PyMC3 Models Documentation

Release 1.0

Nicole Carlson

Jan 11, 2019

Contents

CHAPTER 1

Contents

1.1 Introduction to PyMC3 models

This library was inspired by my own work creating a re-usable Hierarchical Logistic Regression model.

To learn more, you can read this section, watch a video from PyData NYC 2017, or check out the slides .

1.1.1 Quick intro to PyMC3

When building a model with PyMC3, you will usually follow the same four steps:

- **Step 1: Set up** Parameterize your model, choose priors, and insert training data
- **Step 2: Inference** infer your parameters using MCMC sampling (e.g. NUTS) or variational inference (e.g. ADVI)
- **Step 3: Interpret** Check your parameter distributions and model fit
- **Step 4: Predict data** Create posterior samples with your inferred parameters

For a longer discussion of these steps, see *Getting Started*.

1.1.2 Mapping between scikit-learn and PyMC3

This library builds a mapping between the steps above with the methods used by scikit-learn models.

scikit-learn	PyMC3
Fit	Step 1: Set up, Step 2: Inference
Predict	Step 4: Predict Data
Score	Step 4: Predict data
Save/Load	??
??	Step 3: Interpret

The question marks represent things that don't exist in the two libraries on their own.

1.1.3 Comparing scitkit-learn, PyMC3, and PyMC3 Models

Using the mapping above, this library creates easy to use PyMC3 models.

	scikit-learn	PyMC3	PyMC3 models
Find model parameters	Easy	Medium	Easy
Predict new data	Easy	Difficult	Easy
Score a model	Easy	Difficult	Easy
Save a trained model	Easy	Impossible?	Easy
Load a trained model	Easy	Impossible?	Easy
Interpret Parameterization	N/A	Easy	Easy

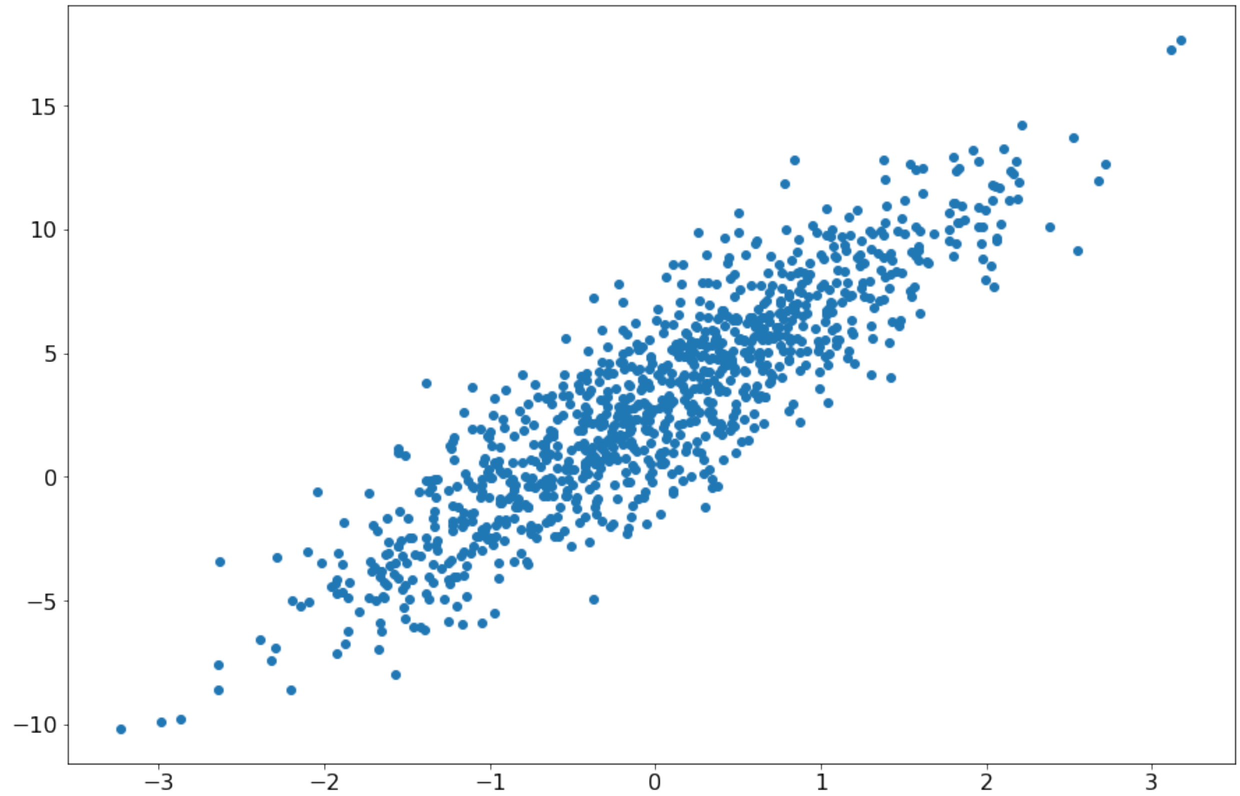
1.2 Getting Started

This section is adapted from my 2017 PyData NYC talk.

To demonstrate how to get started with PyMC3 Models, I'll walk through a simple Linear Regression example. First, I'll go through the example using just PyMC3. Then I'll show you the same example using PyMC3 Models.

1.2.1 Generate Synthetic Data

```
X = np.random.randn(1000, 1)
noise = 2 * np.random.randn(1000, 1)
Y = 4 * X + 3 + noise
```



1.2.2 Fit a model with PyMC3

Step 1: Set up the PyMC3 model

```
lin_reg_model = pm.Model()

model_input = theano.shared(X)

model_output = theano.shared(Y)

with lin_reg_model:

    alpha = pm.Normal('alpha', mu=0, sd=100, shape=(1))
    beta = pm.Normal('beta', mu=0, sd=100, shape=(1))

    s = pm.HalfNormal('s', tau=1)

    mean = alpha + beta * model_input

    y = pm.Normal('y', mu=mean , sd=s, observed=model_output)
```

Step 2: Infer your parameters

```
with lin_reg_model:
    inference = pm.ADVI()
```

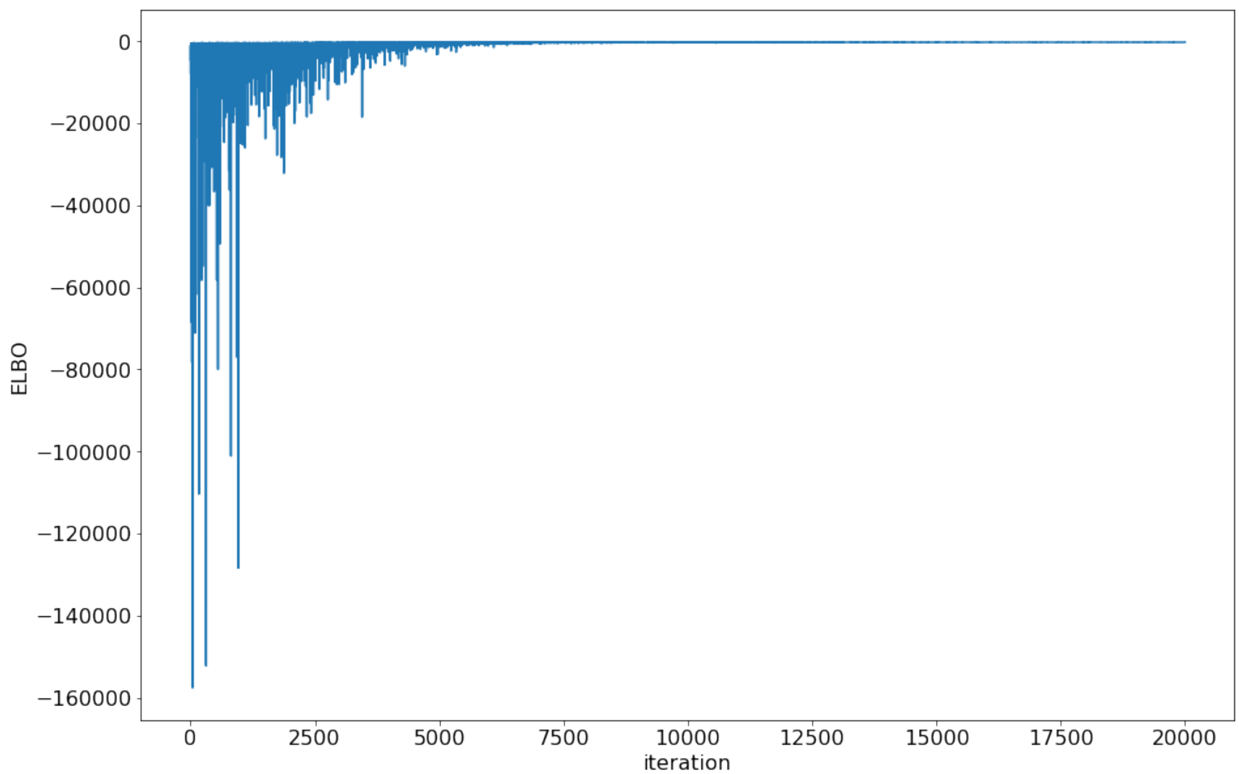
(continues on next page)

(continued from previous page)

```
    approx = pm.fit(
        n=20000,
        method=inference,
        more_replacements={
            model_input: pm.Minibatch(X),
            model_output: pm.Minibatch(Y)
        }
    )
```

Check if minibatch ADVI converged by plotting the ELBO

```
plt.plot(-inference.hist)
plt.ylabel('ELBO')
plt.xlabel('iteration')
```



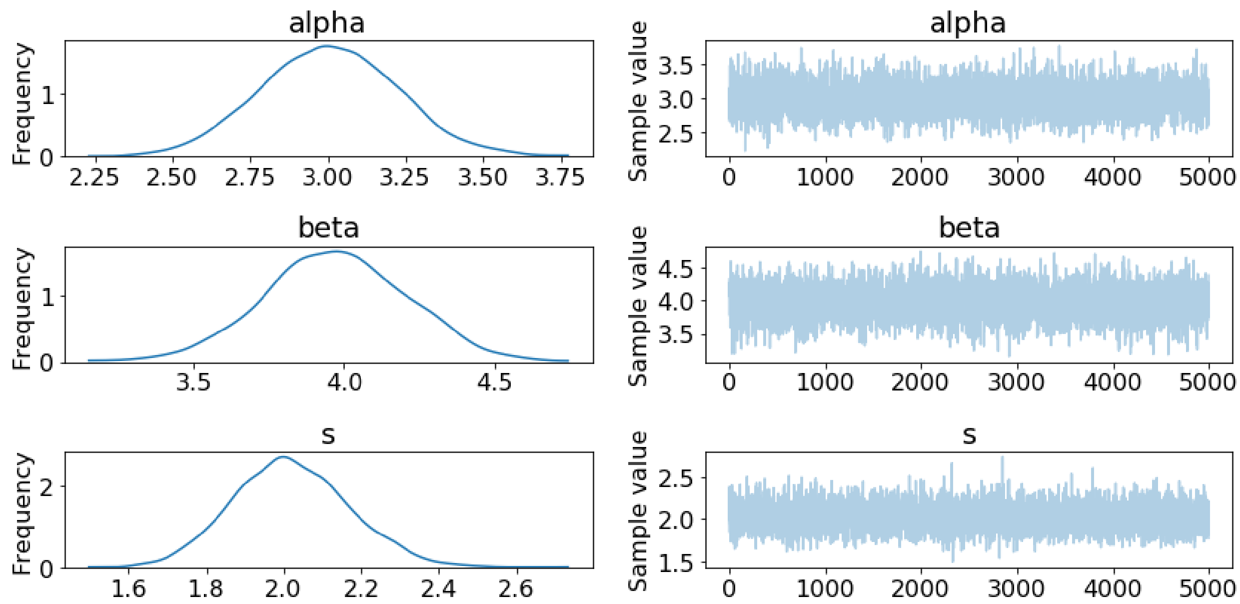
Step 3: Interpret your parameters

To make things a bit easier, I draw samples from the approximation to generate a trace.

```
trace = approx.sample(draws=5000)
summary(trace)
```

```
alpha:

  Mean             SD               MC Error         95% HPD interval
  -------------------------------------------------------------------

  3.002            0.220            0.003            [2.582, 3.440]

  Posterior quantiles:
  2.5             25              50              75              97.5
  |---------------|===============|===============|---------------|

  2.579           2.851           3.000           3.150           3.440


beta:

  Mean             SD               MC Error         95% HPD interval
  -------------------------------------------------------------------

  3.968            0.238            0.003            [3.501, 4.423]

  Posterior quantiles:
  2.5             25              50              75              97.5
  |---------------|===============|===============|---------------|

  3.502           3.809           3.966           4.129           4.424


s:

  Mean             SD               MC Error         95% HPD interval
  -------------------------------------------------------------------

  2.021            0.149            0.002            [1.734, 2.306]

  Posterior quantiles:
  2.5             25              50              75              97.5
  |---------------|===============|===============|---------------|

  1.743           1.918           2.014           2.117           2.322
```

```
traceplot(trace)
```



Step 4: Predict data by creating posterior predictive samples

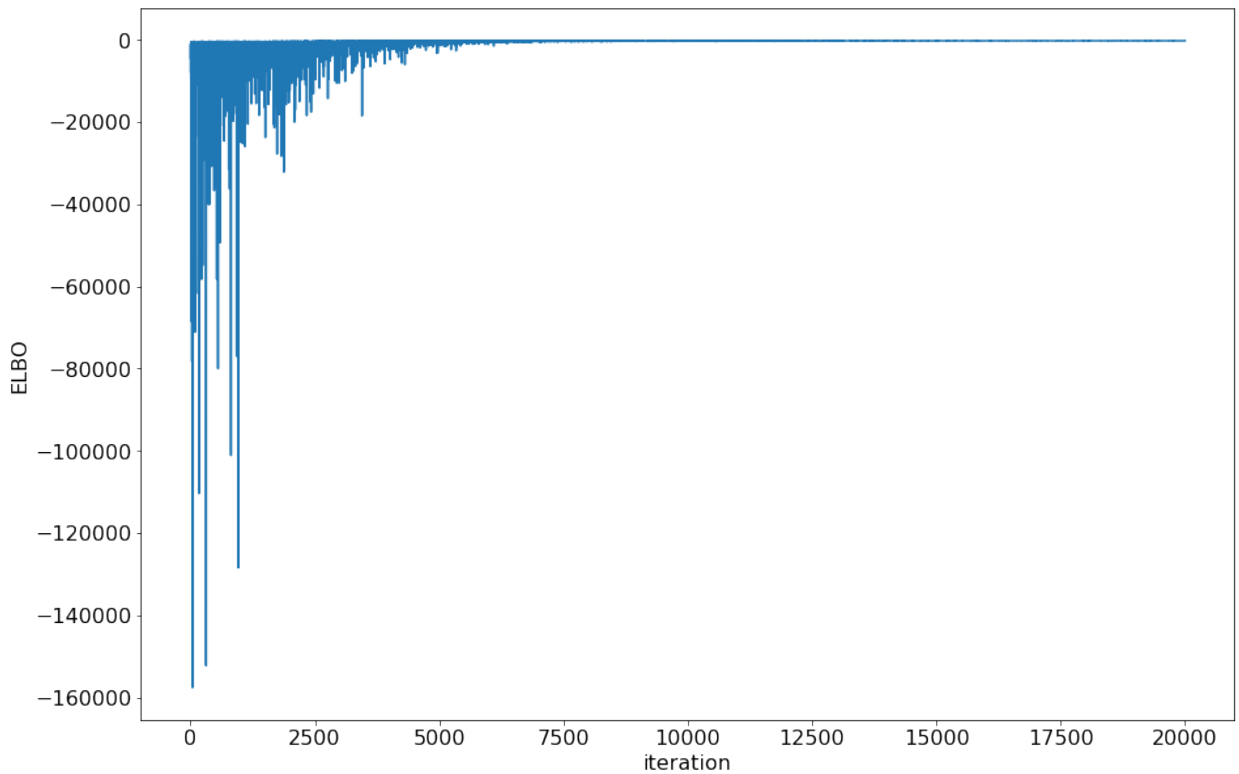
```
ppc = pm.sample_ppc(
    trace[1000:],
    model=lin_reg_model,
    samples=2000
)
pred = ppc['y'].mean(axis=0)
r2_score(Y, pred)
0.794441368799727380
```

1.2.3 Fit a model with PyMC3 Models

Now, we can build a Linear Regression model using PyMC3 models.

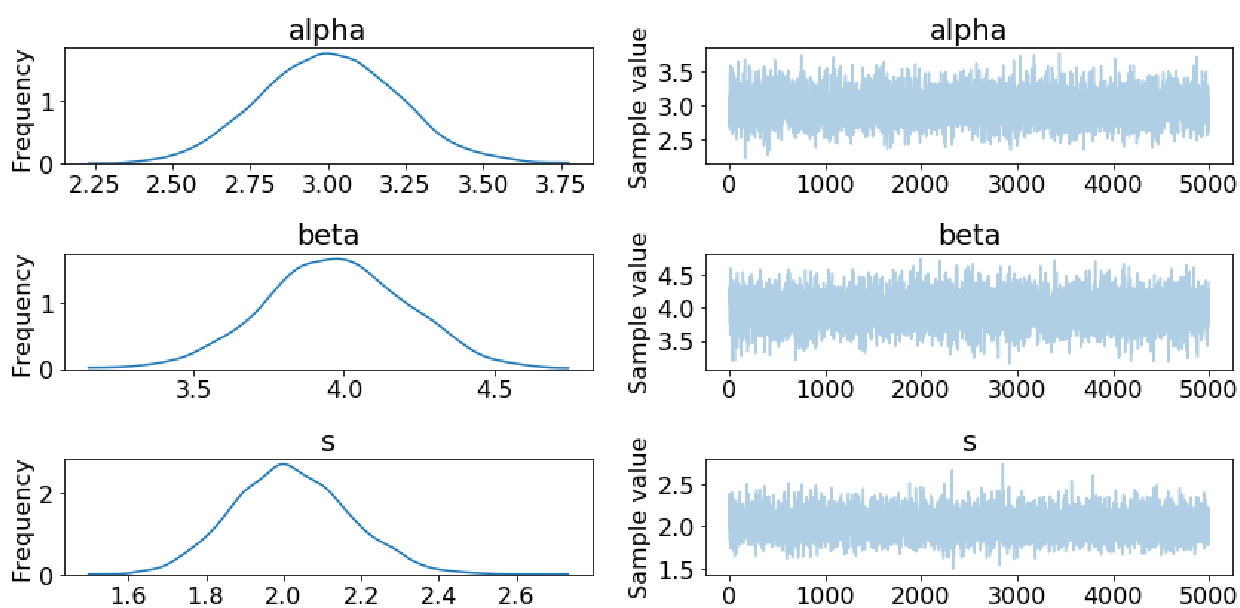
The following is equivalent to Steps 1 and 2 above.

```
LR = LinearRegression()
LR.fit(X, Y, minibatch_size=100)
LR.plot_elbo()
```



The following is equivalent to Step 3 above. Since the trace is saved directly, you can use the same PyMC3 functions (summary and traceplot).

```
traceplot(LR.trace)
```



The following is equivalent to Step 4.

```
Y_predict = LR.predict(X)
LR.score(X, Y)
```

The same type of model can be fit in fewer lines, and the model class follows the scikit-learn API.

If you want a point estimate, you can use the saved summary dataframe:

```
beta = LR.summary['mean']['betas__0_0']
alpha = LR.summary['mean']['alpha__0']
```

1.2.4 Advanced

Saving and Loading your model

```
LR.save('pickle_jar/LR_jar/')
LR2 = LinearRegression()
LR2.load('pickle_jar/LR_jar/')
```

NUTS Inference

The default method of inference for PyMC3 models is minibatch ADVI. This is typically much faster than other methods. However, in some cases, you may want to use the NUTS sampler.

```
LR3 = LinearRegression()
LR3.fit(X, Y, inference_type='nuts', inference_args={'draws': 2000})
```

Now you can use the predict, score methods, etc as above.

Inference Args

If you don't want to use the default arguments for inference, you can pass in inference_args. Check out the PyMC3 documentation for permissible values for the inference_type you are using.

Building your own models

Lastly, if you want to build your own models, you can build them on top of the BayesianModel base class.

1.3 Examples

Check out the notebooks folder.

Currently, the following models have been implemented:

- Linear Regression
- Hierarchical Logistic Regression

1.4 API

1.4.1 pymc3_models

pymc3_models package

Subpackages

pymc3_models.models package

Submodules

pymc3_models.models.HierarchicalLogisticRegression module

class pymc3_models.models.HierarchicalLogisticRegression.**HierarchicalLogisticRegression**

Bases: *pymc3_models.models.BayesianModel*

Custom Hierachical Logistic Regression built using PyMC3.

Methods

create_model()	Creates and returns the PyMC3 model.
fit(X, y, cats[, inference_type, ...])	Train the Hierarchical Logistic Regression model
get_params([deep])	Get parameters for this estimator.
plot_elbo()	Plot the ELBO values after running ADVI minibatch.
predict(X, cats[, num_ppc_samples])	Predicts labels of new data with a trained model
predict_proba(X, cats[, return_std, ...])	Predicts probabilities of new data with a trained Hierarchical Logistic Regression
score(X, y, cats[, num_ppc_samples])	Scores new data with a trained model with sklearn's accuracy_score.
set_params(**params)	Set the parameters of this estimator.

load	
save	

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

fit(*X, y, cats, inference_type='advi', num_advi_sample_draws=10000, minibatch_size=None, inference_args=None*)

Train the Hierarchical Logistic Regression model

Parameters

- **X** (*numpy array*) – shape [num_training_samples, num_pred]

- **y** (*numpy array*) – shape [num_training_samples,]
- **cats** (*numpy array*) – shape [num_training_samples,]
- **inference_type** (*str (defaults to 'advi')*) – specifies which inference method to call Currently, only 'advi' and 'nuts' are supported.
- **num_advi_sample_draws** (*int (defaults to 10000)*) – Number of samples to draw from ADVI approximation after it has been fit; not used if inference_type != 'advi'
- **minibatch_size** (*int (defaults to None)*) – number of samples to include in each minibatch for ADVI If None, minibatch is not run.
- **inference_args** (*dict (defaults to None)*) – arguments to be passed to the inference methods Check the PyMC3 docs for permissable values. If None, default values will be set.

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 saved trace for this model, e.g. given file_prefix = 'path/to/file/' This will attempt to load 'path/to/file/trace.pickle'.
- **load_custom_params** (*bool (defaults to False)*) – flag to indicate whether custom parameters should be loaded

Returns custom_params

Return type Dictionary of custom parameters

predict (*X*, *cats*, *num_ppc_samples=2000*)
Predicts labels of new data with a trained model

Parameters

- **X** (*numpy array*) – shape [num_training_samples, num_pred]
- **cats** (*numpy array*) – shape [num_training_samples,]
- **num_ppc_samples** (*int (defaults to 2000)*) – 'samples' parameter passed to pm.sample_ppc

predict_proba (*X*, *cats*, *return_std=False*, *num_ppc_samples=2000*)
Predicts probabilities of new data with a trained Hierarchical Logistic Regression

Parameters

- **X** (*numpy array*) – shape [num_training_samples, num_pred]
- **cats** (*numpy array*) – shape [num_training_samples,]
- **return_std** (*bool (defaults to False)*) – Flag of whether to return standard deviations with mean probabilities
- **num_ppc_samples** (*int (defaults to 2000)*) – 'samples' parameter passed to pm.sample_ppc

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 trace for this model, e.g. given file_prefix = 'path/to/file/' This will attempt to save to 'path/to/file/trace.pickle'.

- **custom_params** (*dict (defaults to None)*) – Custom parameters to save

score (*X*, *y*, *cats*, *num_ppc_samples=2000*)
Scores new data with a trained model with sklearn's accuracy_score.

Parameters

- **X** (*numpy array*) – shape [num_training_samples, num_pred]
- **y** (*numpy array*) – shape [num_training_samples,]
- **cats** (*numpy array*) – shape [num_training_samples,]
- **num_ppc_samples** (*int (defaults to 2000)*) – 'samples' parameter passed to pm.sample_ppc

pymc3_models.models.LinearRegression module

class pymc3_models.models.LinearRegression.**LinearRegression**
Bases: *pymc3_models.models.BayesianModel*

Linear Regression built using PyMC3.

Methods

create_model()	Creates and returns the PyMC3 model.
fit(X, y[, inference_type, ...])	Train the Linear Regression model
get_params([deep])	Get parameters for this estimator.
plot_elbo()	Plot the ELBO values after running ADVI minibatch.
predict(X[, return_std, num_ppc_samples])	Predicts values of new data with a trained Linear Regression model
score(X, y[, num_ppc_samples])	Scores new data with a trained model with sklearn's r2_score.
set_params(**params)	Set the parameters of this estimator.

load	
save	

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

fit (*X*, *y*, *inference_type='advi'*, *num_advi_sample_draws=10000*, *minibatch_size=None*, *inference_args=None*)
Train the Linear Regression model

Parameters

- **X** (*numpy array*) – shape [num_training_samples, num_pred]
- **y** (*numpy array*) – shape [num_training_samples,]

- **inference_type** (*str (defaults to 'advi')*) – specifies which inference method to call Currently, only 'advi' and 'nuts' are supported.
- **num_advi_sample_draws** (*int (defaults to 10000)*) – Number of samples to draw from ADVI approximation after it has been fit; not used if inference_type != 'advi'
- **minibatch_size** (*int (defaults to None)*) – number of samples to include in each minibatch for ADVI If None, minibatch is not run.
- **inference_args** (*dict (defaults to None)*) – arguments to be passed to the inference methods. Check the PyMC3 docs for permissable values. If None, default values will be set.

`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 saved trace for this model, e.g. given file_prefix = 'path/to/file/' This will attempt to load 'path/to/file/trace.pickle'.
- **load_custom_params** (*bool (defaults to False)*) – flag to indicate whether custom parameters should be loaded

Returns custom_params

Return type Dictionary of custom parameters

`predict` (*X*, *return_std=False*, *num_ppc_samples=2000*)

Predicts values of new data with a trained Linear Regression model

Parameters

- **X** (*numpy array*) – shape [num_training_samples, num_pred]
- **return_std** (*bool (defaults to False)*) – flag of whether to return standard deviations with mean values
- **num_ppc_samples** (*int (defaults to 2000)*) – 'samples' parameter passed to pm.sample_ppc

`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 trace for this model, e.g. given file_prefix = 'path/to/file/' This will attempt to save to 'path/to/file/trace.pickle'.
- **custom_params** (*dict (defaults to None)*) – Custom parameters to save

`score` (*X*, *y*, *num_ppc_samples=2000*)

Scores new data with a trained model with sklearn's r2_score.

Parameters

- **X** (*numpy array*) – shape [num_training_samples, num_pred]
- **y** (*numpy array*) – shape [num_training_samples,]
- **num_ppc_samples** (*int (defaults to 2000)*) – 'samples' parameter passed to pm.sample_ppc

pymc3_models.models.NaiveBayes module

class pymc3_models.models.NaiveBayes.**GaussianNaiveBayes**

Bases: *pymc3_models.models.BayesianModel*

Naive Bayes 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 pymc3_models as pmo
>>>
>>> model = pmo.GaussianNaiveBayes()
>>> model.fit(X,y)
>>> model.predict_proba(X)
>>> model.predict(X)
```

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

Methods

create_model()	Creates and returns the PyMC3 model.
fit(X, y[, inference_type, ...])	Train the Naive Bayes model.
get_params([deep])	Get parameters for this estimator.
plot_elbo()	Plot the ELBO values after running ADVI minibatch.
predict(X)	Classify new data with a trained Naive Bayes model.
predict_proba(X)	Predicts the probabilities that the data points belong to each category.
score(X, y)	Scores new data with a trained model with sklearn's accuracy_score.
set_params(**params)	Set the parameters of this estimator.

load	
save	

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:

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

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

fit(*X*, *y*, *inference_type='advi'*, *num_advi_sample_draws=10000*, *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** (*str (defaults to 'advi')*) – specifies which inference method to call Currently, only 'advi' and 'nuts' are supported.
- **num_advi_sample_draws** (*int (defaults to 10000)*) – Number of samples to draw from ADVI approximation after it has been fit; not used if inference_type != 'advi'
- **minibatch_size** (*int (defaults to None)*) – number of samples to include in each minibatch for ADVI If None, minibatch is not run.
- **inference_args** (*dict (defaults to None)*) – arguments to be passed to the inference methods Check the PyMC3 docs for permissable values. If None, default values will be set.

Returns

Return type The current instance of the GaussianNaiveBayes class.

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 saved trace for this model, e.g. given file_prefix = 'path/to/file/' This will attempt to load 'path/to/file/trace.pickle'.
- **load_custom_params** (*bool (defaults to False)*) – flag to indicate whether custom parameters should be loaded

Returns **custom_params**

Return type Dictionary of custom parameters

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 the 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

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 trace for this model, e.g. given file_prefix = 'path/to/file/' This will attempt to save to 'path/to/file/trace.pickle'.
- **custom_params** (*dict (defaults to None)*) – Custom parameters to save

score (*X, y*)
Scores new data with a trained model with sklearn's accuracy_score.

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

Returns

Return type A float representing the accuracy score of the predictions.

Module contents

class pymc3_models.models.**BayesianModel**

Bases: sklearn.base.BaseEstimator

Bayesian model base class

Methods

get_params([deep])	Get parameters for this estimator.
load(file_prefix[, load_custom_params])	Loads a saved version of the trace, and custom param files with the given file_prefix.
plot_elbo()	Plot the ELBO values after running ADVI minibatch.
save(file_prefix[, custom_params])	Saves the trace and custom params to files with the given file_prefix.
set_params(**params)	Set the parameters of this estimator.

create_model	
fit	
predict	
score	

create_model()

fit()

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 saved trace for this model, e.g. given file_prefix = 'path/to/file/' This will attempt to load 'path/to/file/trace.pickle'.
- **load_custom_params** (*bool (defaults to False)*) – flag to indicate whether custom parameters should be loaded

Returns **custom_params**

Return type Dictionary of custom parameters

plot_elbo()

Plot the ELBO values after running ADVI minibatch.

predict()

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 trace for this model, e.g. given file_prefix = 'path/to/file/' This will attempt to save to 'path/to/file/trace.pickle'.
- **custom_params** (*dict (defaults to None)*) – Custom parameters to save

score()

Submodules

pymc3_models.exc module

exception `pymc3_models.exc.`**`PyMC3ModelsError`**

Bases: `exceptions.Exception`

Module contents

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[1] Murphy, K. P. (2012). Machine learning: a probabilistic perspective.
[1] Murphy, K. P. (2012). Machine learning: a probabilistic perspective.

Python Module Index

p

Index

B

C

F

G

H

L

P

S