pm.forestplot(model2.trace, varnames=["signal_variance", "noise_variance", "length_scale"]);

In [10]: pm.summary(model2.trace, varnames=["signal_variance", "length_scale", "noise_variance"])

Out[10]:                          mean        sd  mc_error   hpd_2.5   hpd_97.5  \
         signal_variance__0  3.354521  5.059969  0.072611  0.004388  10.494186
         length_scale__0_0   2.004001  1.446405  0.013286  0.033810   4.790922
         noise_variance__0   2.544328  0.264045  0.003021  2.074630   3.086981

                                  n_eff      Rhat
         signal_variance__0   3949.565916  1.001348
         length_scale__0_0   12131.009636  0.999767
         noise_variance__0    8803.802924  0.999830

In [11]: # collect the results into a pandas dataframe to display
         # "mp" stands for marginal posterior
         pd.DataFrame({"Parameter": ["length_scale", "signal_variance", "noise_variance"],
                       "Predicted Mean Value": [float(model2.trace["length_scale"].mean(axis=0)),
                                                float(model2.trace["signal_variance"].mean(axis=0)),
                                                float(model2.trace["noise_variance"].mean(axis=0))],
                       "True value": [length_scale_true, signal_variance_true, noise_variance_true]})

Out[11]:          Parameter  Predicted Mean Value  True value
         0     length_scale              2.004001         1.0
         1  signal_variance              3.354521         3.0
         2   noise_variance              2.544328         2.0

In [12]: pm.plot_posterior(model2.trace, varnames=["signal_variance", "noise_variance", "length_scale
                           figsize = [14, 8]);

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

In [28]: y_predict2 = model2.predict(X_test)

100%|| 2000/2000 [00:01<00:00, 1332.79it/s]

In [29]: y_predict2

Out[29]: array([ 0.71831924,  0.70266214,  0.74034292,  0.73223746,  0.76798942,
                 0.78039904,  0.80198739,  0.77559783,  0.74532885,  0.75839183,
                 0.69163726,  0.6490964 ,  0.71534946,  0.65845406,  0.66052402,
                 0.80801464,  0.69148553,  0.61070685,  0.69928683,  0.75866764,
                 0.6620472 ,  0.73977574,  0.70854909,  0.70340364,  0.70960481,
                 0.69097856,  0.69340258,  0.72408786,  0.81266196,  0.79486012,
                 0.72997809,  0.66805751,  0.72690218,  0.71025724,  0.72545681,
                 0.69062513,  0.75047548,  0.64446808,  0.78133024,  0.69365793,
                 0.78675961,  0.7909775 ,  0.66224847,  0.67357815,  0.82613138,
                 0.76196312,  0.76742   ,  0.67757641,  0.67067013,  0.70072039])

In [30]: model2.score(X_test, y_test)

100%|| 2000/2000 [00:01<00:00, 1464.52it/s]

Out[30]: -0.011313490552906202
```

```
In [ ]: model2.save('pickle_jar/')
        model2_new = SparseGaussianProcessRegressor()
        model2_new.load('pickle_jar/')
        model2_new.score(X_test, y_test)
```

7.5.7 Multilayer Perceptron Classifier

Let's set some setting for this Jupyter Notebook.

```
In [2]: %matplotlib inline
        from warnings import filterwarnings
        filterwarnings("ignore")
        import os
        os.environ['MKL_THREADING_LAYER'] = 'GNU'
        os.environ['THEANO_FLAGS'] = 'device=cpu'

        import numpy as np
        import pandas as pd
        import pymc3 as pm
        import seaborn as sns
        import matplotlib.pyplot as plt
        np.random.seed(12345)
        rc = {'xtick.labelsize': 20, 'ytick.labelsize': 20, 'axes.labelsize': 20, 'font.size': 20,
              'legend.fontsize': 12.0, 'axes.titlesize': 10, "figure.figsize": [12, 6]}
        sns.set(rc = rc)
        from IPython.core.interactiveshell import InteractiveShell
        InteractiveShell.ast_node_interactivity = "all"
```

Now, let's import the `MLPClassifier` algorithm from the `pymc-learn` package.

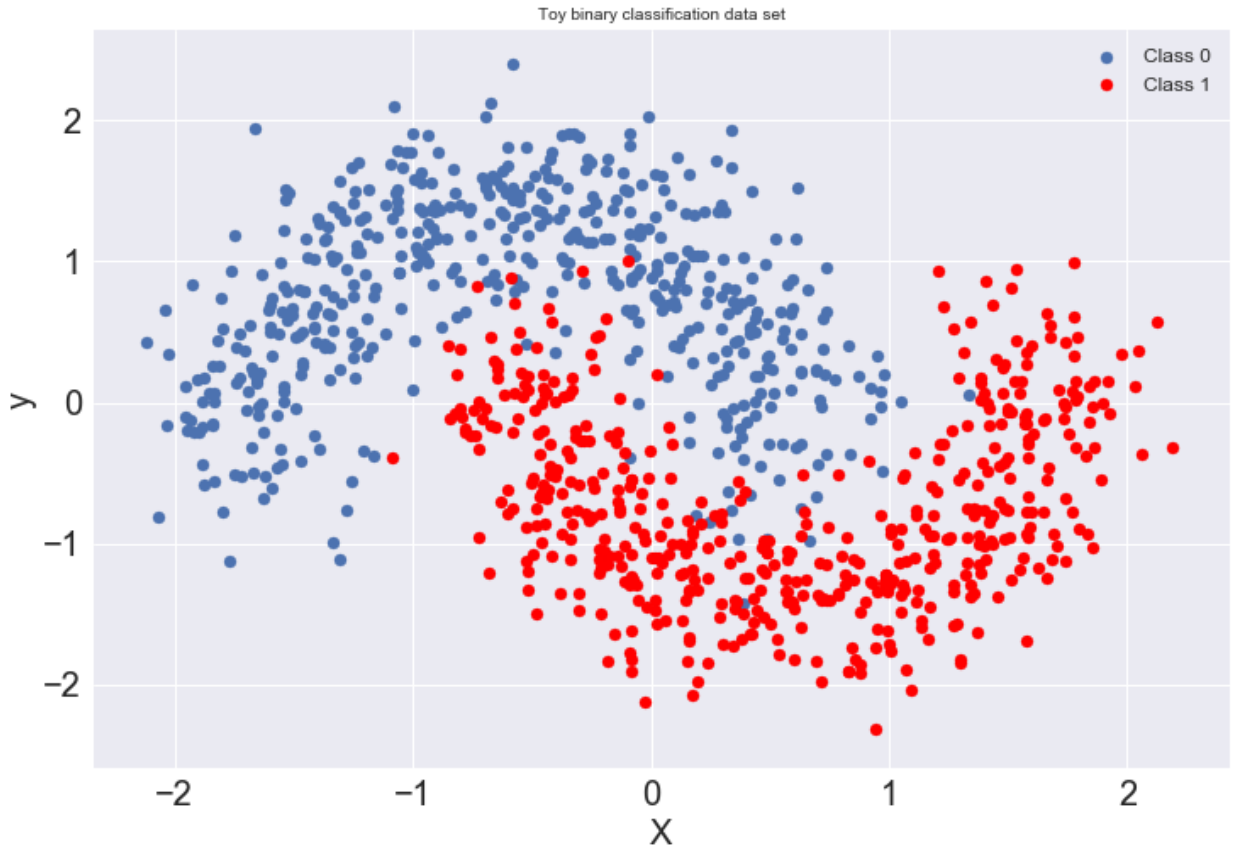
```
In [3]: import pmlearn
        from pmlearn.neural_network import MLPClassifier
        print('Running on pymc-learn v{}'.format(pmlearn.__version__))
```

```
Running on pymc-learn v0.0.1.rc0
```

Step 1: Prepare the data

Generate synthetic data.

```
In [4]: from sklearn.datasets import make_moons
        from sklearn.preprocessing import scale
        import theano
        floatX = theano.config.floatX

        X, y = make_moons(noise=0.2, random_state=0, n_samples=1000)
        X = scale(X)
        X = X.astype(floatX)
        y = y.astype(floatX)

        ## Plot the data
        fig, ax = plt.subplots(figsize=(12, 8))
        ax.scatter(X[y==0, 0], X[y==0, 1], label='Class 0')
        ax.scatter(X[y==1, 0], X[y==1, 1], color='r', label='Class 1')
        sns.despine(); ax.legend()
        ax.set(xlabel='X', ylabel='y', title='Toy binary classification data set');
```




```
In [5]: from sklearn.model_selection import train_test_split
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

Step 2: Instantiate a model

```
In [6]: model = MLPClassifier()
```

Step 3: Perform Inference

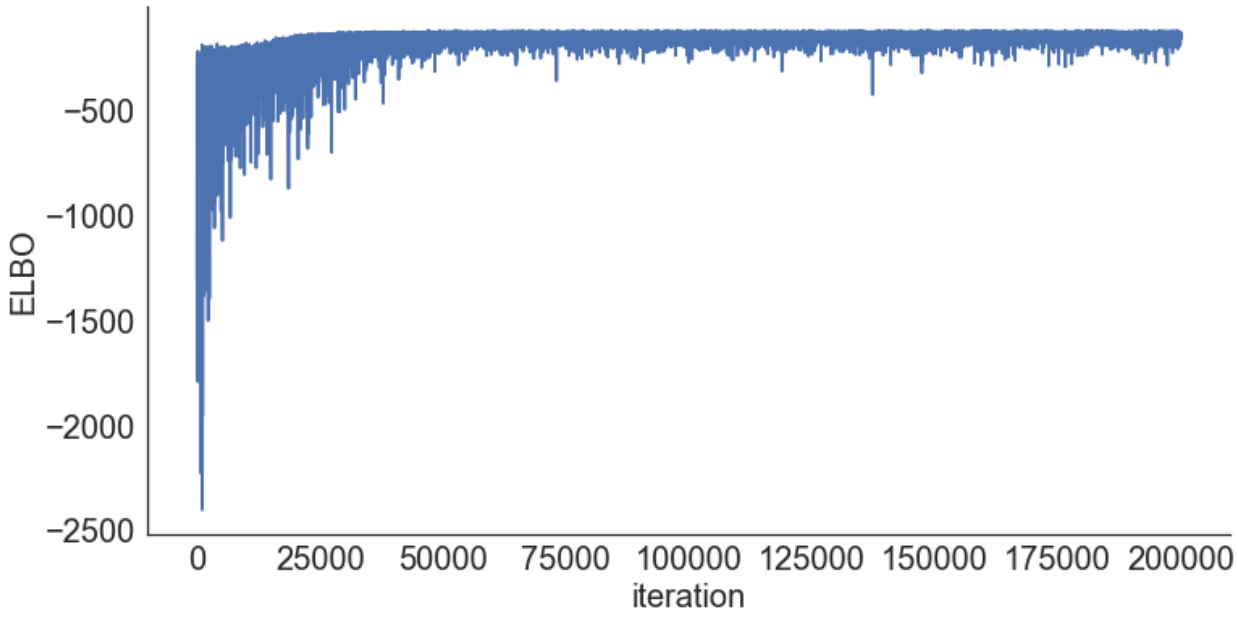
```
In [7]: model.fit(X_train, y_train)
```

```
Average Loss = 140.51: 100%|| 200000/200000 [02:46<00:00, 1203.69it/s]
Finished [100%]: Average Loss = 140.52
```

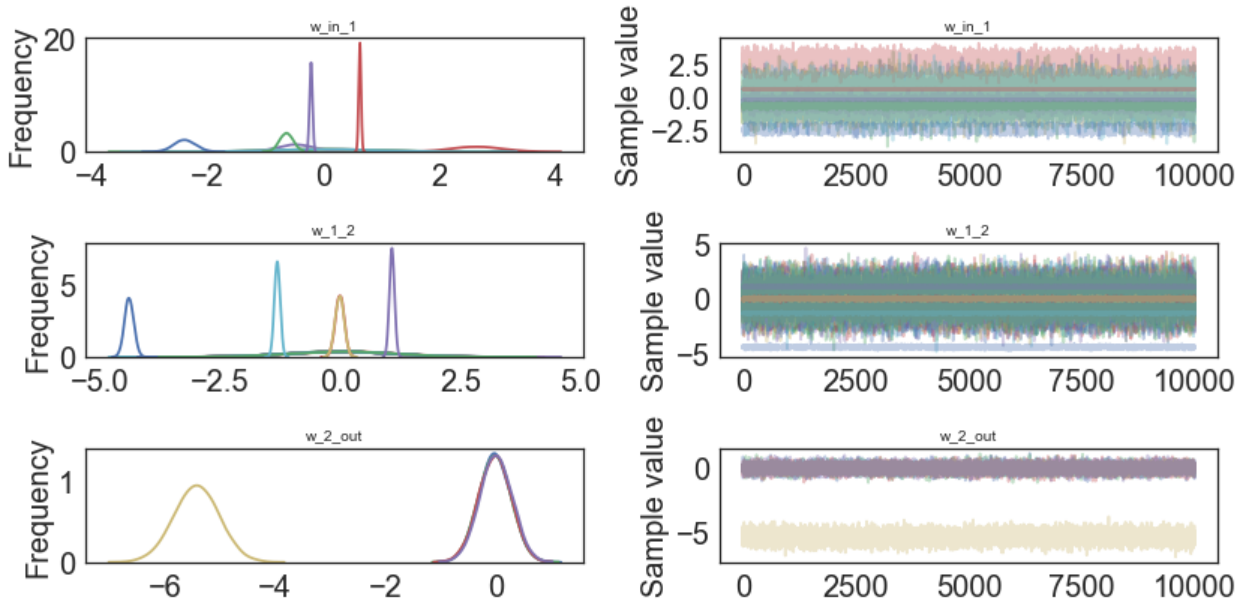
```
Out[7]: MLPClassifier(n_hidden=5)
```

Step 4: Diagnose convergence

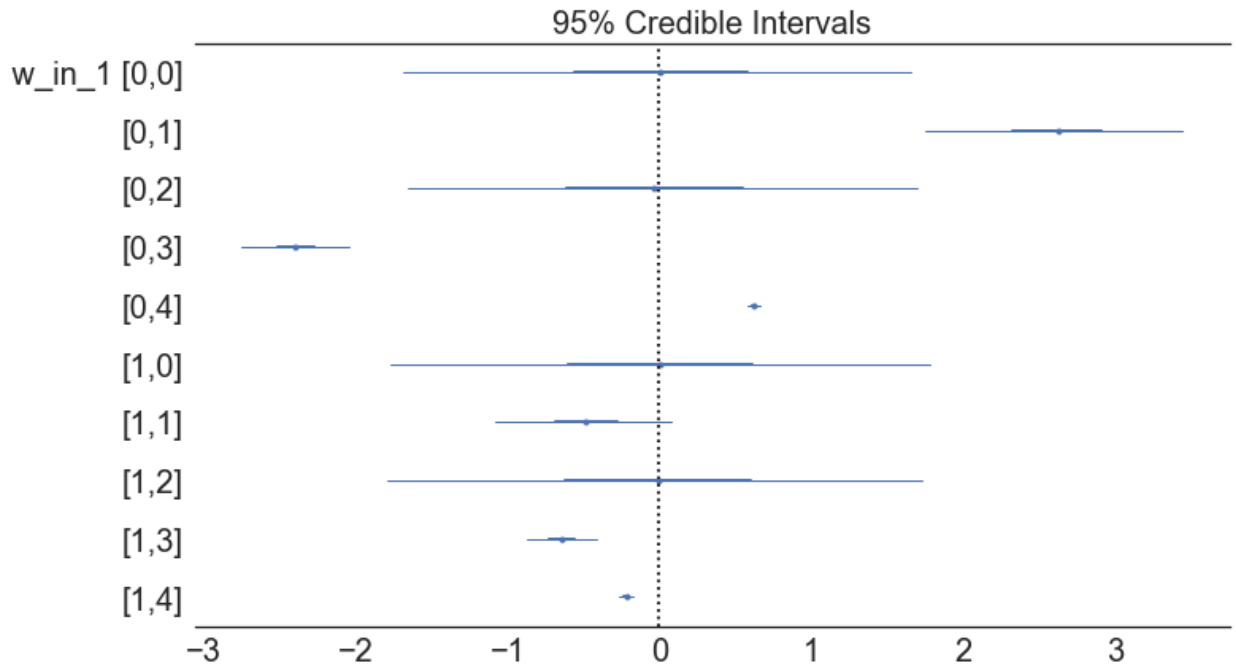
```
In [8]: model.plot_elbo()
```



In [9]: pm.traceplot(model.trace);



In [10]: pm.forestplot(model.trace, varnames=["w_in_1"]);



Step 5: Critize the model

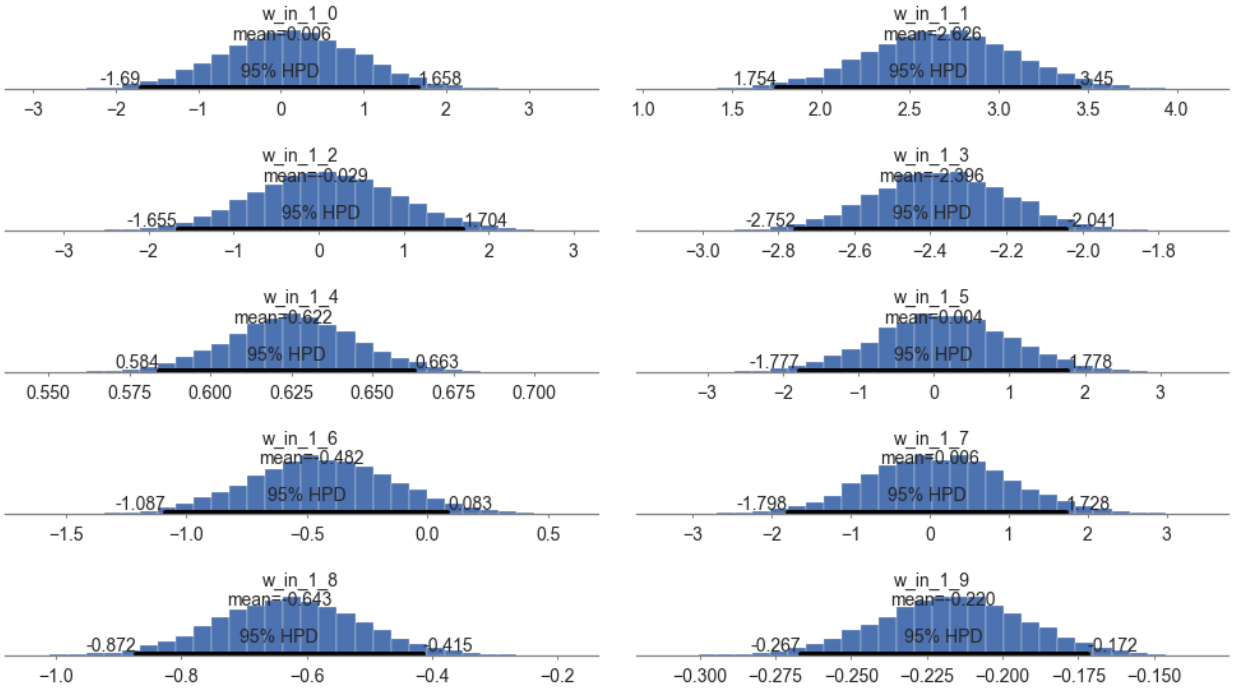
```
In [11]: pm.summary(model.trace)
```

Out[11]:

	mean	sd	mc_error	hpd_2.5	hpd_97.5
w_in_1__0_0	0.005568	0.853588	0.008590	-1.690336	1.657664
w_in_1__0_1	2.625947	0.430473	0.004169	1.754085	3.449592
w_in_1__0_2	-0.029436	0.857784	0.007223	-1.654908	1.703846
w_in_1__0_3	-2.395630	0.181738	0.001690	-2.751851	-2.040889
w_in_1__0_4	0.622473	0.020184	0.000191	0.584093	0.663157
w_in_1__1_0	0.004472	0.913144	0.009296	-1.776564	1.778230
w_in_1__1_1	-0.481947	0.301163	0.003205	-1.086732	0.082603
w_in_1__1_2	0.005991	0.899893	0.007878	-1.798096	1.727599
w_in_1__1_3	-0.643017	0.116311	0.001178	-0.872403	-0.415040
w_in_1__1_4	-0.219521	0.024275	0.000220	-0.266726	-0.171858
w_1_2__0_0	0.047623	1.073161	0.011119	-2.021469	2.151871
w_1_2__0_1	-0.011368	1.077858	0.009985	-2.194813	2.021219
w_1_2__0_2	-0.031757	1.054057	0.009998	-2.083095	2.018043
w_1_2__0_3	0.020650	1.072316	0.010139	-2.107038	2.109442
w_1_2__0_4	-0.004587	0.089081	0.000801	-0.180803	0.168265
w_1_2__1_0	-0.046195	0.995340	0.010568	-2.014609	1.858622
w_1_2__1_1	-0.037873	1.019444	0.009749	-2.040949	1.960829
w_1_2__1_2	-0.021888	1.032725	0.011500	-2.130446	1.901217
w_1_2__1_3	0.017286	1.002461	0.009124	-2.031078	1.903586
w_1_2__1_4	1.081594	0.051777	0.000545	0.979248	1.182610
w_1_2__2_0	0.035723	1.060417	0.009704	-2.036665	2.086344
w_1_2__2_1	0.026448	1.068230	0.010686	-2.102493	2.071060
w_1_2__2_2	-0.029435	1.035190	0.010280	-1.957489	2.077871
w_1_2__2_3	-0.000834	1.059046	0.011346	-2.108175	1.998687
w_1_2__2_4	-0.007699	0.090879	0.000956	-0.183102	0.169459
w_1_2__3_0	-0.032409	1.056289	0.010256	-2.207693	1.981262
w_1_2__3_1	0.034940	1.027592	0.009972	-2.005263	2.052578
w_1_2__3_2	0.008883	1.029169	0.010479	-2.062519	1.946606
w_1_2__3_3	-0.008496	1.036160	0.010453	-2.109220	1.909782

```
w_1_2__3_4  -1.304680  0.057650  0.000542 -1.413446 -1.188361
w_1_2__4_0  -0.011010  1.082071  0.010123 -2.120924  2.138763
w_1_2__4_1  -0.034882  1.062334  0.009535 -2.165315  2.008977
w_1_2__4_2  -0.003337  1.065358  0.010463 -2.068589  2.064908
w_1_2__4_3   0.045525  1.057940  0.010719 -2.045978  2.035685
w_1_2__4_4  -4.385587  0.094443  0.000898 -4.570870 -4.199793
w_2_out__0  -0.010038  0.286819  0.002983 -0.563166  0.559095
w_2_out__1  -0.013392  0.286911  0.002836 -0.572484  0.545143
w_2_out__2  -0.013202  0.291041  0.003212 -0.586738  0.550436
w_2_out__3   0.013324  0.289007  0.003026 -0.581552  0.551274
w_2_out__4  -5.422810  0.401398  0.003642 -6.218766 -4.651419
```

In [12]:
```
pm.plot_posterior(model.trace, varnames=["w_in_1"],
                  figsize = [14, 8]);
```



Step 6: Use the model for prediction

In [13]:
```
y_pred = model.predict(X_test)
```

```
100%|| 2000/2000 [00:00<00:00, 2578.43it/s]
```

In [14]:
```
y_pred
```

Out[14]:
```
array([False,  True,  True, False, False,  True,  True,  True,  True,
       False, False,  True,  True, False,  True,  True,  True, False,
       False,  True,  True,  True,  True,  True, False,  True, False,
       False, False, False, False,  True, False,  True,  True, False,
        True,  True,  True, False, False, False,  True,  True, False,
       False, False,  True, False, False, False,  True,  True, False,
        True,  True,  True, False,  True, False,  True, False,  True,
       False, False,  True,  True, False, False,  True, False, False,
       False,  True,  True,  True,  True,  True,  True, False, False,
       False, False, False,  True,  True,  True, False, False, False,
        True,  True,  True,  True, False,  True, False,  True,  True,
       False,  True,  True,  True,  True, False, False, False,  True,
```

```
                False, False, False, False,  True, False, False,  True, False,
                 True,  True, False, False,  True,  True,  True, False,  True,
                 True,  True, False,  True, False,  True,  True, False, False,
                 True,  True,  True,  True, False,  True,  True, False, False,
                 True, False,  True, False,  True,  True, False,  True, False,
                 True,  True,  True, False, False, False, False,  True, False,
                 True, False,  True, False, False, False, False,  True,  True,
                False,  True, False,  True, False,  True, False,  True,  True,
                 True,  True, False,  True,  True,  True, False, False,  True,
                False,  True,  True, False,  True, False,  True, False, False,
                 True, False, False,  True,  True, False,  True,  True,  True,
                False,  True,  True,  True,  True,  True, False,  True, False,
                 True,  True, False,  True, False, False,  True,  True, False,
                False,  True,  True,  True, False, False,  True,  True,  True,
                False,  True, False, False,  True, False, False,  True, False,
                False,  True,  True, False, False,  True,  True, False,  True,
                 True, False, False, False,  True, False, False, False, False,
                False, False, False,  True, False,  True, False, False, False,
                 True, False,  True, False, False,  True, False, False,  True,
                False,  True,  True,  True, False,  True,  True,  True,  True,
                 True, False, False, False,  True, False,  True, False, False,
                False, False, False], dtype=bool)
```

```
In [15]: model.score(X_test, y_test)
```

```
100%|| 2000/2000 [00:00<00:00, 2722.33it/s]
```

```
Out[15]: 0.95999999999999996
```

```
In [16]: model.save('pickle_jar/mlpc')
```

Use already trained model for prediction

```
In [17]: model_new = MLPClassifier()
         model_new.load('pickle_jar/mlpc')
         model_new.score(X_test, y_test)
```

```
100%|| 2000/2000 [00:00<00:00, 2619.50it/s]
```

```
Out[17]: 0.95999999999999996
```

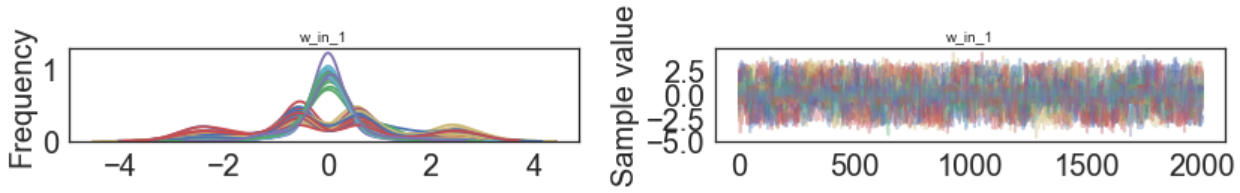
MCMC

```
In [18]: model2 = MLPClassifier()
         model2.fit(X_train, y_train, inference_type='nuts')
```

```
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [w_2_out, w_1_2, w_in_1]
100%|| 2500/2500 [04:25<00:00,  9.42it/s]
There were 125 divergences after tuning. Increase `target_accept` or reparameterize.
There were 228 divergences after tuning. Increase `target_accept` or reparameterize.
There were 210 divergences after tuning. Increase `target_accept` or reparameterize.
There were 32 divergences after tuning. Increase `target_accept` or reparameterize.
The estimated number of effective samples is smaller than 200 for some parameters.
```

```
Out[18]: MLPClassifier(n_hidden=5)
```

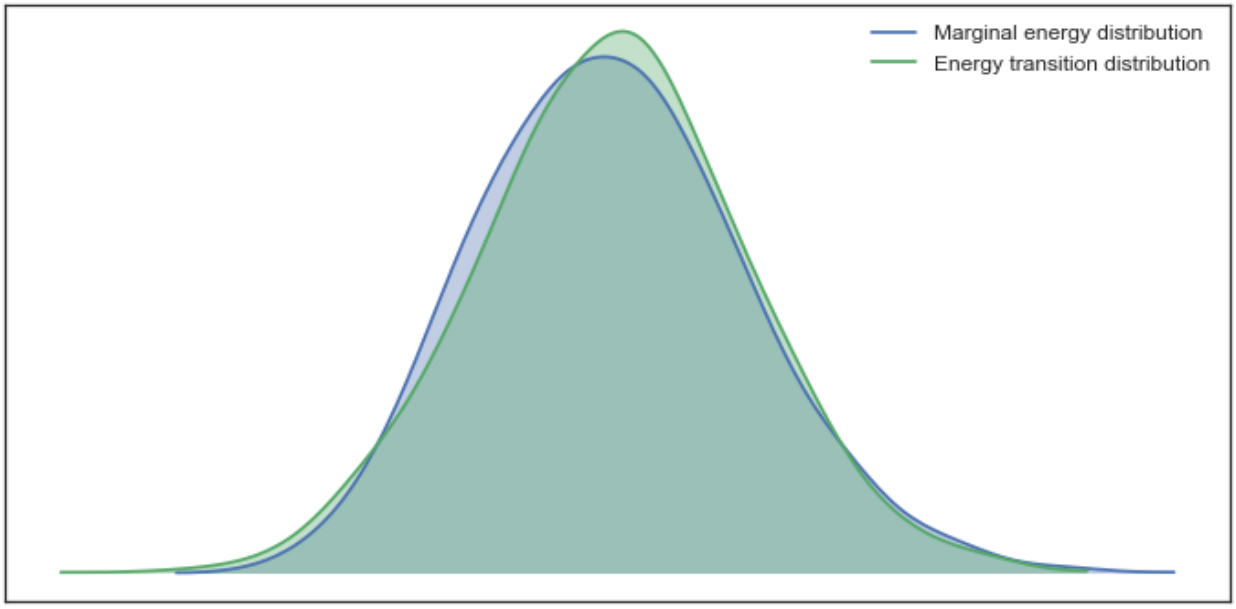
```
In [19]: pm.traceplot(model2.trace, varnames=["w_in_1"]);
```



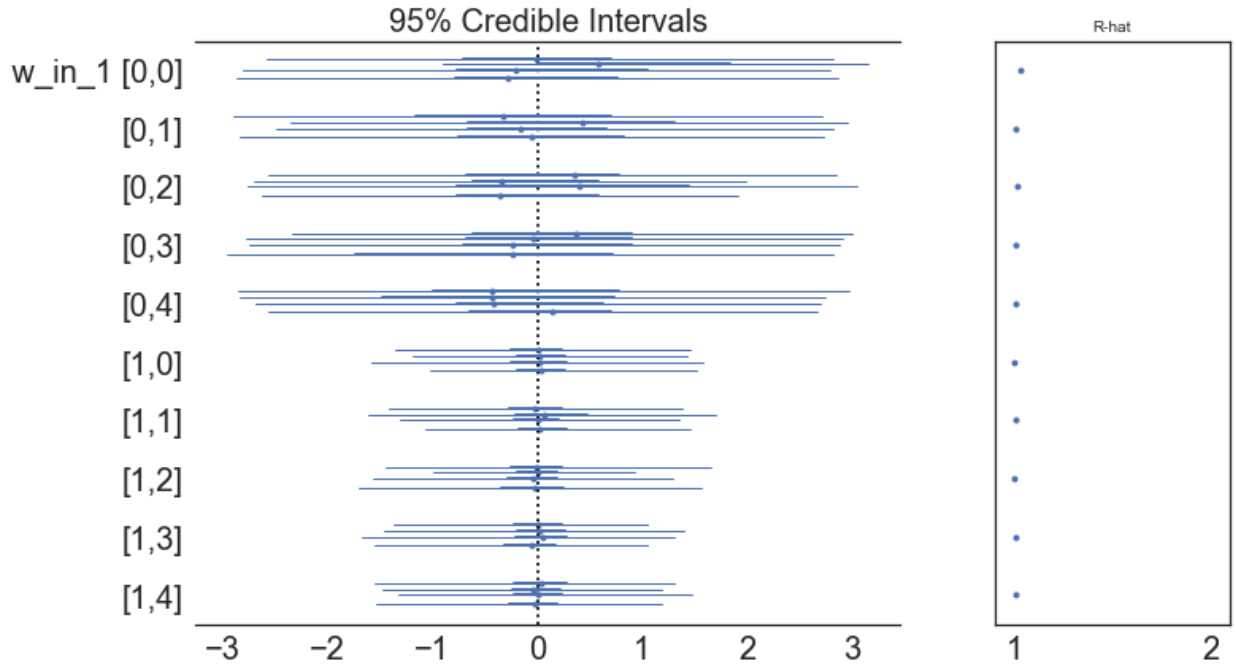
In [20]: pm.gelman_rubin(model2.trace)

```
Out[20]: {'w_in_1': array([[ 1.03392059,  1.00889386,  1.0116798 ,  1.00952281,  1.00310832],
                [ 1.00180089,  1.00662138,  1.00244567,  1.00753298,  1.00338383]]),
          'w_1_2': array([[ 0.99999921,  1.00373929,  1.00043873,  1.00022153,  1.00150073],
                [ 1.00202154,  1.00028483,  1.00173403,  1.00384901,  1.00022611],
                [ 1.00035073,  1.00026924,  1.00524066,  1.00006522,  1.00168698],
                [ 1.00206691,  1.00377702,  1.00243599,  1.00069978,  1.00472955],
                [ 0.99978974,  0.99992665,  1.00151647,  1.00214903,  1.00018014]]),
          'w_2_out': array([ 1.01048089,  1.0018095 ,  1.00558228,  1.00216195,  1.00162127])}
```

In [21]: pm.energyplot(model2.trace);



In [22]: pm.forestplot(model2.trace, varnames=["w_in_1"]);



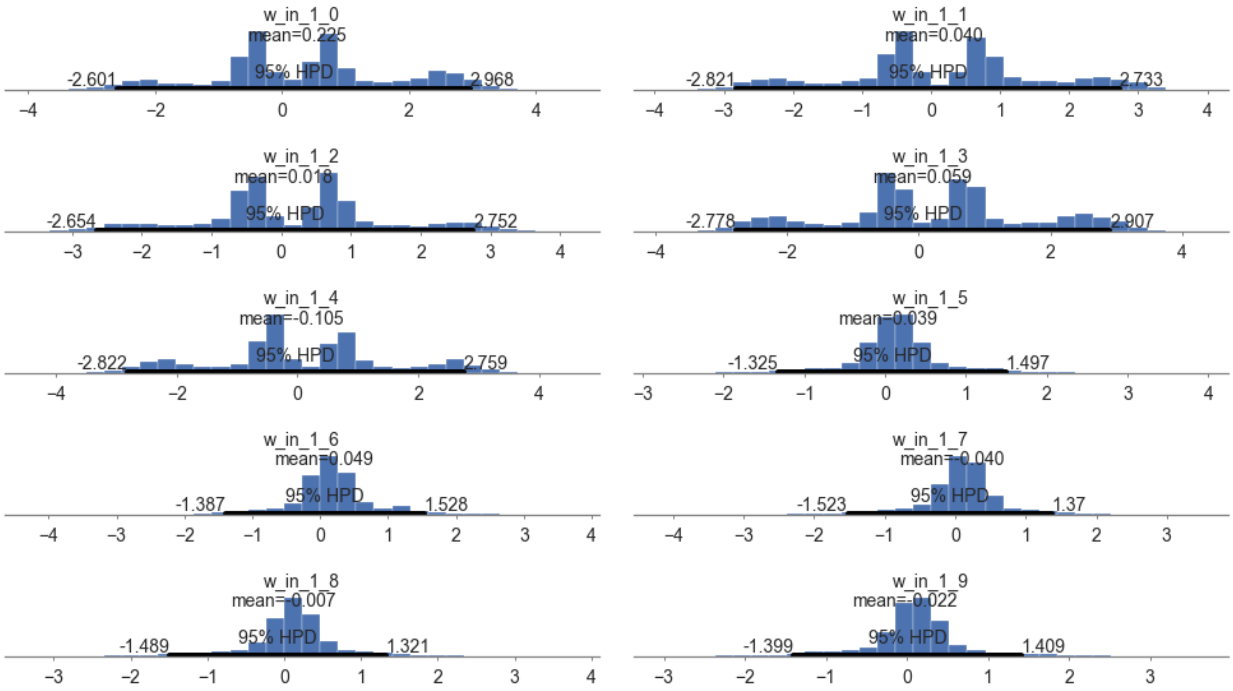
In [23]: `pm.summary(model2.trace)`

Out[23]:

	mean	sd	mc_error	hpd_2.5	hpd_97.5	n_eff
w_in_1__0_0	0.225135	1.453129	0.090766	-2.600963	2.967663	165.262816
w_in_1__0_1	0.040237	1.432005	0.092519	-2.821038	2.733033	156.605213
w_in_1__0_2	0.018382	1.318325	0.084790	-2.654008	2.752361	152.086243
w_in_1__0_3	0.059441	1.520335	0.099235	-2.778439	2.907296	171.541523
w_in_1__0_4	-0.105049	1.467413	0.106934	-2.821507	2.759036	142.862990
w_in_1__1_0	0.038815	0.617805	0.021710	-1.325060	1.497353	811.520570
w_in_1__1_1	0.048561	0.651136	0.033623	-1.387215	1.528355	315.526252
w_in_1__1_2	-0.040393	0.630075	0.029703	-1.522670	1.369573	350.442761
w_in_1__1_3	-0.006621	0.615670	0.023998	-1.488595	1.321337	588.732171
w_in_1__1_4	-0.022356	0.602055	0.023388	-1.399013	1.409409	630.152499
w_1_2__0_0	-0.030117	1.222961	0.032388	-2.222790	2.459064	1465.347411
w_1_2__0_1	-0.033481	1.257416	0.036231	-2.528749	2.320204	1122.000446
w_1_2__0_2	0.058013	1.255037	0.032539	-2.337473	2.535129	1445.368417
w_1_2__0_3	0.012622	1.216279	0.031562	-2.336343	2.350345	1548.011815
w_1_2__0_4	-0.022592	1.238055	0.036278	-2.382896	2.395463	1216.492733
w_1_2__1_0	0.004957	1.255870	0.033091	-2.390911	2.475214	1327.119255
w_1_2__1_1	0.045525	1.240332	0.026389	-2.355288	2.492054	1738.691574
w_1_2__1_2	-0.080770	1.252471	0.034783	-2.602587	2.212237	1506.411171
w_1_2__1_3	0.011486	1.249588	0.030010	-2.432115	2.427352	1752.220349
w_1_2__1_4	0.019531	1.218963	0.030665	-2.313649	2.383177	1517.420895
w_1_2__2_0	-0.022256	1.261408	0.033143	-2.522347	2.358721	1228.355488
w_1_2__2_1	-0.016605	1.260692	0.030260	-2.431030	2.443035	1762.632045
w_1_2__2_2	-0.039904	1.277497	0.036323	-2.547160	2.316859	1293.686512
w_1_2__2_3	-0.007594	1.257849	0.034130	-2.638556	2.284691	1585.661623
w_1_2__2_4	0.024879	1.207795	0.029777	-2.424846	2.292940	1751.025064
w_1_2__3_0	0.005390	1.242961	0.034378	-2.474162	2.308738	1271.123642
w_1_2__3_1	0.039325	1.312166	0.049990	-2.564783	2.642687	449.518792
w_1_2__3_2	-0.021349	1.300328	0.050331	-2.933611	2.246004	431.211084
w_1_2__3_3	0.019744	1.245684	0.027273	-2.478284	2.358122	1915.759049
w_1_2__3_4	-0.010321	1.256208	0.036211	-2.341684	2.450475	1107.995158
w_1_2__4_0	-0.058451	1.235791	0.027882	-2.581343	2.202579	1695.014178
w_1_2__4_1	0.012964	1.266865	0.035280	-2.335460	2.498519	1313.828194

```
w_1_2__4_2  -0.020461  1.286353  0.034562 -2.579003  2.365088  1301.591220
w_1_2__4_3   0.009367  1.227171  0.026680 -2.364722  2.360280  1782.650186
w_1_2__4_4  -0.022705  1.233417  0.031601 -2.367693  2.367387  1551.870684
w_2_out__0  -0.029666  2.331660  0.076394 -4.488181  4.396616   764.040782
w_2_out__1  -0.037847  2.354610  0.085388 -4.587552  4.386218   720.542202
w_2_out__2   0.127920  2.375094  0.092250 -4.319635  4.449930   523.489082
w_2_out__3   0.030048  2.295993  0.076132 -4.349295  4.310776   840.771489
w_2_out__4   0.002714  2.244351  0.078376 -4.295934  4.466476   769.253983

                Rhat
w_in_1__0_0  1.033921
w_in_1__0_1  1.008894
w_in_1__0_2  1.011680
w_in_1__0_3  1.009523
w_in_1__0_4  1.003108
w_in_1__1_0  1.001801
w_in_1__1_1  1.006621
w_in_1__1_2  1.002446
w_in_1__1_3  1.007533
w_in_1__1_4  1.003384
w_1_2__0_0   0.999999
w_1_2__0_1   1.003739
w_1_2__0_2   1.000439
w_1_2__0_3   1.000222
w_1_2__0_4   1.001501
w_1_2__1_0   1.002022
w_1_2__1_1   1.000285
w_1_2__1_2   1.001734
w_1_2__1_3   1.003849
w_1_2__1_4   1.000226
w_1_2__2_0   1.000351
w_1_2__2_1   1.000269
w_1_2__2_2   1.005241
w_1_2__2_3   1.000065
w_1_2__2_4   1.001687
w_1_2__3_0   1.002067
w_1_2__3_1   1.003777
w_1_2__3_2   1.002436
w_1_2__3_3   1.000700
w_1_2__3_4   1.004730
w_1_2__4_0   0.999790
w_1_2__4_1   0.999927
w_1_2__4_2   1.001516
w_1_2__4_3   1.002149
w_1_2__4_4   1.000180
w_2_out__0   1.010481
w_2_out__1   1.001810
w_2_out__2   1.005582
w_2_out__3   1.002162
w_2_out__4   1.001621
```

```
In [24]: pm.plot_posterior(model2.trace, varnames=["w_in_1"],
                          figsize = [14, 8]);
```




```
In [25]: y_pred2 = model2.predict(X_test)

100%|| 2000/2000 [00:00<00:00, 2569.74it/s]

In [26]: y_pred2

Out[26]: array([False,  True,  True, False, False,  True,  True,  True,  True,
                False, False,  True,  True, False,  True,  True,  True, False,
                False,  True,  True,  True,  True,  True, False,  True, False,
                False, False, False, False,  True, False,  True,  True, False,
                 True,  True,  True, False, False, False,  True,  True, False,
                False, False,  True, False, False, False,  True,  True, False,
                 True,  True,  True, False,  True, False,  True, False,  True,
                False, False,  True,  True, False, False,  True, False, False,
                False,  True,  True,  True,  True,  True,  True, False, False,
                False, False, False,  True,  True,  True, False, False, False,
                 True,  True,  True,  True, False,  True, False,  True,  True,
                False,  True,  True,  True,  True, False, False, False,  True,
                False, False, False, False,  True, False, False,  True, False,
                 True,  True, False, False,  True,  True,  True, False,  True,
                 True,  True, False,  True, False,  True,  True, False, False,
                 True,  True,  True,  True, False,  True,  True, False, False,
                 True, False,  True, False,  True,  True, False,  True, False,
                 True,  True,  True, False, False, False, False,  True, False,
                 True, False,  True, False, False, False, False,  True,  True,
                False,  True, False,  True, False,  True, False,  True,  True,
                 True,  True, False,  True,  True,  True, False, False,  True,
                False,  True,  True, False,  True, False,  True, False, False,
                 True, False, False,  True,  True, False,  True,  True,  True,
                False,  True,  True,  True,  True,  True, False,  True, False,
                 True,  True, False,  True, False, False,  True,  True, False,
                False,  True,  True,  True, False, False,  True,  True,  True,
                False,  True, False, False,  True, False, False,  True, False,
                False,  True,  True, False, False,  True,  True, False,  True,
                 True, False, False, False,  True, False, False, False, False,
                False, False, False,  True, False,  True, False, False, False,
```

```
                True, False,  True, False, False,  True, False, False,  True,
               False,  True,  True,  True, False,  True,  True,  True,  True,
                True, False, False, False,  True, False,  True, False, False,
               False, False, False], dtype=bool)
```

```
In [27]: model2.score(X_test, y_test)
```

```
100%|| 2000/2000 [00:00<00:00, 2645.86it/s]
```

```
Out[27]: 0.96333333333333337
```

```
In [28]: model2.save('pickle_jar/mlpc2')
         model2_new = MLPClassifier()
         model2_new.load('pickle_jar/mlpc2')
         model2_new.score(X_test, y_test)
```

```
100%|| 2000/2000 [00:00<00:00, 2550.55it/s]
```

```
Out[28]: 0.95999999999999996
```

7.6 Classification

7.6.1 Gaussian Naive Bayes

Let's set some setting for this Jupyter Notebook.

```
In [2]: %matplotlib inline
        from warnings import filterwarnings
        filterwarnings("ignore")
        import os
        os.environ['MKL_THREADING_LAYER'] = 'GNU'
        os.environ['THEANO_FLAGS'] = 'device=cpu'

        import numpy as np
        import pandas as pd
        import pymc3 as pm
        import seaborn as sns
        import matplotlib.pyplot as plt
        np.random.seed(12345)
        rc = {'xtick.labelsize': 20, 'ytick.labelsize': 20, 'axes.labelsize': 20, 'font.size': 20,
              'legend.fontsize': 12.0, 'axes.titlesize': 10, "figure.figsize": [12, 6]}
        sns.set(rc = rc)
        from IPython.core.interactiveshell import InteractiveShell
        InteractiveShell.ast_node_interactivity = "all"
```

Now, let's import the `GaussianNB` (GaussianNaiveBayes) model from the `pymc-learn` package.

```
In [3]: import pmlearn
        from pmlearn.naive_bayes import GaussianNB
        print('Running on pymc-learn v{}'.format(pmlearn.__version__))
```

```
Running on pymc-learn v0.0.1.rc0
```

Step 1: Prepare the data

Use the popular iris dataset.

```
In [4]: # Load the data and split in train and test set
        from sklearn.datasets import load_iris
        X = load_iris().data
```

```
        y = load_iris().target
        X.shape
```

```
Out[4]: (150, 4)
```

```
In [5]: from sklearn.model_selection import train_test_split
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

Step 2: Instantiate a model

```
In [6]: model = GaussianNB()
```

Step 3: Perform Inference

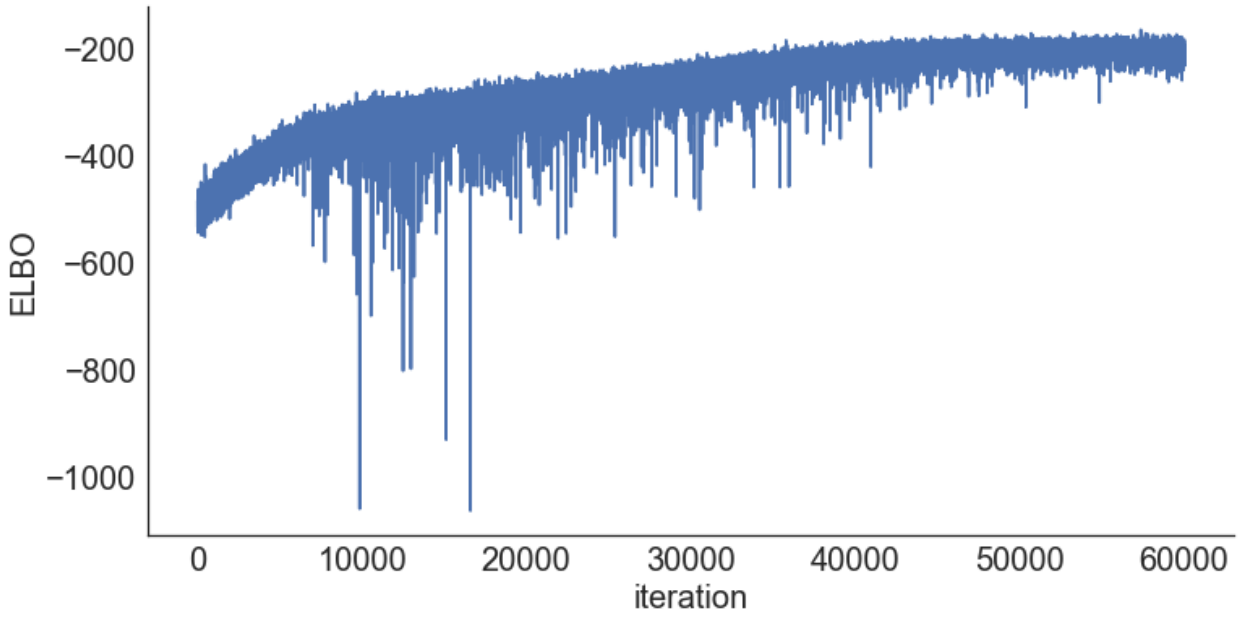
```
In [7]: model.fit(X_train, y_train, minibatch_size=20, inference_args={'n': 60000})
```

```
Average Loss = 201.71: 100%|| 60000/60000 [00:47<00:00, 1272.93it/s]
Finished [100%]: Average Loss = 201.84
```

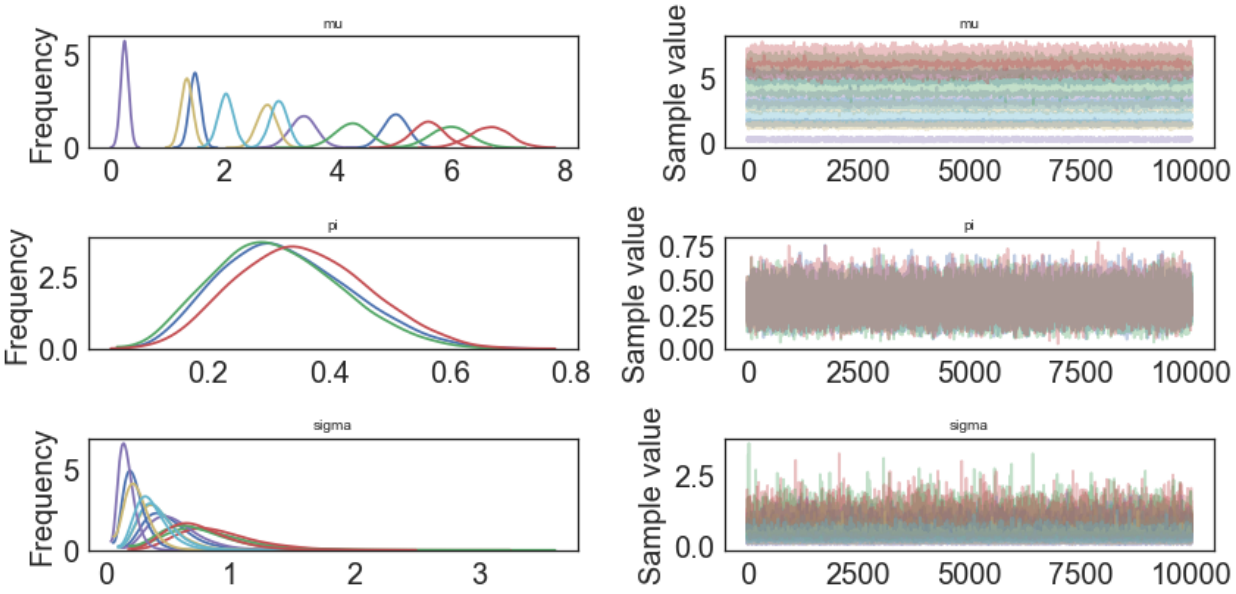
```
Out[7]: GaussianNB()
```

Step 4: Diagnose convergence

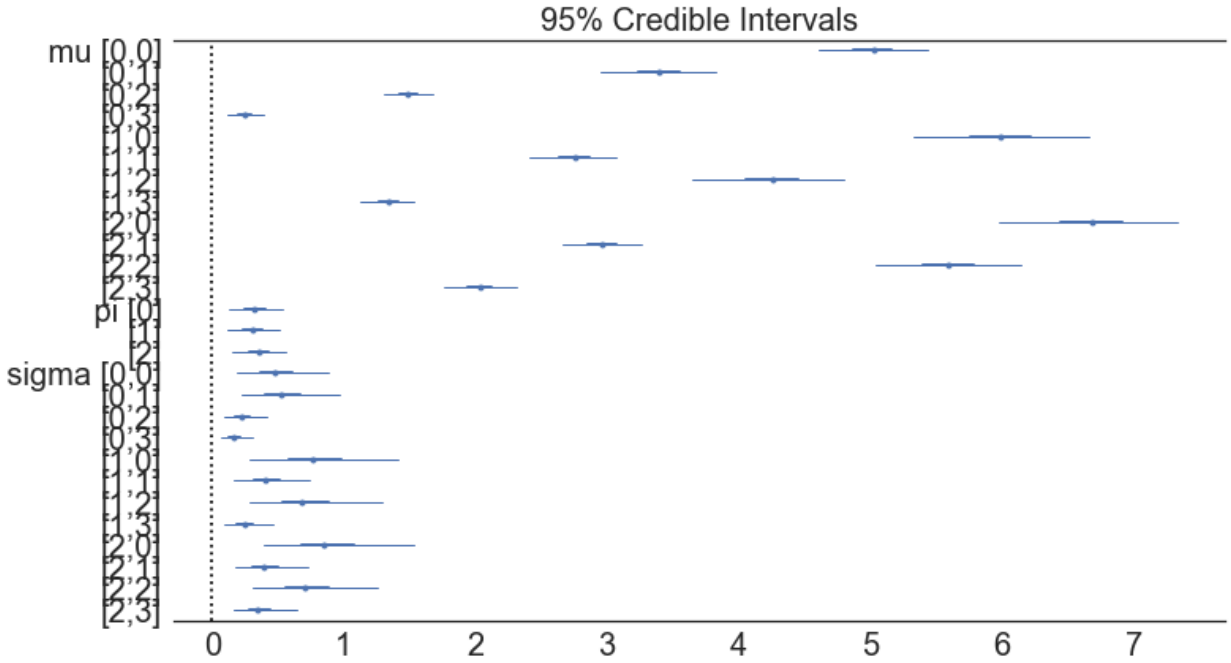
```
In [8]: model.plot_elbo()
```




```
In [9]: pm.traceplot(model.trace);
```




```
In [10]: pm.forestplot(model.trace);
```



Step 5: Critize the model

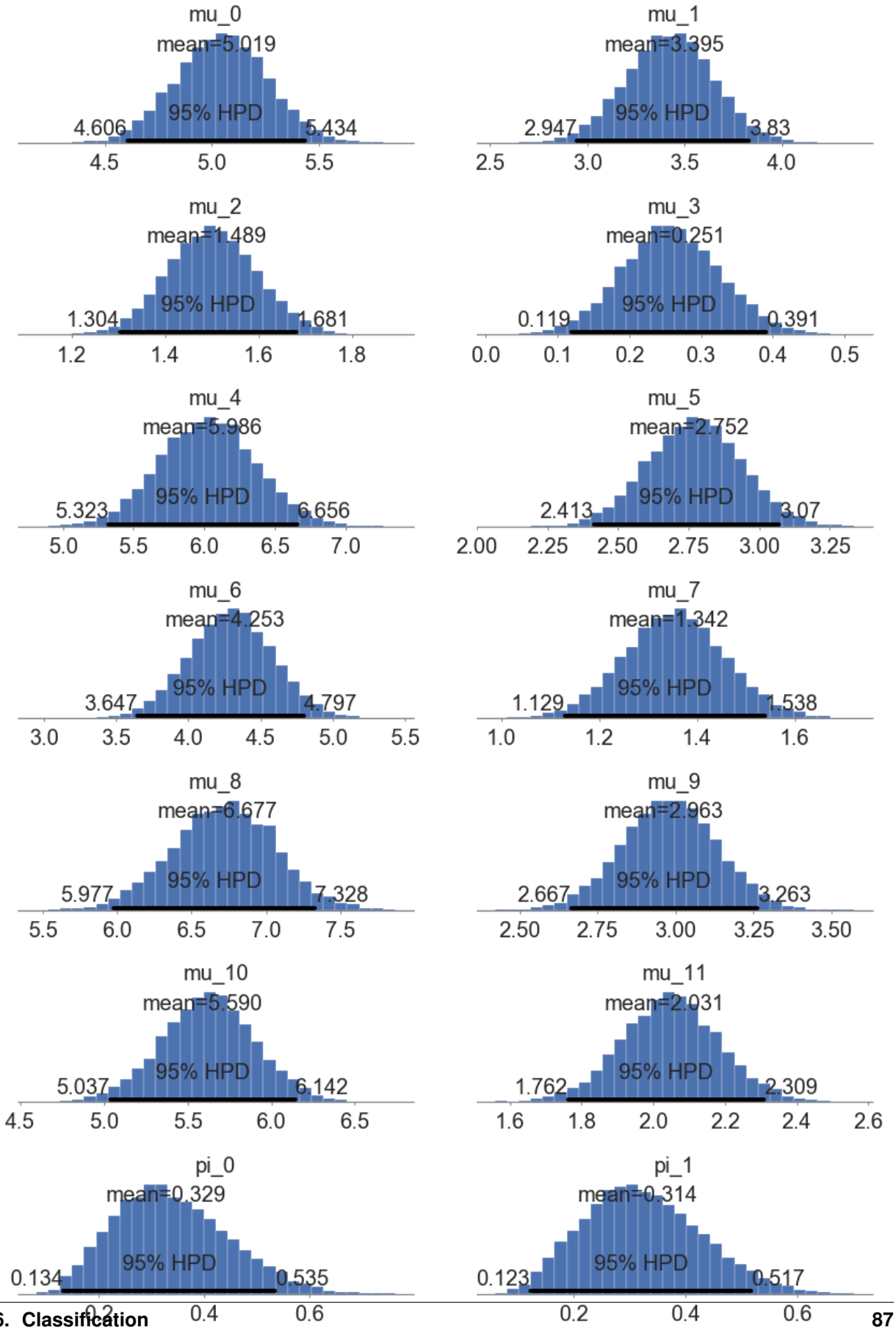
```
In [11]: pm.summary(model.trace)
```

Out[11]:

	mean	sd	mc_error	hpd_2.5	hpd_97.5
mu__0_0	5.019129	0.212662	0.002406	4.605529	5.433732
mu__0_1	3.394539	0.225085	0.002227	2.947272	3.830176
mu__0_2	1.488766	0.096130	0.001063	1.304366	1.681172
mu__0_3	0.250523	0.069351	0.000623	0.119370	0.391107
mu__1_0	5.986291	0.340178	0.003272	5.322534	6.656125
mu__1_1	2.751523	0.166468	0.001648	2.413300	3.069630

```
mu__1_2     4.253440  0.292789  0.003280  3.647427  4.796967
mu__1_3     1.342020  0.104551  0.000995  1.128888  1.538075
mu__2_0     6.677160  0.343993  0.003418  5.976978  7.327982
mu__2_1     2.962960  0.153138  0.001455  2.666622  3.262982
mu__2_2     5.589977  0.280147  0.002411  5.036929  6.141556
mu__2_3     2.031164  0.137477  0.001347  1.761671  2.309202
pi__0       0.328850  0.106844  0.001078  0.134363  0.535277
pi__1       0.313944  0.104287  0.001116  0.122916  0.517405
pi__2       0.357206  0.107097  0.000971  0.155631  0.564292
sigma__0_0  0.503346  0.190957  0.001960  0.186719  0.881564
sigma__0_1  0.558681  0.207242  0.001941  0.225201  0.965511
sigma__0_2  0.236719  0.089451  0.000974  0.095018  0.419134
sigma__0_3  0.174252  0.065734  0.000752  0.068633  0.306643
sigma__1_0  0.815532  0.317363  0.003118  0.289326  1.414879
sigma__1_1  0.428283  0.157381  0.001597  0.168046  0.743495
sigma__1_2  0.734456  0.278321  0.002860  0.284165  1.292951
sigma__1_3  0.261675  0.104147  0.000933  0.094439  0.464214
sigma__2_0  0.905180  0.316311  0.003079  0.389695  1.532829
sigma__2_1  0.419737  0.150378  0.001294  0.180018  0.725100
sigma__2_2  0.748047  0.259204  0.002559  0.313343  1.256877
sigma__2_3  0.371764  0.129162  0.001235  0.164041  0.641036
```

In [12]:
```
pm.plot_posterior(model.trace);
```


mu_0
mean=5.019
95% HPD
4.606
5.434
4.5
5.0
5.5
mu_1
mean=3.395
95% HPD
2.947
3.83
2.5
3.0
3.5
4.0
mu_2
mean=1.489
95% HPD
1.304
1.681
1.2
1.4
1.6
1.8
mu_3
mean=0.251
95% HPD
0.119
0.391
0.0
0.1
0.2
0.3
0.4
0.5
mu_4
mean=5.986
95% HPD
5.323
6.656
5.0
5.5
6.0
6.5
7.0
mu_5
mean=2.752
95% HPD
2.413
3.07
2.00
2.25
2.50
2.75
3.00
3.25
mu_6
mean=4.253
95% HPD
3.647
4.797
3.0
3.5
4.0
4.5
5.0
5.5
mu_7
mean=1.342
95% HPD
1.129
1.538
1.0
1.2
1.4
1.6
mu_8
mean=6.677
95% HPD
5.977
7.328
5.5
6.0
6.5
7.0
7.5
mu_9
mean=2.963
95% HPD
2.667
3.263
2.50
2.75
3.00
3.25
3.50
mu_10
mean=5.590
95% HPD
5.037
6.142
4.5
5.0
5.5
6.0
6.5
mu_11
mean=2.031
95% HPD
1.762
2.309
1.6
1.8
2.0
2.2
2.4
2.6
pi_0
mean=0.329
95% HPD
0.134
0.535
0.2
0.4
0.6
pi_1
mean=0.314
95% HPD
0.123
0.517
0.2
0.4
0.6
pi_2
mean=0.357
sigma_0
mean=0.503

Step 6: Use the model for prediction

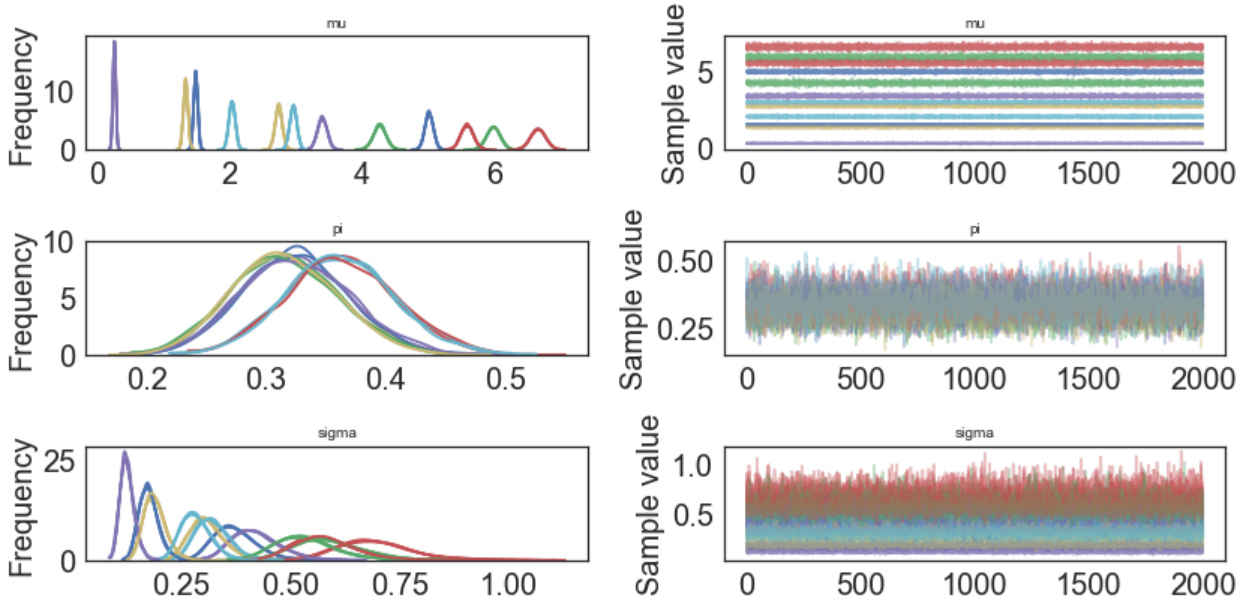
```
In [13]: y_probs = model.predict_proba(X_test)

In [14]: y_predicted = model.predict(X_test)

In [15]: model.score(X_test, y_test)

Out[15]: 0.9555555555555556

In [16]: model.save('pickle_jar/gaussian_nb')
```

Use already trained model for prediction

```
In [17]: model_new = GaussianNB()

In [18]: model_new.load('pickle_jar/gaussian_nb')

In [19]: model_new.score(X_test, y_test)

Out[19]: 0.9555555555555556
```

MCMC

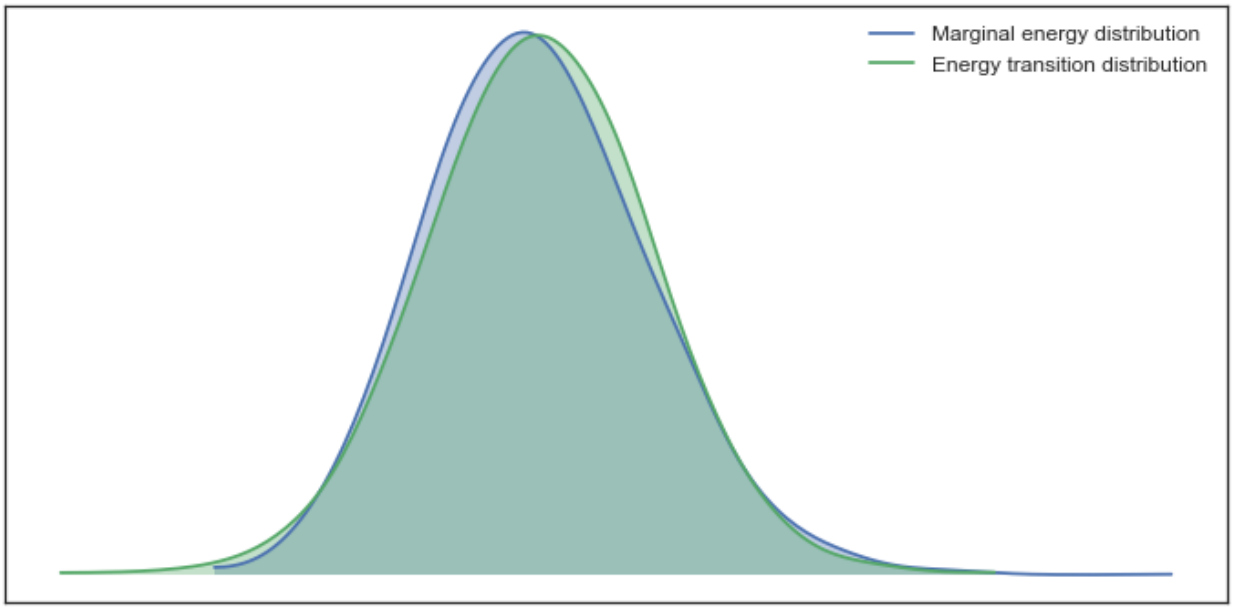
```
In [20]: model2 = GaussianNB()
         model2.fit(X_train, y_train, inference_type='nuts')

Multiprocess sampling (4 chains in 4 jobs)
NUTS: [sigma_log__, mu, pi_stickbreaking__]
100%|| 2500/2500 [00:08<00:00, 301.33it/s]

Out[20]: GaussianNB()

In [21]: pm.traceplot(model2.trace);
```




```
In [22]: pm.gelman_rubin(model2.trace)

Out[22]: {'mu': array([[ 0.99985081,  0.9999332 ,  0.99978104,  0.99996147],
                       [ 0.99982672,  0.99994778,  0.99991087,  0.99986595],
                       [ 0.99983744,  0.99977463,  0.99991918,  0.99980338]]),
```

```
         'pi': array([ 0.99985966,  0.99988375,  1.00008299]),
         'sigma': array([[ 0.99987713,  0.99998867,  1.00005796,  0.99990652],
                [ 0.99976896,  0.99990503,  0.99990054,  0.9999672 ],
                [ 0.9998285 ,  0.99984983,  0.99997151,  0.99980225]])}
```

In [23]:
```
pm.energyplot(model2.trace);
```



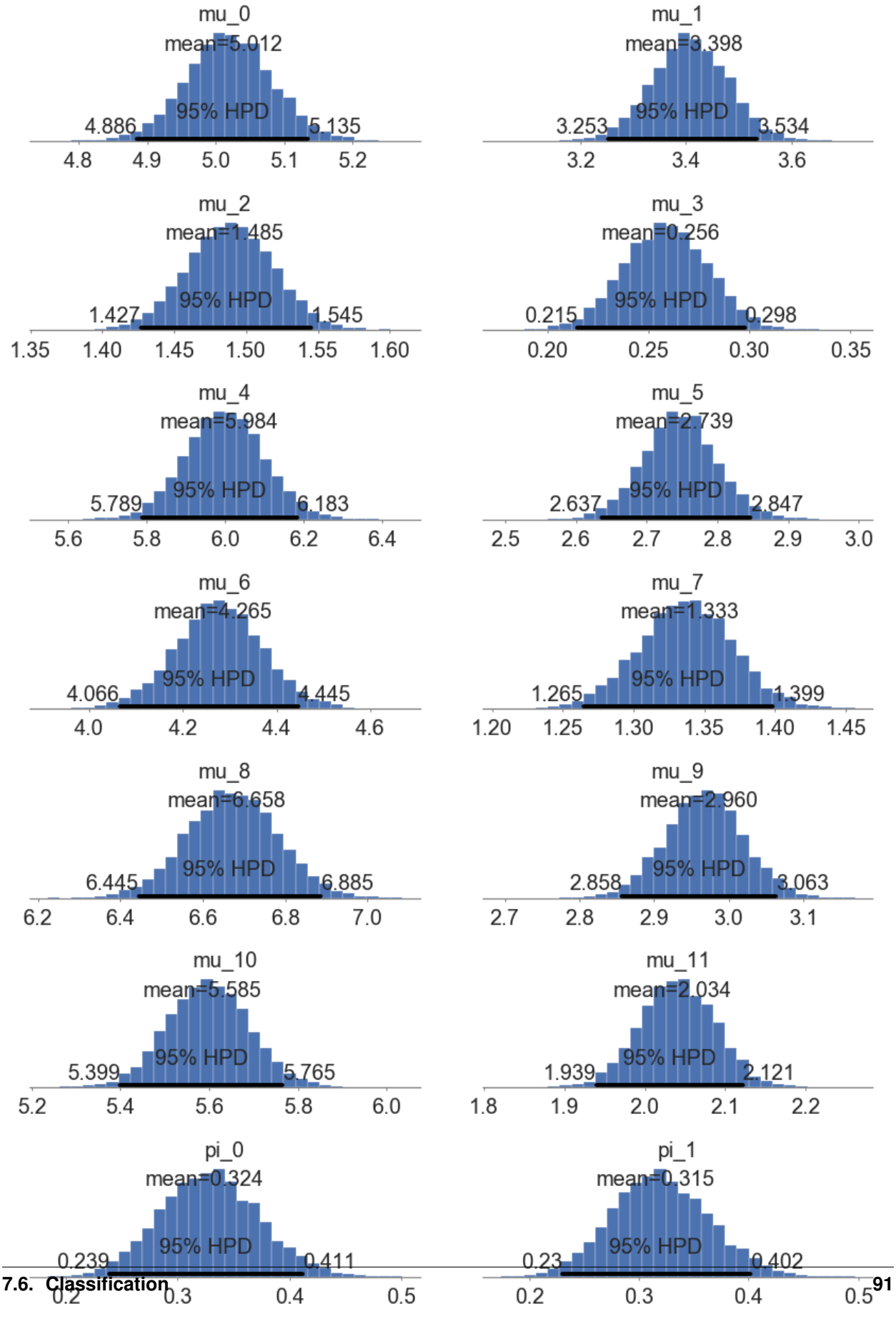
In [24]:
```
pm.summary(model2.trace)
```

Out[24]:

	mean	sd	mc_error	hpd_2.5	hpd_97.5	n_eff
mu__0_0	5.012208	0.063459	0.000506	4.885851	5.135442	14783.745310
mu__0_1	3.397522	0.072278	0.000569	3.252539	3.533969	14301.507246
mu__0_2	1.485131	0.030497	0.000254	1.426674	1.544899	16911.345807
mu__0_3	0.255895	0.021337	0.000169	0.214929	0.297620	13970.212384
mu__1_0	5.984250	0.101500	0.000834	5.788512	6.182700	13085.818483
mu__1_1	2.738908	0.053064	0.000481	2.636860	2.846584	14679.479750
mu__1_2	4.264700	0.095694	0.000827	4.066109	4.445209	16974.414575
mu__1_3	1.332982	0.033943	0.000311	1.264938	1.398529	12851.649197
mu__2_0	6.657839	0.111913	0.000745	6.444689	6.885482	16906.114757
mu__2_1	2.960206	0.052699	0.000399	2.857604	3.063223	15651.929453
mu__2_2	5.585155	0.093777	0.000720	5.398764	5.765116	16570.157691
mu__2_3	2.034040	0.047202	0.000361	1.939357	2.121193	14943.741321
pi__0	0.323797	0.044186	0.000370	0.239178	0.411420	18313.620592
pi__1	0.314711	0.044053	0.000342	0.230061	0.401513	17898.713381
pi__2	0.361493	0.045533	0.000373	0.276315	0.455464	18869.885537
sigma__0_0	0.371764	0.048002	0.000382	0.281731	0.463962	14608.499926
sigma__0_1	0.420122	0.054393	0.000512	0.322898	0.529358	13256.255999
sigma__0_2	0.175457	0.022088	0.000208	0.133977	0.218597	11831.316071
sigma__0_3	0.125911	0.016275	0.000139	0.096134	0.158375	15934.091949
sigma__1_0	0.586714	0.077468	0.000673	0.449892	0.739071	11844.705535
sigma__1_1	0.308644	0.039708	0.000326	0.238049	0.389450	15426.313774
sigma__1_2	0.533438	0.069761	0.000576	0.412220	0.680833	16527.525060
sigma__1_3	0.189063	0.024813	0.000211	0.144906	0.240522	14229.895510
sigma__2_0	0.691582	0.082996	0.000631	0.537945	0.859956	15429.823316
sigma__2_1	0.320985	0.038669	0.000337	0.249556	0.396254	12245.493763
sigma__2_2	0.576983	0.069838	0.000523	0.449596	0.716449	13538.105853
sigma__2_3	0.282929	0.034015	0.000285	0.223283	0.351827	15569.676214

```
                 Rhat
mu__0_0      0.999851
mu__0_1      0.999933
mu__0_2      0.999781
mu__0_3      0.999961
mu__1_0      0.999827
mu__1_1      0.999948
mu__1_2      0.999911
mu__1_3      0.999866
mu__2_0      0.999837
mu__2_1      0.999775
mu__2_2      0.999919
mu__2_3      0.999803
pi__0        0.999860
pi__1        0.999884
pi__2        1.000083
sigma__0_0   0.999877
sigma__0_1   0.999989
sigma__0_2   1.000058
sigma__0_3   0.999907
sigma__1_0   0.999769
sigma__1_1   0.999905
sigma__1_2   0.999901
sigma__1_3   0.999967
sigma__2_0   0.999828
sigma__2_1   0.999850
sigma__2_2   0.999972
sigma__2_3   0.999802
```

In [25]:
```
pm.plot_posterior(model2.trace);
```


mu_0
mean=5.012
95% HPD
4.886
5.135
4.8
4.9
5.0
5.1
5.2
mu_1
mean=3.398
95% HPD
3.253
3.534
3.2
3.4
3.6
mu_2
mean=1.485
95% HPD
1.427
1.545
1.35
1.40
1.45
1.50
1.55
1.60
mu_3
mean=0.256
95% HPD
0.215
0.298
0.20
0.25
0.30
0.35
mu_4
mean=5.984
95% HPD
5.789
6.183
5.6
5.8
6.0
6.2
6.4
mu_5
mean=2.739
95% HPD
2.637
2.847
2.5
2.6
2.7
2.8
2.9
3.0
mu_6
mean=4.265
95% HPD
4.066
4.445
4.0
4.2
4.4
4.6
mu_7
mean=1.333
95% HPD
1.265
1.399
1.20
1.25
1.30
1.35
1.40
1.45
mu_8
mean=6.658
95% HPD
6.445
6.885
6.2
6.4
6.6
6.8
7.0
mu_9
mean=2.960
95% HPD
2.858
3.063
2.7
2.8
2.9
3.0
3.1
mu_10
mean=5.585
95% HPD
5.399
5.765
5.2
5.4
5.6
5.8
6.0
mu_11
mean=2.034
95% HPD
1.939
2.121
1.8
1.9
2.0
2.1
2.2
pi_0
mean=0.324
95% HPD
0.239
0.411
0.2
0.3
0.4
0.5
pi_1
mean=0.315
95% HPD
0.23
0.402
0.2
0.3
0.4
0.5
pi_2
sigma_0

```
In [26]: y_predict2 = model2.predict(X_test)

In [27]: model2.score(X_test, y_test)

Out[27]: 0.9555555555555556

In [28]: model2.save('pickle_jar/gaussian_nb2')
         model2_new = GaussianNB()
         model2_new.load('pickle_jar/gaussian_nb2')
         model2_new.score(X_test, y_test)

Out[28]: 0.9555555555555556
```

7.7 Mixture Models

7.8 Bayesian Neural Networks

API Reference

`pymc-learn` leverages and extends the Base template provided by the PyMC3 Models project: https://github.com/parsing-science/pymc3_models.

- *API*

7.9 API

7.9.1 pmlearn

pmlearn package

Subpackages

pmlearn.gaussian_process package

Subpackages

pmlearn.gaussian_process.tests package

Submodules

pmlearn.gaussian_process.tests.test_gpr module

Testing for Gaussian process regression

class pmlearn.gaussian_process.tests.test_gpr.**TestGaussianProcessRegressor**

Bases: object

setup_method()

teardown_method()

Tear down

class pmlearn.gaussian_process.tests.test_gpr.**TestGaussianProcessRegressorFit**
Bases: *pmlearn.gaussian_process.tests.test_gpr.TestGaussianProcessRegressor*

test_advi_fit_returns_correct_model()

class pmlearn.gaussian_process.tests.test_gpr.**TestGaussianProcessRegressorPredict**
Bases: *pmlearn.gaussian_process.tests.test_gpr.TestGaussianProcessRegressor*

test_predict_raises_error_if_not_fit()

test_predict_returns_mean_predictions_and_std()

test_predict_returns_predictions()

class pmlearn.gaussian_process.tests.test_gpr.**TestGaussianProcessRegressorSaveAndLoad**
Bases: *pmlearn.gaussian_process.tests.test_gpr.TestGaussianProcessRegressor*

test_save_and_load_work_correctly()

class pmlearn.gaussian_process.tests.test_gpr.**TestGaussianProcessRegressorScore**
Bases: *pmlearn.gaussian_process.tests.test_gpr.TestGaussianProcessRegressor*

test_score_matches_sklearn_performance()

class pmlearn.gaussian_process.tests.test_gpr.**TestSparseGaussianProcessRegressor**
Bases: object

setup_method()

teardown_method()
Tear down

class pmlearn.gaussian_process.tests.test_gpr.**TestSparseGaussianProcessRegressorFit**
Bases: *pmlearn.gaussian_process.tests.test_gpr.TestSparseGaussianProcessRegressor*

test_advi_fit_returns_correct_model()

class pmlearn.gaussian_process.tests.test_gpr.**TestSparseGaussianProcessRegressorPredict**
Bases: *pmlearn.gaussian_process.tests.test_gpr.TestSparseGaussianProcessRegressor*

test_predict_raises_error_if_not_fit()

test_predict_returns_mean_predictions_and_std()

test_predict_returns_predictions()

class pmlearn.gaussian_process.tests.test_gpr.**TestSparseGaussianProcessRegressorSaveAndLoad**
Bases: *pmlearn.gaussian_process.tests.test_gpr.TestSparseGaussianProcessRegressor*

test_save_and_load_work_correctly()

class pmlearn.gaussian_process.tests.test_gpr.**TestSparseGaussianProcessRegressorScore**
Bases: *pmlearn.gaussian_process.tests.test_gpr.TestSparseGaussianProcessRegressor*

test_score_matches_sklearn_performance()

class pmlearn.gaussian_process.tests.test_gpr.**TestStudentsTProcessRegressor**
Bases: object

setup_method()

tearDown()

class pmlearn.gaussian_process.tests.test_gpr.**TestStudentsTProcessRegressorFit**
Bases: *pmlearn.gaussian_process.tests.test_gpr.TestStudentsTProcessRegressor*

test_advi_fit_returns_correct_model()

class pmlearn.gaussian_process.tests.test_gpr.**TestStudentsTProcessRegressorPredict**
Bases: *pmlearn.gaussian_process.tests.test_gpr.TestStudentsTProcessRegressor*

test_predict_raises_error_if_not_fit()

test_predict_returns_mean_predictions_and_std()

test_predict_returns_predictions()

class pmlearn.gaussian_process.tests.test_gpr.**TestStudentsTProcessRegressorSaveAndLoad**
Bases: *pmlearn.gaussian_process.tests.test_gpr.TestStudentsTProcessRegressor*

test_save_and_load_work_correctly()

class pmlearn.gaussian_process.tests.test_gpr.**TestStudentsTProcessRegressorScore**
Bases: *pmlearn.gaussian_process.tests.test_gpr.TestStudentsTProcessRegressor*

test_score_matches_sklearn_performance()

Module contents

Submodules

pmlearn.gaussian_process.gpc module

pmlearn.gaussian_process.gpr module

Gaussian process regression.

class pmlearn.gaussian_process.gpr.**GaussianProcessRegressor**(*prior_mean=None*, *kernel=None*)
Bases: *pmlearn.base.BayesianModel*, *pmlearn.gaussian_process.gpr.GaussianProcessRegressorMixin*

Gaussian Process Regression built using PyMC3.

Fit a Gaussian process model and estimate model parameters using MCMC algorithms or Variational Inference algorithms

Parameters

- **prior_mean** (*mean object*) – The mean specifying the mean function of the GP. If None is passed, the mean "pm.gp.mean.Zero()" is used as default.
- **kernel** (*covariance function (kernel)*) – The function specifying the covariance of the GP. If None is passed, the kernel "RBF()" is used as default.

Examples

```
>>> from sklearn.datasets import make_friedman2
>>> from pmlearn.gaussian_process import GaussianProcessRegressor
>>> from pmlearn.gaussian_process.kernels import DotProduct, WhiteKernel
>>> X, y = make_friedman2(n_samples=500, noise=0, random_state=0)
>>> kernel = DotProduct() + WhiteKernel()
>>> gpr = GaussianProcessRegressor(kernel=kernel).fit(X, y)
>>> gpr.score(X, y)
0.3680...
>>> gpr.predict(X[:2,:], return_std=True)
(array([653.0..., 592.1...]), array([316.6..., 316.6...]))
```

Rasmussen and Williams (2006). Gaussian Processes for Machine Learning.

create_model()

Creates and returns the PyMC3 model.

Note: The size of the shared variables must match the size of the training data. Otherwise, setting the shared variables later will raise an error. See http://docs.pymc.io/advanced_theano.html

Returns **model**

Return type the PyMC3 model.

load(*file_prefix*)

Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns **custom_params**

Return type Dictionary of custom parameters

save(*file_prefix*)

Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

class pmlearn.gaussian_process.gpr.**GaussianProcessRegressorMixin**

Bases: *pmlearn.base.BayesianRegressorMixin*

Mixin class for Gaussian Process Regression

predict (*X*, *return_std=False*)
Perform Prediction

Predicts values of new data with a trained Gaussian Process Regression model

Parameters

- **X** (*numpy array, shape [n_samples, n_features]*) –
- **return_std** (*Boolean*) – Whether to return standard deviations with mean values. Defaults to False.

class pmlearn.gaussian_process.gpr.**SparseGaussianProcessRegressor**(*prior_mean=None*, *kernel=None*)

Bases: *pmlearn.base.BayesianModel*, *pmlearn.gaussian_process.gpr.GaussianProcessRegressorMixin*

Sparse Gaussian Process Regression built using PyMC3.

Fit a Sparse Gaussian process model and estimate model parameters using MCMC algorithms or Variational Inference algorithms

Parameters **prior_mean** (*mean object*) – The mean specifying the mean function of the GP. If None is passed, the mean "pm.gp.mean.Zero()" is used as default.

Examples

```
>>> from sklearn.datasets import make_friedman2
>>> from pmlearn.gaussian_process import SparseGaussianProcessRegressor
>>> from sklearn.gaussian_process.kernels import DotProduct, WhiteKernel
>>> X, y = make_friedman2(n_samples=500, noise=0, random_state=0)
>>> kernel = DotProduct() + WhiteKernel()
>>> sgpr = SparseGaussianProcessRegressor(kernel=kernel).fit(X, y)
>>> sgpr.score(X, y)
0.3680...
>>> sgpr.predict(X[:2,:], return_std=True)
(array([653.0..., 592.1...]), array([316.6..., 316.6...]))
```

Rasmussen and Williams (2006). Gaussian Processes for Machine Learning.

create_model()
Creates and returns the PyMC3 model.

Note: The size of the shared variables must match the size of the training data. Otherwise, setting the shared variables later will raise an error. See http://docs.pymc.io/advanced_theano.html

Returns **model**

Return type the PyMC3 model

load (*file_prefix*)
Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"

- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns **custom_params**

Return type Dictionary of custom parameters

save(*file_prefix*)
Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

class pmlearn.gaussian_process.gpr.**StudentsTProcessRegressor**(*prior_mean=None*, *kernel=None*)

Bases: *pmlearn.base.BayesianModel*, *pmlearn.gaussian_process.gpr.GaussianProcessRegressorMixin*

StudentsT Process Regression built using PyMC3.

Fit a StudentsT process model and estimate model parameters using MCMC algorithms or Variational Inference algorithms

Parameters **prior_mean** (*mean object*) – The mean specifying the mean function of the StudentsT process. If None is passed, the mean "pm.gp.mean.Zero()" is used as default.

Examples

```
>>> from sklearn.datasets import make_friedman2
>>> from pmlearn.gaussian_process import StudentsTProcessRegressor
>>> from sklearn.gaussian_process.kernels import DotProduct, WhiteKernel
>>> X, y = make_friedman2(n_samples=500, noise=0, random_state=0)
>>> kernel = DotProduct() + WhiteKernel()
>>> spr = StudentsTProcessRegressor(kernel=kernel).fit(X, y)
>>> spr.score(X, y)
0.3680...
>>> spr.predict(X[:2,:], return_std=True)
(array([653.0..., 592.1...]), array([316.6..., 316.6...]))
```

Rasmussen and Williams (2006). Gaussian Processes for Machine Learning.

create_model()
Creates and returns the PyMC3 model.

Note: The size of the shared variables must match the size of the training data. Otherwise, setting the shared variables later will raise an error. See http://docs.pymc.io/advanced_theano.html

Returns **model**

Return type the PyMC3 model

load(*file_prefix*)
Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns custom_params

Return type Dictionary of custom parameters

save(*file_prefix*)
Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

pmlearn.gaussian_process.kernels module

Kernels for Gaussian process regression and classification.

class pmlearn.gaussian_process.kernels.**DotProduct**(*input_dim*, *ls=None*, *ls_inv=None*, *active_dims=None*)

Bases: pymc3.gp.cov.Exponential

Dot-Product kernel from pymc3.gp.cov.Exponential

The DotProduct kernel is non-stationary and can be obtained from linear regression by putting N(0, 1) priors on the coefficients of x_d (d = 1, . . . , D) and a prior of N(0, sigma_0^2) on the bias. The DotProduct kernel is invariant to a rotation of the coordinates about the origin, but not translations. It is parameterized by a parameter sigma_0^2. For sigma_0^2 =0, the kernel is called the homogeneous linear kernel, otherwise it is inhomogeneous.

The kernel is given by k(x_i, x_j) = sigma_0 ^ 2 + x_i cdot x_j

The DotProduct kernel is commonly combined with exponentiation.

class pmlearn.gaussian_process.kernels.**RBF**(*input_dim*, *ls=None*, *ls_inv=None*, *active_dims=None*)

Bases: pymc3.gp.cov.ExpQuad

Radial-basis function kernel from pymc3.gp.cov.ExpQuad

The RBF kernel is a stationary kernel. It is also known as the "squared exponential" kernel. It is parameterized by a length-scale parameter length_scale>0, which can either be a scalar (isotropic variant of the kernel) or a vector with the same number of dimensions as the inputs X (anisotropic variant of the kernel).

The kernel is given by:

k(x_i, x_j) = exp(-1 / 2 d(x_i / length_scale, x_j / length_scale)^2)

This kernel is infinitely differentiable, which implies that GPs with this kernel as covariance function have mean square derivatives of all orders, and are thus very smooth.

class pmlearn.gaussian_process.kernels.**WhiteKernel**(*noise_level=1.0*)

Bases: pymc3.gp.cov.WhiteNoise

White kernel from ‘‘pymc3.gp.cov.WhiteNoise..

The main use-case of this kernel is as part of a sum-kernel where it explains the noise-component of the signal. Tuning its parameter corresponds to estimating the noise-level.

k(x_1, x_2) = noise_level if x_1 == x_2 else 0

Module contents

The *pmlearn.gaussian_process* module implements Gaussian Process based regression and classification.

class pmlearn.gaussian_process.**GaussianProcessRegressor**(*prior_mean=None*, *kernel=None*)

Bases: *pmlearn.base.BayesianModel*, *pmlearn.gaussian_process.gpr.GaussianProcessRegressorMixin*

Gaussian Process Regression built using PyMC3.

Fit a Gaussian process model and estimate model parameters using MCMC algorithms or Variational Inference algorithms

Parameters

- **prior_mean** (*mean object*) – The mean specifying the mean function of the GP. If None is passed, the mean “pm.gp.mean.Zero()” is used as default.
- **kernel** (*covariance function (kernel)*) – The function specifying the covariance of the GP. If None is passed, the kernel “RBF()” is used as default.

Examples

```
>>> from sklearn.datasets import make_friedman2
>>> from pmlearn.gaussian_process import GaussianProcessRegressor
>>> from pmlearn.gaussian_process.kernels import DotProduct, WhiteKernel
>>> X, y = make_friedman2(n_samples=500, noise=0, random_state=0)
>>> kernel = DotProduct() + WhiteKernel()
>>> gpr = GaussianProcessRegressor(kernel=kernel).fit(X, y)
>>> gpr.score(X, y)
0.3680...
>>> gpr.predict(X[:2,:], return_std=True)
(array([653.0..., 592.1...]), array([316.6..., 316.6...]))
```

Rasmussen and Williams (2006). Gaussian Processes for Machine Learning.

create_model()

Creates and returns the PyMC3 model.

Note: The size of the shared variables must match the size of the training data. Otherwise, setting the shared variables later will raise an error. See http://docs.pymc.io/advanced_theano.html

Returns model

Return type the PyMC3 model.

load(*file_prefix*)
Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns custom_params

Return type Dictionary of custom parameters

save(*file_prefix*)
Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

class pmlearn.gaussian_process.**StudentsTProcessRegressor**(*prior_mean=None*, *kernel=None*)
Bases: *pmlearn.base.BayesianModel*, *pmlearn.gaussian_process.gpr.GaussianProcessRegressorMixin*

StudentsT Process Regression built using PyMC3.

Fit a StudentsT process model and estimate model parameters using MCMC algorithms or Variational Inference algorithms

Parameters **prior_mean** (*mean object*) – The mean specifying the mean function of the StudentsT process. If None is passed, the mean "pm.gp.mean.Zero()" is used as default.

Examples

```
>>> from sklearn.datasets import make_friedman2
>>> from pmlearn.gaussian_process import StudentsTProcessRegressor
>>> from sklearn.gaussian_process.kernels import DotProduct, WhiteKernel
>>> X, y = make_friedman2(n_samples=500, noise=0, random_state=0)
>>> kernel = DotProduct() + WhiteKernel()
>>> spr = StudentsTProcessRegressor(kernel=kernel).fit(X, y)
>>> spr.score(X, y)
0.3680...
```

(continues on next page)

(continued from previous page)

```
>>> spr.predict(X[:2,:], return_std=True)
(array([653.0..., 592.1...]), array([316.6..., 316.6...]))
```

Rasmussen and Williams (2006). Gaussian Processes for Machine Learning.

create_model()
Creates and returns the PyMC3 model.

Note: The size of the shared variables must match the size of the training data. Otherwise, setting the shared variables later will raise an error. See http://docs.pymc.io/advanced_theano.html

Returns **model**

Return type the PyMC3 model

load(*file_prefix*)
Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns **custom_params**

Return type Dictionary of custom parameters

save(*file_prefix*)
Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

class pmlearn.gaussian_process.**SparseGaussianProcessRegressor**(*prior_mean=None*, *kernel=None*)
Bases: *pmlearn.base.BayesianModel*, *pmlearn.gaussian_process.gpr.GaussianProcessRegressorMixin*

Sparse Gaussian Process Regression built using PyMC3.

Fit a Sparse Gaussian process model and estimate model parameters using MCMC algorithms or Variational Inference algorithms

Parameters **prior_mean** (*mean object*) – The mean specifying the mean function of the GP. If None is passed, the mean "pm.gp.mean.Zero()" is used as default.

Examples

```
>>> from sklearn.datasets import make_friedman2
>>> from pmlearn.gaussian_process import SparseGaussianProcessRegressor
>>> from sklearn.gaussian_process.kernels import DotProduct, WhiteKernel
>>> X, y = make_friedman2(n_samples=500, noise=0, random_state=0)
>>> kernel = DotProduct() + WhiteKernel()
>>> sgpr = SparseGaussianProcessRegressor(kernel=kernel).fit(X, y)
>>> sgpr.score(X, y)
0.3680...
>>> sgpr.predict(X[:2,:], return_std=True)
(array([653.0..., 592.1...]), array([316.6..., 316.6...]))
```

Rasmussen and Williams (2006). Gaussian Processes for Machine Learning.

create_model()
Creates and returns the PyMC3 model.

Note: The size of the shared variables must match the size of the training data. Otherwise, setting the shared variables later will raise an error. See http://docs.pymc.io/advanced_theano.html

Returns **model**

Return type the PyMC3 model

load(*file_prefix*)
Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns **custom_params**

Return type Dictionary of custom parameters

save(*file_prefix*)
Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

pmlearn.linear_model package

Subpackages

pmlearn.linear_model.tests package

Submodules

pmlearn.linear_model.tests.test_base module

Testing for Linear regression

class pmlearn.linear_model.tests.test_base.**TestLinearRegression**
Bases: object

setup_method()

teardown_method()

class pmlearn.linear_model.tests.test_base.**TestLinearRegressionFit**
Bases: *pmlearn.linear_model.tests.test_base.TestLinearRegression*

test_advi_fit_returns_correct_model()

test_nuts_fit_returns_correct_model()

class pmlearn.linear_model.tests.test_base.**TestLinearRegressionPredict**
Bases: *pmlearn.linear_model.tests.test_base.TestLinearRegression*

test_predict_raises_error_if_not_fit()

test_predict_returns_mean_predictions_and_std()

test_predict_returns_predictions()

class pmlearn.linear_model.tests.test_base.**TestLinearRegressionSaveandLoad**
Bases: *pmlearn.linear_model.tests.test_base.TestLinearRegression*

test_save_and_load_work_correctly()

class pmlearn.linear_model.tests.test_base.**TestLinearRegressionScore**
Bases: *pmlearn.linear_model.tests.test_base.TestLinearRegression*

test_score_matches_sklearn_performance()

pmlearn.linear_model.tests.test_logistic module

Testing for Logistic regression

class pmlearn.linear_model.tests.test_logistic.**TestHierarchicalLogisticRegression**
Bases: object

setup_method()

teardown_method()

class pmlearn.linear_model.tests.test_logistic.**TestHierarchicalLogisticRegressionFit**
Bases: *pmlearn.linear_model.tests.test_logistic.TestHierarchicalLogisticRegression*

test_advi_fit_returns_correct_model()

class pmlearn.linear_model.tests.test_logistic.**TestHierarchicalLogisticRegressionPredict**
Bases: *pmlearn.linear_model.tests.test_logistic.TestHierarchicalLogisticRegression*

test_predict_returns_predictions()

class pmlearn.linear_model.tests.test_logistic.**TestHierarchicalLogisticRegressionPredictPro**
Bases: *pmlearn.linear_model.tests.test_logistic.TestHierarchicalLogisticRegression*

test_predict_proba_raises_error_if_not_fit()

test_predict_proba_returns_probabilities()

test_predict_proba_returns_probabilities_and_std()

class pmlearn.linear_model.tests.test_logistic.**TestHierarchicalLogisticRegressionSaveandLoa**
Bases: *pmlearn.linear_model.tests.test_logistic.TestHierarchicalLogisticRegression*

test_save_and_load_work_correctly()

class pmlearn.linear_model.tests.test_logistic.**TestHierarchicalLogisticRegressionScore**
Bases: *pmlearn.linear_model.tests.test_logistic.TestHierarchicalLogisticRegression*

test_score_scores()

Module contents

Submodules

pmlearn.linear_model.base module

Generalized Linear models.

class pmlearn.linear_model.base.**BayesianLinearClassifierMixin**
Bases: sklearn.base.ClassifierMixin

Mixin for linear classifiers models in pmlearn

fit(*X*, *y*, *cats*, *inference_type='advi'*, *minibatch_size=None*, *inference_args=None*)
Train the Hierarchical Logistic Regression model

Parameters

- **X** (*numpy array, shape [n_samples, n_features]*) –
- **y** (*numpy array, shape [n_samples,]*) –
- **cats** (*numpy array, shape [n_samples,]*) –
- **inference_type** (*string, specifies which inference method to call.*) – Defaults to 'advi'. Currently, only 'advi' and 'nuts' are supported
- **minibatch_size** (*number of samples to include in each minibatch for*) – ADVI, defaults to None, so minibatch is not run by default
- **inference_args** (*dict, arguments to be passed to the inference methods.*) – Check the PyMC3 docs for permissable values. If no arguments are specified, default values will be set.

predict(*X*, *cats*)
Predicts labels of new data with a trained model

Parameters

- **X** (*numpy array, shape [n_samples, n_features]*) –
- **cats** (*numpy array, shape [n_samples,]*) –

predict_proba(*X*, *cats*, *return_std=False*)
Predicts probabilities of new data with a trained Hierarchical Logistic Regression

Parameters

- **X** (*numpy array, shape [n_samples, n_features]*) –
- **cats** (*numpy array, shape [n_samples,]*) –
- **return_std** (*Boolean flag of whether to return standard deviations with*) –
- **probabilities. Defaults to False.** (*mean*) –

score(*X*, *y*, *cats*)
Scores new data with a trained model.

Parameters

- **X** (*numpy array, shape [n_samples, n_features]*) –
- **y** (*numpy array, shape [n_samples,]*) –
- **cats** (*numpy array, shape [n_samples,]*) –

class pmlearn.linear_model.base.**LinearRegression**
Bases: *pmlearn.base.BayesianModel*, *pmlearn.base.BayesianRegressorMixin*

Linear Regression built using PyMC3.

create_model()
Creates and returns the PyMC3 model.

Note: The size of the shared variables must match the size of the training data. Otherwise, setting the shared variables later will raise an error. See http://docs.pymc.io/advanced_theano.html

Returns

Return type the PyMC3 model

load(*file_prefix*)
Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns **custom_params**

Return type Dictionary of custom parameters

save(*file_prefix*)
Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

pmlearn.linear_model.logistic module

Logistic regression models.

class pmlearn.linear_model.logistic.**HierarchicalLogisticRegression**
Bases: *pmlearn.base.BayesianModel*, *pmlearn.linear_model.base.BayesianLinearClassifierMixin*

Custom Hierachical Logistic Regression built using PyMC3.

create_model()
Creates and returns the PyMC3 model.

Note: The size of the shared variables must match the size of the training data. Otherwise, setting the shared variables later will raise an error. See http://docs.pymc.io/advanced_theano.html

Returns

Return type the PyMC3 model

load(*file_prefix*)
Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns **custom_params**

Return type Dictionary of custom parameters

save(*file_prefix*)
Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"

- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

class pmlearn.linear_model.logistic.**LogisticRegression**
Bases: *pmlearn.base.BayesianModel*, *pmlearn.linear_model.base.BayesianLinearClassifierMixin*

Bayesian Logistic Regression built using PyMC3

create_model()
Creates and returns the PyMC3 model.

Note: The size of the shared variables must match the size of the training data. Otherwise, setting the shared variables later will raise an error. See http://docs.pymc.io/advanced_theano.html

Returns

Return type the PyMC3 model

load(*file_prefix*)
Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns **custom_params**

Return type Dictionary of custom parameters

save(*file_prefix*)
Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

Module contents

The *pmlearn.linear_model* module implements generalized linear models. It includes Bayesian Regression, Bayesian Logistic Regression, Hierarchical Logistic Regression.

class pmlearn.linear_model.**LinearRegression**
Bases: *pmlearn.base.BayesianModel*, *pmlearn.base.BayesianRegressorMixin*

Linear Regression built using PyMC3.

create_model()
Creates and returns the PyMC3 model.

Note: The size of the shared variables must match the size of the training data. Otherwise, setting the shared variables later will raise an error. See http://docs.pymc.io/advanced_theano.html

Returns

Return type the PyMC3 model

load(*file_prefix*)
Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns custom_params

Return type Dictionary of custom parameters

save(*file_prefix*)
Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

class pmlearn.linear_model.**HierarchicalLogisticRegression**
Bases: *pmlearn.base.BayesianModel*, *pmlearn.linear_model.base.BayesianLinearClassifierMixin*

Custom Hierachical Logistic Regression built using PyMC3.

create_model()
Creates and returns the PyMC3 model.

Note: The size of the shared variables must match the size of the training data. Otherwise, setting the shared variables later will raise an error. See http://docs.pymc.io/advanced_theano.html

Returns

Return type the PyMC3 model

load(*file_prefix*)
Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns **custom_params**

Return type Dictionary of custom parameters

save(*file_prefix*)

Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

class pmlearn.linear_model.**LogisticRegression**

Bases: *pmlearn.base.BayesianModel*, *pmlearn.linear_model.base.BayesianLinearClassifierMixin*

Bayesian Logistic Regression built using PyMC3

create_model()

Creates and returns the PyMC3 model.

Note: The size of the shared variables must match the size of the training data. Otherwise, setting the shared variables later will raise an error. See http://docs.pymc.io/advanced_theano.html

Returns

Return type the PyMC3 model

load(*file_prefix*)

Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns **custom_params**

Return type Dictionary of custom parameters

save (*file_prefix*)
Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

pmlearn.mixture package

Subpackages

pmlearn.mixture.tests package

Submodules

pmlearn.mixture.tests.test_dirichlet_process module

class pmlearn.mixture.tests.test_dirichlet_process.**DirichletProcessMixturePredictTestCase**(*
Bases: *pmlearn.mixture.tests.test_dirichlet_process.DirichletProcessMixtureTestCase*

test_predict_raises_error_if_not_fit()

class pmlearn.mixture.tests.test_dirichlet_process.**DirichletProcessMixtureTestCase**(*methodNa*
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

tearDown()
Hook method for deconstructing the test fixture after testing it.

pmlearn.mixture.tests.test_gaussian_mixture module

class pmlearn.mixture.tests.test_gaussian_mixture.**GaussianMixturePredictTestCase**(*methodName=*
Bases: *pmlearn.mixture.tests.test_gaussian_mixture.GaussianMixtureTestCase*

test_predict_raises_error_if_not_fit()

class pmlearn.mixture.tests.test_gaussian_mixture.**GaussianMixtureTestCase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

tearDown()
Hook method for deconstructing the test fixture after testing it.

Module contents

Submodules

pmlearn.mixture.dirichlet_process module

Dirichlet Process Mixture Model.

class pmlearn.mixture.dirichlet_process.**DirichletProcessMixture**
Bases: *pmlearn.base.BayesianModel*, *pmlearn.base.BayesianDensityMixin*

Custom Dirichlet Process Mixture Model built using PyMC3.

create_model()
Creates and returns the PyMC3 model.

Note: The size of the shared variables must match the size of the training data. Otherwise, setting the shared variables later will raise an error. See http://docs.pymc.io/advanced_theano.html

The DensityDist class is used as the likelihood term. The second argument, logp_gmix(mus, pi, np.eye(D)), is a python function which recieves observations (denoted by 'value') and returns the tensor representation of the log-likelihood.

Returns

Return type the PyMC3 model

load(*file_prefix*)
Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns **custom_params**

Return type Dictionary of custom parameters

predict_proba(*X*, *return_std=False*)
Predicts probabilities of new data with a trained Dirichlet Process Mixture Model

Parameters

- **X** (*numpy array, shape [n_samples, n_features]*) –
- **cats** (*numpy array, shape [n_samples,]*) –
- **return_std** (*Boolean flag*) – Boolean flag of whether to return standard deviations with mean probabilities. Defaults to False.

save(*file_prefix*)
Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

pmlearn.mixture.gaussian_mixture module

Gaussian Mixture Model.

class pmlearn.mixture.gaussian_mixture.**GaussianMixture**

Bases: *pmlearn.base.BayesianModel*, *pmlearn.base.BayesianDensityMixin*

Custom Gaussian Mixture Model built using PyMC3.

create_model()

Creates and returns the PyMC3 model.

Note: The size of the shared variables must match the size of the training data. Otherwise, setting the shared variables later will raise an error. See http://docs.pymc.io/advanced_theano.html

Returns

Return type the PyMC3 model

load(*file_prefix*)

Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns **custom_params**

Return type Dictionary of custom parameters

save(*file_prefix*)

Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

pmlearn.mixture.util module

class pmlearn.mixture.util.**logp_gmix**(*mus*, *pi*, *tau*, *num_training_samples*)
Bases: object

Module contents

The *pmlearn.mixture* module implements mixture models.

class pmlearn.mixture.**GaussianMixture**
Bases: *pmlearn.base.BayesianModel*, *pmlearn.base.BayesianDensityMixin*

Custom Gaussian Mixture Model built using PyMC3.

create_model()
Creates and returns the PyMC3 model.

Note: The size of the shared variables must match the size of the training data. Otherwise, setting the shared variables later will raise an error. See http://docs.pymc.io/advanced_theano.html

Returns

Return type the PyMC3 model

load(*file_prefix*)
Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns **custom_params**

Return type Dictionary of custom parameters

save(*file_prefix*)
Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

class pmlearn.mixture.**DirichletProcessMixture**
Bases: *pmlearn.base.BayesianModel*, *pmlearn.base.BayesianDensityMixin*

Custom Dirichlet Process Mixture Model built using PyMC3.

create_model()
Creates and returns the PyMC3 model.

Note: The size of the shared variables must match the size of the training data. Otherwise, setting the shared variables later will raise an error. See http://docs.pymc.io/advanced_theano.html

The DensityDist class is used as the likelihood term. The second argument, logp_gmix(mus, pi, np.eye(D)), is a python function which recieves observations (denoted by 'value') and returns the tensor representation of the log-likelihood.

Returns

Return type the PyMC3 model

load(*file_prefix*)
Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns custom_params

Return type Dictionary of custom parameters

predict_proba(*X*, *return_std=False*)
Predicts probabilities of new data with a trained Dirichlet Process Mixture Model

Parameters

- **X** (*numpy array, shape [n_samples, n_features]*) –
- **cats** (*numpy array, shape [n_samples,]*) –
- **return_std** (*Boolean flag*) – Boolean flag of whether to return standard deviations with mean probabilities. Defaults to False.

save(*file_prefix*)
Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

pmlearn.naive_bayes package

Subpackages

pmlearn.naive_bayes.tests package

Submodules

pmlearn.naive_bayes.tests.test_naive_bayes module

Module contents

Submodules

pmlearn.naive_bayes.naive_bayes module

Naive Bayes models.

class pmlearn.naive_bayes.naive_bayes.**GaussianNB**

Bases: *pmlearn.base.BayesianModel*, *pmlearn.naive_bayes.naive_bayes.GaussianNBClassifierMixin*

Gaussian Naive Bayes (GaussianNB) classification built using PyMC3.

The Gaussian Naive Bayes algorithm assumes that the random variables that describe each class and each feature are independent and distributed according to Normal distributions.

Example

```
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> Y = np.array([1, 1, 1, 2, 2, 2])
>>> from pmlearn.naive_bayes import GaussianNB
>>> clf = GaussianNB()
>>> clf.fit(X, Y)
GaussianNB(priors=None, var_smoothing=1e-09)
>>> print(clf.predict([[-0.8, -1]]))
[1]
>>> clf_pf = GaussianNB()
>>> clf_pf.partial_fit(X, Y, np.unique(Y))
GaussianNB(priors=None, var_smoothing=1e-09)
>>> print(clf_pf.predict([[-0.8, -1]]))
[1]
```

See the documentation of the *create_model* method for details on the model itself.

create_model()

Creates and returns the PyMC3 model.

We note x_{jc} the value of the j-th element of the data vector x conditioned on x belonging to the class c. The Gaussian Naive Bayes algorithm models x_{jc} as:

$$x_{jc} \sim Normal(\mu_{jc}, \sigma_{jc})$$

While the probability that x belongs to the class c is given by the categorical distribution:

$$P(y = c|x_i) = Cat(\pi_1, \ldots, \pi_C)$$

where π_i is the probability that a vector belongs to category i.

We assume that the π_i follow a Dirichlet distribution:

$$\pi \sim Dirichlet(\alpha)$$

with hyperparameter $\alpha = [1, .., 1]$. The μ_{jc} are sampled from a Normal distribution centred on 0 with variance 100, and the σ_{jc} are sampled from a HalfNormal distribuion of variance 100:

$$\mu_{jc} \sim Normal(0, 100)$$
$$\sigma_{jc} \sim HalfNormal(100)$$

Note that the Gaussian Naive Bayes model is equivalent to a Gaussian mixture with a diagonal covariance [1].

Returns

Return type A PyMC3 model

References

perspective.

load (*file_profile*)
Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns custom_params

Return type Dictionary of custom parameters

save (*file_prefix*)
Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

class pmlearn.naive_bayes.naive_bayes.**GaussianNBClassifierMixin**

Bases: *pmlearn.base.BayesianClassifierMixin*

Mixin class for naive Bayes classifiers

fit(*X*, *y*, *inference_type='advi'*, *minibatch_size=None*, *inference_args=None*)

Train the Naive Bayes model.

Parameters

- **X** (*numpy array, shape [num_training_samples, num_pred]*) – Contains the data points.
- **y** (*numpy array, shape [num_training_samples,]*) – Contains the category of the data points.
- **inference_type** (*string, specifies which inference method to call.*) – Default is 'advi'. Currently, only 'advi' and 'nuts' are implemented.
- **minibatch_size** (*int, number of samples to include in each minibatch*) – for ADVI. Defaults to None so minibatch is not run by default.
- **inference_args** (*dict, arguments to be passed to the inference methods.*) – Check the PyMC3 documentation.

Returns

Return type The current instance of the GaussianNB class.

normalize(*array*)

Normalize values in the array to get probabilities. :param array: :type array: numpy array of shape [1,]

Returns

Return type A normalized array

predict(*X*)

Classify new data with a trained Naive Bayes model. The output is the point estimate of the posterior predictive distribution that corresponds to the one-hot loss function.

Parameters **X** (*numpy array, shape [num_training_samples, num_pred]*) – Contains the data to classify.

Returns

- *A numpy array of shape [num_training_samples,] that contains the*
- *predicted class to which the data points belong.*

predict_proba(*X*)

Predicts the probabilities that data points belong to each category.

Given a new data point $\vec{x}$, we want to estimate the probability that it belongs to a category c. Following the notations in [1], the probability reads:

$$P(y = c|\vec{x}, \mathcal{D}) = P(y = c|\mathcal{D}) \prod_{j=1}^{n_{dims}} P(x_j|y = c, \mathcal{D})$$

We previously used the data $\mathcal{D}$ to estimate the distribution of the parameters $\vec{\mu}$, $\vec{\pi}$ and $\vec{\sigma}$. To compute the above probability, we need to integrate over the values of these parameters:

$$P(y = c|\vec{x}, \mathcal{D}) = \left[\int Cat(y = c|\vec{\pi}) P(\vec{\pi}|\ \mathcal{D}) \mathrm{d}\vec{\pi}\right] \int P(\vec{x}|\vec{\mu}, \vec{\sigma})\ P(\vec{\mu}|\mathcal{D})\ P(\vec{\sigma}|\mathcal{D})\ \mathrm{d}\vec{\mu}\mathrm{d}\vec{\sigma}$$

Parameters **X** (*numpy array, shape [num_training_samples, num_pred]*) – Contains the points for which we want to predict the class

Returns

- *A numpy array of shape [num_training_samples, num_cats] that contains*
- *the probabilities that each sample belong to each category.*

References

perspective.

Module contents

The *pmlearn.naive_bayes* module implements Naive Bayes algorithms. These are supervised learning methods based on applying Bayes' theorem with strong (naive) feature independence assumptions.

class pmlearn.naive_bayes.**GaussianNB**

Bases: *pmlearn.base.BayesianModel*, *pmlearn.naive_bayes.naive_bayes.GaussianNBClassifierMixin*

Gaussian Naive Bayes (GaussianNB) classification built using PyMC3.

The Gaussian Naive Bayes algorithm assumes that the random variables that describe each class and each feature are independent and distributed according to Normal distributions.

Example

```
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> Y = np.array([1, 1, 1, 2, 2, 2])
>>> from pmlearn.naive_bayes import GaussianNB
>>> clf = GaussianNB()
>>> clf.fit(X, Y)
GaussianNB(priors=None, var_smoothing=1e-09)
>>> print(clf.predict([[-0.8, -1]]))
[1]
>>> clf_pf = GaussianNB()
>>> clf_pf.partial_fit(X, Y, np.unique(Y))
GaussianNB(priors=None, var_smoothing=1e-09)
>>> print(clf_pf.predict([[-0.8, -1]]))
[1]
```

See the documentation of the *create_model* method for details on the model itself.

create_model()

Creates and returns the PyMC3 model.

We note x_{jc} the value of the j-th element of the data vector x conditioned on x belonging to the class c. The Gaussian Naive Bayes algorithm models x_{jc} as:

$$x_{jc} \sim Normal(\mu_{jc}, \sigma_{jc})$$

While the probability that x belongs to the class c is given by the categorical distribution:

$$P(y = c|x_i) = Cat(\pi_1, \ldots, \pi_C)$$

where π_i is the probability that a vector belongs to category i.

We assume that the π_i follow a Dirichlet distribution:

$$\pi \sim Dirichlet(\alpha)$$

with hyperparameter $\alpha = [1, .., 1]$. The μ_{jc} are sampled from a Normal distribution centred on 0 with variance 100, and the σ_{jc} are sampled from a HalfNormal distribuion of variance 100:

$$\mu_{jc} \sim Normal(0, 100)$$
$$\sigma_{jc} \sim HalfNormal(100)$$

Note that the Gaussian Naive Bayes model is equivalent to a Gaussian mixture with a diagonal covariance [1].

Returns

Return type A PyMC3 model

References

perspective.

load(*file_profile*)
Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns custom_params

Return type Dictionary of custom parameters

save(*file_prefix*)
Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

pmlearn.neural_network package

Subpackages

pmlearn.neural_network.tests package

Submodules

pmlearn.neural_network.tests.test_mutlilayer_perceptron module

Module contents

Submodules

pmlearn.neural_network.multilayer_perceptron module

Multilayer perceptron

class pmlearn.neural_network.multilayer_perceptron.**MLPClassifier**(*n_hidden=5*)
Bases: *pmlearn.base.BayesianModel*, *pmlearn.base.BayesianClassifierMixin*

Multilayer perceptron classification built using PyMC3.

Fit a Multilayer perceptron classification model and estimate model parameters using MCMC algorithms or Variational Inference algorithms

Examples

http://twiecki.github.io/blog/2016/06/01/bayesian-deep-learning/

create_model()

load(*file_prefix*)
Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns custom_params

Return type Dictionary of custom parameters

save(*file_prefix*)
Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

Module contents

The *pmlearn.neural_network* module includes models based on neural networks.

class pmlearn.neural_network.**MLPClassifier**(*n_hidden=5*)

Bases: *pmlearn.base.BayesianModel*, *pmlearn.base.BayesianClassifierMixin*

Multilayer perceptron classification built using PyMC3.

Fit a Multilayer perceptron classification model and estimate model parameters using MCMC algorithms or Variational Inference algorithms

Examples

http://twiecki.github.io/blog/2016/06/01/bayesian-deep-learning/

create_model()

load(*file_prefix*)

Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = "path/to/file/" This will attempt to load "path/to/file/trace.pickle"
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns **custom_params**

Return type Dictionary of custom parameters

save(*file_prefix*)

Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = "path/to/file/" This will attempt to save to "path/to/file/trace.pickle"
- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

pmlearn.tests package

Submodules

pmlearn.tests.test_base module

Unit tests for *pmlearn.base*.

class pmlearn.tests.test_base.**TestBayesianModel**
Bases: object

Test class for base bayesian model

test_create_model_raises_not_implemented_error()
Assert that NotImplementedError is raised

Module contents

Submodules

pmlearn.base module

Base classes for all Bayesian models.

class pmlearn.base.**BayesianClassifierMixin**
Bases: sklearn.base.ClassifierMixin

Mixin for regression models in pmlearn

fit(*X*, *y*, *inference_type='advi'*, *minibatch_size=None*, *inference_args=None*)
Train the Multilayer perceptron model

Parameters

- **X** (*numpy array, shape [n_samples, n_features]*) –
- **y** (*numpy array, shape [n_samples,]*) –
- **inference_type** (*string, specifies which inference method to call.*) –
- **to 'advi'. Currently, only 'advi' and 'nuts' are supported** (*Defaults*) –
- **minibatch_size** (*number of samples to include in each minibatch*) –
- **ADVI, defaults to None, so minibatch is not run by default** (*for*) –
- **inference_args** (*dict, arguments to be passed to the inference methods.*) –
- **the PyMC3 docs for permissable values. If no arguments are** (*Check*) –
- **default values will be set.** (*specified,*) –

predict(*X*)
Predicts labels of new data with a trained model

Parameters **X** (*numpy array, shape [n_samples, n_features]*) –

predict_proba(*X*, *return_std=False*)
Perform Prediction

Predicts values of new data with a trained Gaussian Process Regression model

Parameters

- **X** (*numpy array, shape [n_samples, n_features]*) –
- **return_std** (*Boolean*) – Whether to return standard deviations with mean values. Defaults to False.

class pmlearn.base.**BayesianDensityMixin**
Bases: sklearn.base.DensityMixin

Mixin for regression models in pmlearn

fit(*X*, *num_components*, *inference_type='advi'*, *minibatch_size=None*, *inference_args=None*)
Train the Gaussian Mixture Model model

Parameters

- **X** (*numpy array, shape [n_samples, n_features]*) –
- **n_truncate** (*numpy array, shape [n_samples,]*) –
- **inference_type** (*string, specifies which inference method to call.*) –
- **to 'advi'. Currently, only 'advi' and 'nuts' are supported** (*Defaults*) –
- **minibatch_size** (*number of samples to include in each minibatch for*) –
- **ADVI,** –
- **to None, so minibatch is not run by default** (*defaults*) –
- **inference_args** (*dict, arguments to be passed to the inference methods.*) –
- **the PyMC3 docs for permissable values. If no arguments are** (*Check*) –
- **specified,** –
- **values will be set.** (*default*) –

predict(*X*)
Predicts labels of new data with a trained model

Parameters

- **X** (*numpy array, shape [n_samples, n_features]*) –
- **cats** (*numpy array, shape [n_samples,]*) –

predict_proba(*X*, *return_std=False*)
Predicts probabilities of new data with a trained GaussianMixture Model

Parameters

- **X** (*numpy array, shape [n_samples, n_features]*) –
- **cats** (*numpy array, shape [n_samples,]*) –

- **return_std** (*Boolean flag of whether to return standard deviations with*) –
- **probabilities. Defaults to False.** (*mean*) –

score(*X*, *y*, *cats*)
Scores new data with a trained model.

Parameters

- **X** (*numpy array, shape [n_samples, n_features]*) –
- **y** (*numpy array, shape [n_samples,]*) –
- **cats** (*numpy array, shape [n_samples,]*) –

class pmlearn.base.**BayesianModel**
Bases: sklearn.base.BaseEstimator

Base class for all Bayesian models in pymc-learn

Notes

All Bayesian models should specify all the parameters that can be set at the class level in their __init__ as explicit keyword arguments (no *args or **kwargs‘‘).

create_model()
Create model

load(*file_prefix*, *load_custom_params=False*)
Loads a saved version of the trace, and custom param files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to load the*) –
- **trace for this model.** (*saved*) – Ex: given file_prefix = “path/to/file/” This will attempt to load “path/to/file/trace.pickle”
- **load_custom_params** (*Boolean flag to indicate whether custom parameters*) –
- **be loaded. Defaults to False.** (*should*) –

Returns **custom_params**

Return type Dictionary of custom parameters

plot_elbo()
Plot the ELBO values after running ADVI minibatch.

save(*file_prefix*, *custom_params=None*)
Saves the trace and custom params to files with the given file_prefix.

Parameters

- **file_prefix** (*str, path and prefix used to identify where to save the*) –
- **for this model.** (*trace*) – Ex: given file_prefix = “path/to/file/” This will attempt to save to “path/to/file/trace.pickle”

- **custom_params** (*Dictionary of custom parameters to save.*) – Defaults to None

class pmlearn.base.**BayesianRegressorMixin**
Bases: sklearn.base.RegressorMixin

Mixin for regression models in pmlearn

fit(*X*, *y*, *inference_type='advi'*, *minibatch_size=None*, *inference_args=None*)
Train the Linear Regression model

Parameters

- **X** (*numpy array, shape [n_samples, n_features]*) –
- **y** (*numpy array, shape [n_samples,]*) –
- **inference_type** (*string, specifies which inference method to call.*) – Defaults to 'advi'. Currently, only 'advi' and 'nuts' are supported
- **minibatch_size** (*number of samples to include in each minibatch for*) – ADVI, defaults to None, so minibatch is not run by default
- **inference_args** (*dict, arguments to be passed to the inference methods.*) – Check the PyMC3 docs for permissable values. If no arguments are specified, default values will be set.

predict(*X*, *return_std=False*)
Predicts values of new data with a trained Linear Regression model

Parameters

- **X** (*numpy array, shape [n_samples, n_features]*) –
- **return_std** (*Boolean flag*) – Boolean flag of whether to return standard deviations with mean values. Defaults to False.

pmlearn.exceptions module

The *pmlearn.exceptions* module includes all custom warnings and error classes used across pymc-learn.

exception pmlearn.exceptions.**NotFittedError**
Bases: ValueError, AttributeError

Exception class to raise if estimator is used before fitting. This class inherits from both ValueError and AttributeError to help with exception handling and backward compatibility. .. rubric:: Examples

```
>>> from pmlearn.gaussian_process import GaussianProcessRegressor
>>> from pmlearn.exceptions import NotFittedError
>>> try:
...     GaussianProcessRegressor().predict([[1, 2], [2, 3], [3, 4]])
... except NotFittedError as e:
...     print(repr(e))
...
NotFittedError('This GaussianProcessRegressor instance is not fitted yet'.)
```

Module contents

Probabilistic machine learning module for Python

pmlearn is a Python module for practical probabilistic machine learning built on top of scikit-learn and PymC3.

It aims to provide simple and efficient solutions to learning problems that are accessible to everybody and reusable in various contexts: machine-learning as a versatile tool for science and engineering. See http://pymc-learn.org for complete documentation.

Help & reference

7.10 Contributing

Thank you for considering contributing to `pymc-learn`! Please read these guidelines before submitting anything to the project.

Some ways to contribute:

- Open an issue on the Github Issue Tracker. (Please check that it has not already been reported or addressed in a PR.)
- Improve the docs!
- Add a new machine-learning model. Please follow the guidelines below.
- Add/change existing functionality in the base function classes for ML.
- Something I haven't thought of?

7.10.1 Pull/Merge Requests

To create a Pull Request against this library, please fork the project and work from there.

Steps

1. Fork the project via the Fork button on Github
2. Clone the repo to your local disk, and add the base repository as a remote.

```
git clone https://github/<YOUR-GITHUB-USERNAME>/pymc-learn.git
cd pymc-learn
git remote add upstream https://github.com/pymc-learn/pymc-learn.git
```

3. Create a new branch for your PR.

```
git checkout -b my-new-feature-branch
```

Always use a `feature` branch. It's good practice to never routinely work on the `master` branch.

4. Install requirements (probably in a virtual environment)

```
conda create --name myenv python=3.6 pip
conda activate myenv
pip install -r requirements.txt
pip install -r requirements_dev.txt
```

NOTE: On Windows, in your Anaconda Prompt, run `activate myenv`.

5. Develop your feature. Add changed files using `git add` and then `git commit` files:

```
git add <my_new_model.py>
git commit
```

to record your changes locally. After committing, it is a good idea to sync with the base repository in case there have been any changes:

```
git fetch upstream
git rebase upstream/master
```

Then push the changes to your Github account with:

```
git push -u origin my-new-feature-branch
```

6. Submit a Pull Request! Go to the Github web page of your fork of the `pymc-learn` repo. Click the 'Create pull request' button to send your changes to the project maintainers for review. This will send an email to the committers.

Pull Request Checklist

- Ensure your code has followed the Style Guidelines below
- Make sure you have written tests where appropriate
- Make sure the tests pass

```
conda activate myenv
python -m pytest
```

NOTE: On Windows, in your Anaconda Prompt, run `activate myenv`.

- Update the docs where appropriate. You can rebuild them with the commands below.

```
cd pymc-learn/docs
sphinx-apidoc -f -o api/ ../pmlearn/
make html
```

- Update the CHANGELOG

Style Guidelines

For the most part, this library follows PEP8 with a couple of exceptions.

Notes:

- Indent with 4 spaces
- Lines can be 80 characters long
- Docstrings should be written as numpy docstrings
- Your code should be Python 3 compatible
- When in doubt, follow the style of the existing code

Contact

To report an issue with `pymc-learn` please use the issue tracker.

Finally, if you need to get in touch for information about the project, send us an e-mail.

7.10.2 Transitioning from PyMC3 to PyMC4

7.11 Changelog

7.11.1 0.0.1 / 2019-01-01

Models

- Added Gaussian process regression, students t process, sparse gaussian process
- Added Multilayer perceptron

Documentation

- Added docs for above models

7.12 Citations

To cite `pymc-learn` in publications, please use the following:

```
Emaasit, Daniel (2018). Pymc-learn: Practical probabilistic machine
learning in Python. arXiv preprint arXiv:1811.00542.
```

Or using BibTex as follows:

```
@article{emaasit2018pymc,
  title={Pymc-learn: Practical probabilistic machine learning in {P}ython},
  author={Emaasit, Daniel and others},
  journal={arXiv preprint arXiv:1811.00542},
  year={2018}
}
```

If you want to cite `pymc-learn` for its API, you may also want to consider this reference:

```
Carlson, Nicole (2018). Custom PyMC3 models built on top of the scikit-learn
API. https://github.com/parsing-science/pymc3_models
```

Or using BibTex as follows:

```
@article{Pymc3_models,
  title={pymc3_models: Custom PyMC3 models built on top of the scikit-learn API,
  author={Carlson, Nicole},
  journal={},
  url={https://github.com/parsing-science/pymc3_models}
  year={2018}
}
```

7.12.1 Papers using pymc-learn

Emaasit, D., and D, Jones. (2018). Custom PyMC3 nonparametric models built on top of scikit-learn API. The Inaugural International Conference on Probabilistic Programming

```
@InProceedings{ emaasit2018custom,
  author    = { Emaasit, Daniel, and Jones, David },
  title     = { Custom PyMC3 nonparametric models built on top of scikit-learn API},
  booktitle = { The Inaugural International Conference on Probabilistic Programming },
  pages     = {  },
  year      = { 2018 },
  editor    = {  }
}
```

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Bibliography

[RW2006] Carl Eduard Rasmussen and Christopher K.I. Williams, “Gaussian Processes for Machine Learning”, MIT Press 2006, Link to an official complete PDF version of the book here .

Python Module Index

p

Index

H

L

M

N

P

R

S

T

W