# probflow Documentation

**Brendan Hasz**

**Mar 28, 2021**

# CONTENTS:

CHAPTER

ONE

# USER GUIDE

## 1.1 Bayesian Modeling

This is just a brief high-level intro to Bayesian modeling, including what a "Bayesian model" is, and why and when they're useful. To jump to how to use ProbFlow to work with Bayesian models, skip to *Selecting a Backend and Datatype*.

A statistical model represents how data are generated. There are different kinds of models - such as discriminative models (where the model tries to predict some target $y$ based on features $x$), or generative models (where the model is just trying to determine how some target data is generated, without necessarily any features to inform it) - but the common theme is that a model embodies the process by which data is generated, and some assumptions about how that happens.

For example, a linear regression is a simple discriminative model - we're fitting a line to data points so that given any $x$ value, we can predict the corresponding $y$ value.

Most models have parameters, which are values that define the model's predictions. With a linear regression model, there are two parameters which define the shape of the line: the intercept ($b$) and the slope ($m$).

To fit a normal non-Bayesian model, we would want to find the parameters which allow the model to best predict the data. In the case of the linear regression, we want to find the best slope value and the best intercept value, such that the line best fits the data.

But how do we know that a certain value for any parameter really is the "true" value? In the linear regression example above, the best-fit line has an intercept of 0.8, but surely those points could have been generated from a line with an intercept of 1.0? Or even conceivably 1.5, or 2! Though those intercepts seem increasingly unlikely.

With a probabilistic model, instead of simply looking for the single "best" value for each parameter, we want to know how likely *any* parameter value is. That is, we want to find a probability distribution over possible parameter values.

The Bayesian framework adds another layer to probabilistic modeling. With Bayesian analysis, we also specify a "prior" distribution for each parameter - that is, the probability distribution which we *expect* the parameter to take, before having seen any of the data. Then, there's the probability distribution over parameter values the data (together with the model) suggests the parameter should take - this is the "likelihood" distribution. Bayesian analysis combines the prior and likelihood distributions in a mathematically sound way to create what's called the "posterior" distribution - the probability distribution over a parameter's value to which we should update our beliefs *after* having taken into account the data.

With a Bayesian analysis, when we have a lot of data and that data suggests the true parameter value is different than what we expected, that evidence overwhelms our prior beliefs, and the posterior will more strongly reflect the likelihood than the prior distribution. On the other hand, if there isn't much data or if the data is very noisy, the meager

evidence shouldn't be enough to convince us our priors were wrong, and the posterior will more strongly reflect the prior distribution.

There are a few different methods for fitting Bayesian models. Simple models can be solved analytically, but for more complicated models we have to settle for approximations of the posterior distributions. Markov chain monte carlo (MCMC) is one method which uses sampling to estimate the posterior distribution, and is usually very accurate, but can also be very slow for large models or with large datasets. Variational inference is a different method which uses simple distributions to approximate the posteriors. While this means variational inference is often not as accurate as MCMC (because the simple variational posterior can't always perfectly match the true posterior), it is usually much faster. Even faster is the *method ProbFlow uses* to fit models, called stochastic variational inference via "Bayes by backprop".

From a scientific standpoint, Bayesian modeling and analysis encourages being transparent about your assumptions, and shifts the focus to probabilistic interpretations of analysis results, instead of thinking of results in terms of binary it-is-or-it-isn't "significance".

But from a practical or engineering standpoint, why use a probabilistic Bayesian model instead of just a normal one? Bayesian models get you certain information you can't get from a normal model:

- **Parameter uncertainty**. The posterior distributions give information as to how uncertain we should be about the values of specific parameters.
- **Predictive uncertainty**. The predictions of Bayesian models are also probabilistic. While they can just predict the most likely target value given some features, they are also able to output probability distributions over the expected target value. This can be handy when you need to know how confident your model is in its predictions.
- **The ability to separate uncertainty from different sources**. There are different sources of uncertainty in a model. Two main types are epistemic uncertainty (uncertainty due to not having enough data, and therefore having uncertainty as to the true parameter values) and aleatoric uncertainty (uncertainty due to noise, or at least due to factors we can't or haven't measured). Bayesian models allow you to determine how much uncertainty is due to each of these causes. If epistemic uncertainty dominates, collecting more data will improve your model's performance. But if aleatoric uncertainty dominates, you should consider collecting *different* data (or accepting that you've done your best!).
- **Built-in regularization**. Both the strength of priors, and the random sampling used during stochastic variational inference, provide strong regularization and make it more difficult for Bayesian models to overfit.
- **The ability to inject domain knowledge into models**. Priors can be used to add domain knowledge to Bayesian models by biasing parameter posteriors towards values which experts believe are more valid or likely.
- **Provides a framework for making decisions** using your model's probabilistic predictions via Bayesian decision theory.

However, there are a few important disadvantages to using Bayesian models:

- **Computation time**. Bayesian models nearly always take longer to fit than non-probabilistic, non-Bayesian version of the same model.
- **Prior choices are left up to you, and they affect the result**. The choice of the parameters' priors have to be made by the person building the model, and it's usually best to use somewhat informative priors. Unfortunately, the prior can bias the result! So the result of a Bayesian analysis is never really "unbiased."
- **Harder to debug**. Bayesian models often take more time and effort to get working correctly than, say, gradient boosted decision trees or a simple t-test. You have to decide whether that time is worth it for the advantages using a Bayesian model can have for your application.

This has been a quick and *very* non-formal intro to Bayesian modeling, but for an actual introduction, some things I'd suggest taking a look at:

- Doing Bayesian Data Analysis by John Krusche
- Bayesian Data Analysis by Andrew Gelman & co
- The Stan user guide
- Probabilistic Programming and Bayesian Methods for Hackers by Cameron Davidson-Pilon

## 1.2 Selecting a Backend and Datatype

Before building models with ProbFlow, you'll want to decide which backend to use, and what default datatype to use.

### 1.2.1 Setting the Backend

What I mean by "backend" is the system which performs the automatic differentiation required to fit models with stochastic variational inference. ProbFlow currently supports two backends: TensorFlow and PyTorch. TensorFlow is the default backend, but you can set which backend to use:

```
import probflow as pf

pf.set_backend('pytorch') #or 'tensorflow'
```

You can see which backend is currently being used by:

```
pf.get_backend()
```

ProbFlow will only use operations specific to the backend you've chosen, and you can only use operations from your chosen backend when specifying your models via ProbFlow.

### 1.2.2 Setting the Datatype

You can also set the default datatype ProbFlow uses for creating the variable tensors. This datatype much match the datatype of the data you're fitting. The default datatype is `tf.dtypes.float32` when TensorFlow is the backend, and `torch.float32` when PyTorch is the backend.

You can see which is the current default datatype with:

```
pf.get_datatype()
```

And you can set the default datatype with `pf.set_datatype`. For example, to instead use double precision with the TensorFlow backend:

```
pf.set_datatype(tf.dtypes.float64)
```

---

**Personal opinion warning!**

I'd gently recommend sticking to the default float32 datatype. Variational inference is super noisy as is, so do we *really* need all that extra precision? Single precision is also a lot faster on most GPUs. If your data is of a different type, just cast it with (for numpy arrays and pandas DataFrames) `.astype('float32')`.

---

## 1.3 Distributions

Probability *Distributions* describe the probability of either a *Parameter* or a datapoint taking a given value. In ProbFlow, they're used mainly in three places: as parameters' priors, as parameters' variational distributions, and as observation distributions (the predicted distribution of the data which the model predicts). However, you can also use them as stand-alone objects or for other reasons within your models.

### 1.3.1 Creating a distribution object

To create a distribution, just create an instance of a *Distribution*

```
dist = pf.distributions.Normal(1, 2)
```

All ProbFlow distributions are also included in the main namespace, so you can just do:

```
dist = pf.Normal(1, 2)
```

### 1.3.2 Using a distribution as a prior

See *Setting the Prior*.

### 1.3.3 Using a distribution as a variational posterior

See *Specifying the Variational Posterior*.

### 1.3.4 Using a distribution as an observation distribution

See *Specifying the observation distribution*.

### 1.3.5 Getting the log probability of a value along a distribution

ProbFlow distribution objects can also be used in a stand-alone way, and they return values which are tensors of the backend type (e.g. if your backend is Tensorflow, they will return `tf.Tensor` objects, not numpy arrays).

To get the log probability of some value along a probability distribution, use the `log_prob` method:

```
dist = pf.Normal(3, 2)

x = np.linspace(-10, 10, 100)
log_p = dist.log_prob(x)
plt.plot(x, np.exp(log_p))
```

### 1.3.6 Getting the mean and mode of a distribution

To get the mean of a distribution, use the `mean` method:

```
>>> dist = pf.Gamma(4, 5)
>>> dist.mean()
<tf.Tensor: shape=(), dtype=float32, numpy=0.8>
```

And to get the mode, use the `mode` method:

```
>>> dist.mode()
<tf.Tensor: shape=(), dtype=float32, numpy=0.6>
```

### 1.3.7 Getting samples from a distribution

To draw random samples from a distribution, use the `sample` method:

```
>>> dist = pf.Normal(4, 5)
>>> dist.sample()
<tf.Tensor: shape=(), dtype=float32, numpy=5.124513>
```

You can take multiple samples from the same distribution using the `n` keyword argument:

```
>>> dist.sample(n=5)
<tf.Tensor: shape=(5,), dtype=float32, numpy=array([3.9323747, 2.1640768, 5.909429 ,␣
↪7.7332597, 3.4620957], dtype=float32)>
```

If the shape of the distribution's arguments have >1 dimension, the shape of the samples array will be `(Nsamples, DistributionShape1, ..., DistributionShapeN)`:

```
>>> mean = np.random.randn(3, 4)
>>> std = np.exp(np.random.randn(3, 4))
>>> dist = pf.Normal(mean, std)
>>> dist.sample(n=5).shape
TensorShape([5, 3, 4])
```

### 1.3.8 Rolling your own distribution

ProbFlow includes most common *Distributions* but to create a custom distribution which uses a tensorflow_probability.distributions.Distribution or a torch.distributions.distribution just create a class which inherits from *`BaseDistribution`*, and implements the following methods:

- `__init__`: should store references to the tensor(s) to be used as the distribution's parameters
- `__call__`: should return a backend distribution object (a tensorflow_probability.distributions.Distribution or a torch.distributions.distribution )

For example,

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

class NormalDistribution(pf.BaseDistribution):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, mean, std):
        self.mean = mean
        self.std = std

    def __call__(self):
        return tfd.Normal(self.mean, self.std)
```

Or, to implement a probability distribution completely from scratch, create a class which inherits from *BaseDistribution* and implements the following methods:

- `__init__(*args)`: should store references to the tensor(s) to be used as the distribution's parameters
- `log_prob(x)`: should return the log probability of some data (`x`) along this distribution
- `mean()`: the mean of the distribution
- `mode()`: the mode of the distribution
- `sample(n=1)`: should return `n` sample(s) from the distribution
- (and you do **not** need to implement `__call__` in this case)

For example, to manually implement a Normal distribution:

```
class NormalDistribution(pf.BaseDistribution):

    def __init__(self, mean, std):
        self.mean_tensor = mean
        self.std_tensor = std

    def log_prob(self, x):
        return (
            -0.5*tf.math.square((x-self.mean_tensor)/self.std_tensor)
            - tf.log(self.std*2.506628274631)
        )

    def mean(self):
        return self.mean_tensor

    def mode(self):
        return self.mean_tensor

    def sample(self, n=1):
        return tf.random.normal(shape=n, mean=self.mean_tensor, stddev=self.std_
↪tensor)
```

## 1.4 Parameters

*Parameters* are values which characterize the behavior of a model. For example, the intercept and slope of the best-fit line in a linear regression are the parameters of that linear regression model. When fitting a model, we want to find the values of the parameters which best allow the model to explain the data. However, with Bayesian modeling we want not only to find the single *best* value for each parameter, but a probability distribution which describes how likely any given value of a parameter is to be the best or true value.

Parameters have both priors (probability distributions which describe how likely we think different values for the parameter are *before* taking into consideration the current data), and posteriors (probability distributions which describe

how likely we think different values for the parameter are *after* taking into consideration the current data). The prior is set to a specific distribution before fitting the model. While the *type* of distribution used for the posterior is set before fitting the model, the shape of that distribution (the value of the variables which define the distribution's shape) is optimized while fitting the model. See the *Mathematical Details* section for more info.

### 1.4.1 Creating a Parameter

To create a parameter, create an instance of the *Parameter* class:

```
import probflow as pf
param = pf.Parameter()
```

The parameter can be given a unique name via the `name` keyword argument, which will be shown when plotting information about the parameter:

```
param = pf.Parameter(name='weight')
```

You can also create an array of independent parameters of the same type using the `shape` keyword argument. For example, to create a vector of 3 independent parameters,

```
param = pf.Parameter(shape=3)
```

or a $5 \times 3$ matrix of parameters:

```
param = pf.Parameter(shape=[5, 3])
```

#### Specifying the Variational Posterior

To set what distribution to use as the variational posterior for a parameter, pass a *Distribution* class to the `posterior` keyword argument of the parameter when initializing that parameter. For example, to create a parameter which uses a normal distribution for its variational posterior:

```
param = pf.Parameter(posterior=pf.Normal)
```

The default variational posterior on a parameter is a *Normal* distribution, but this can be changed to any *Distribution* class. For example, to instead create a parameter which uses a Cauchy distribution as the variational posterior:

```
param = pf.Parameter(posterior=pf.Cauchy)
```

#### Setting the Prior

To set the prior on a parameter, pass a *Distribution* object to the `prior` keyword argument of the constructor when initializing that parameter. For example, to create a parameter with a prior of Normal(0, 1):

```
param = pf.Parameter(prior=pf.Normal(0, 1))
```

The default prior on a parameter is a *Normal* distribution with a mean of 0 and a standard deviation of 1. However, we can manually set the prior on any parameter to be any *Distribution* (with any parameters). The only limitation is that the backend must be able to analytically compute the Kullback–Leibler divergence between the prior and the posterior (which is usually possible as long as you use the same type of distribution for both the prior and posterior).

### Transforming Parameters

Sometimes it's useful to transform the samples from a parameter after sampling from the variational posterior distribution. You could do this manually each time, of course, but for convenience ProbFlow provides a `transform` keyword argument. The transform should be a function which uses operations from your backend (either TensorFlow or PyTorch) to transform the sample values.

For example, a parameter which uses a normal distribution as the variational posterior will have posterior samples above and below zero:

```
param = pf.Parameter()
param.posterior_plot()
```

But if we initialize a parameter with a softplus transform, the samples will all be greater than zero (because the samples were piped through the softplus function):

```
import tensorflow as tf
param = pf.Parameter(transform=tf.nn.softplus)
param.posterior_plot()
```

One thing to note is that the transform only applies to samples from the variational posterior distribution, and doesn't transform both the prior and posterior distributions with a Jacobian adjustment. So, make sure the prior and untransformed posterior are in the same space! I.e. if you're using a Normal variational posterior with an exponential `transform`, you should use a Normal prior, not a LogNormal prior.

### Setting the variable initializers

The posterior distributions have one or more variables which determine the shape of the distribution (these are the variables which are optimized over the course of training). You can set how the values of the variables are initialized at the beginning of training. The default is to use Xavier initialization (aka Glorot initialization) for the mean of the default Normal posterior distribution, and a shifted Xavier initializer for the standard deviation variable.

To use a custom initializer, use the `initializer` keyword argument to the parameter constructor. Pass a dictionary where the keys are the variable names and the values are functions which have one argument - the parameter's shape - and return a tensor of initial values. For example, to create a matrix of parameters with Normal priors and posteriors, and initialize the posterior's `loc` (the mean) variable by drawing values from a normal distribution, and the `scale` (the standard deviation) parameter with all ones:

```
def randn_fn(shape):
    return tf.random.normal(shape)

def ones_fn(shape):
    return tf.ones(shape)

init_dict = {'loc': randn_fn, 'scale': ones_fn}
param = pf.Parameter(initializer=init_dict)
```

#### Setting the variable transforms

The raw untransformed variables can be transformed before they are used to construct the variational posterior. This comes in handy when you want the underlying variables which are being optimized to be unconstrained, but require the variables to take certain values to construct the variational posterior.

For example, if we're using a normal distribution as the variational posterior for a parameter, we need the standard deviation parameter to be positive (because the variance can't be negative!). But, we want to optimize the variable in unconstrained space. In this case, we can use a softplus function to transform the unconstrained (raw) variable into a value which is always positive. To define what transforms to use for each unconstrained variable, pass a dictionary to the `var_transform` keyword argument when initializing a parameter, where the keys are strings (the names of the variables) and the values are callables (the transforms to apply, note that these transforms must use only backend operations).

```
transforms = {'loc': None, 'scale': tf.nn.softplus}

param = pf.Parameter(initializer=init_dict,
                     var_transform=transforms)
```

There's no transformation for the `loc` variable because that variable can take any value, and so it doesn't need to be transformed.

The transforms in the example above are the default transforms for a Parameter, which assumes a Gaussian variational posterior.

The `var_transform` keyword argument can be used with more complicated functions, for example see the implementation of *`MultivariateNormalParameter`* which uses `var_transform` to implement the log Cholesky reparameterization, which transforms $N(N+1)/2$ unconstrained variables in to a $N \times N$ covariance matrix.

### 1.4.2 Working with Parameters

After a parameter is created, you can take samples from its variational posterior, as well as examine and plot the posterior and priors.

#### Sampling from a Parameter's variational posterior

Parameters return a sample from their variational posterior when called, as a backend tensor:

```
>>> param = pf.Parameter()
>>> param()
<tf.Tensor: shape=(1,), dtype=float32, numpy=array([1.516305])>
```

This method should be used for sampling from parameters' posteriors inside a model, because it returns a backend tensor. Depending on the context, the sample will either be a random sample from the variational distribution (used during model fitting, drawing epistemic samples, or predictive samples) or the variational posterior's mean (used during prediction, drawing aleatoric samples, and computing the residuals). See *creating a model* for more info.

### Sampling from the variational posterior via slicing

You can also index a parameter to take a slice of a sample from the parameter's variational posterior distribution. For example, if you have a parameter matrix,

```
param = pf.Parameter(shape=[5, 3])
```

Then slicing it will yield samples from the variational posteriors. That is, doing `param[<some_slice>]` is equivalent to doing `param()[<some_slice>]`. For example,

```
>>> assert param[:, :] == param()
True
>>> param[0, :]  # -> shape (3,)
[1.23, 4.56, 7.89]
>>> param[:, 1]  # -> shape (5,)
[1.2, 3.4, 5.6, 7.8, 9.0]
>>> param[:2, :]  # -> shape (2,3)
[[1.2, 3.4, 5.6, 7.8, 9.0],
 [1.2, 3.4, 5.6, 7.8, 9.0]]
```

This will even work with vectors of indexes as slices, for example:

```
ix = tf.constant([0, 2, 2, 0, 1, 4, 3])
param = pf.Parameter(shape=[5, 3])
sample = param[ix, :]

#sample is a tf.Tensor
#sample.shape == (7, 3)
#sample[0, :] == sample[3, :]
#sample[1, :] == sample[2, :]
```

### Examining a Parameter's variational posterior

To get the mean of a parameter as a numpy array (*not* as a backend tensor), use the *posterior_mean* method:

```
param = pf.Parameter()
mean = param.posterior_mean()
```

To get a sample from the variational posterior distribution as a numpy array (as opposed to getting a sample as a backend tensor using `__call__`), use the *posterior_sample* method:

```
sample = param.posterior_sample()
# sample is a scalar ndarray
```

To specify how many samples to get, use the `n` keyword argument:

```
samples = param.posterior_sample(n=1000)
# samples is a ndarray with shape (1000,)
```

To compute confidence intervals on a parameter's posterior, use the *posterior_ci* method:

```
lb, ub = param.posterior_ci()
#lb is lower bound on 95% confidence interval
#ub is upper bound
```

The default is to compute the 95% confidence interval using 10,000 samples from the posterior, but these defaults can be changed with the `ci` and `n` keyword arguments. For example, to compute the 80% confidence interval using 100k samples,

```
lb, ub = param.posterior_ci(ci=0.8, n=100000)
```

To plot the variational posterior distribution, use the *posterior_plot* method:

```
param.posterior_plot()
```

To plot confidence intervals, use the `ci` keyword argument. For example, to plot the 90% confidence intervals:

```
param.posterior_plot(ci=0.9)
```

The default plot style is a kernel-density-estimated distribution, but this can be changed with the `style` keyword. For example, to plot with the histogram style:

```
param.posterior_plot(style='hist', ci=0.9)
```

Or just using plain lines:

```
param.posterior_plot(style='line')
```

### Examining a Parameter's prior

Similarly, you can take samples from a parameter's prior distribution using the *prior_sample* method:

```
samples = param.prior_sample(n=1000)
# samples is a ndarray with shape (1000,)
```

And you can plot the prior distribution using the *prior_plot* method:

```
param.prior_plot()
```

### Bayesian updating

Bayesian updating consists of updating a Parameter's prior to match its posterior distribution, after having observed some data. This is, after all, the main point of Bayesian inference! Because ProbFlow uses variational inference, both these distributions have an analytical form (i.e. they're both known probability distributions with known parameters - not a set of MCMC samples or something), and so we can literally just set the prior distribution's variables to be equal to the current posterior distribution's variables!

To perform a Bayesian update on a Parameter, just use the *Parameter.bayesian_update()* method. For example, suppose we have some parameter which has some prior distribution:

```
initializer={"loc": 2, "scale": -1}

param = pf.Parameter(initializer=initializer)
param.prior_plot()
```

The posterior distribution differs from the prior (in this example, simply because of its initialization, but in practice this will be because we've fit the model to some data and thus the likelihood has pulled the posterior away from the prior):

```
param.posterior_plot()
```

We can perform a Bayesian update by using the *Model.bayesian_update()* method:

```
param.bayesian_update()
```

Now, the prior has been udpated to match the current posterior:

```
param.prior_plot()
```

However in practice, you probably won't need to perform Bayesian updates of parameters individually. Usually, you'll fit a model to some data, then call *Model.bayesian_update()* on that model (which updates all the parameters in the model), then fit the model to more, new, data, etc. See the user guide entry for *performing a Bayesian update on a model*.

### 1.4.3 Specialized Parameters

ProbFlow includes several specialized types of parameters. These are all just parameters with specialized default posterior types, priors, initializers, and transforms, etc, which are suited for that parameter type.

#### Scale Parameter

TL;DR: to make a standard deviation parameter, use the *ScaleParameter* class:

```
std_dev = pf.ScaleParameter()
```

A parameter which comes up often in Bayesian modeling is a "scale" parameter. For example, the standard deviation ($\sigma$) in a linear regression with normally-distributed noise:

$$p(y \mid x) = \mathcal{N}(\beta x + \beta_0,\ \sigma)$$

This $\sigma$ parameter cannot take values below 0, because the standard deviation cannot be negative. So, we can't use the default posterior and prior for a parameter (which is a *Normal* distribution for the posterior and `Normal(0, 1)` for the prior), because this default allows negative values.

In Bayesian modeling, the gamma distribution is often used as a posterior for the precision. The precision is the reciprocal of the variance, and so the inverse gamma distribution can be used as a variational posterior for the variance.

However, many models are parameterized in terms of the standard deviation, which is the square root of the variance. So, to create a standard deviation parameter, we could first construct a variance parameter ($\sigma^2$) which uses an inverse

gamma distribution as its variational posterior:

$$\sigma^2 \sim \text{InverseGamma}(\alpha, \beta)$$

and then transform this into a standard deviation parameter ($\sigma$):

$$\sigma = \sqrt{\sigma^2}$$

This could be accomplished with ProbFlow by *setting the posterior* and *the prior* to `InverseGamma`, which means we would also have to specify the initializers and variable transformations accordingly, and then *transform the parameter* with a square root:

```
def randn(shape):
    return tf.random.normal(shape)

std_dev = pf.Parameter(posterior=pf.InverseGamma,
                       prior=pf.InverseGamma(5, 5),
                       transform=lambda x: tf.sqrt(x),
                       initializer={'concentration': randn,
                                    'scale': randn},
                       var_transform={'concentration': tf.nn.softplus,
                                      'scale': tf.nn.softplus})
```

Since that's such a pain, ProbFlow provides a `ScaleParameter` class which automatically creates a parameter with the above variational posterior and transforms, etc. This makes it much easier to create a scale parameter:

```
std_dev = pf.ScaleParameter()
```

### Categorical Parameter

Another type of specialized parameter provided by ProbFlow is the `CategoricalParameter`, which uses a categorical distribution as the variational posterior, and a uniform categorical prior.

To specify how many categories the parameter represents, use the `k` keyword argument. For example, to create a categorical parameter which takes one of three classes,

```
>>> cat_param = pf.CategoricalParameter(k=3)
>>> cat_param.posterior_sample(n=10)
array([1, 0, 0, 1, 2, 2, 0, 0, 0, 1], dtype=int32)
```

The `shape` keyword argument can be used as with a normal parameter:

```
>>> cat_param = pf.CategoricalParameter(k=3, shape=5)
>>> cat_param.posterior_sample()
array([1, 1, 1, 0, 2], dtype=int32)
>>> cat_param.posterior_sample(n=10)
array([[0, 2, 2, 2, 2],
       [0, 0, 2, 0, 2],
       [1, 2, 1, 1, 0],
       [1, 2, 1, 0, 0],
       [0, 2, 0, 2, 0],
       [1, 0, 0, 2, 0],
       [1, 1, 0, 0, 2],
       [0, 2, 1, 0, 0],
       [2, 0, 2, 1, 1],
       [2, 0, 0, 1, 2]], dtype=int32)
```

### Dirichlet Parameter

The *DirichletParameter* is similar to the Categorical parameter, except it uses the Dirichlet distribution as the variational posterior, and a uniform Dirichlet prior. This means that samples from this parameter return categorical probability distributions, *not* samples from a categorical distribution. The `k` keyword argument controls how many categories the parameter represents:

```
>>> param = pf.DirichletParameter(k=3)
>>> param.posterior_sample()
array([0.46866438, 0.10728444, 0.4240512 ], dtype=float32)
>>> param.posterior_sample(n=2)
array([[0.7168123 , 0.04725884, 0.2359289 ],
       [0.17213224, 0.3357264 , 0.4921414 ]], dtype=float32)
```

And the `shape` keyword argument can be used to set the size of the parameter array as usual:

```
>>> param = pf.DirichletParameter(k=2, shape=3)
>>> param.posterior_sample()
array([[0.59154356, 0.40845647],
       [0.31525746, 0.68474257],
       [0.20236614, 0.7976338 ]], dtype=float32)
```

Note that samples from this parameter have size `(shape[0], ... shape[n], k)`, not `(shape[0], ... shape[n])`.

### Bounded Parameter

The *BoundedParameter* can be used to represent a parameter which has both an upper and a lower bound, and uses a normal posterior and prior, with a logit transform to bound the parameter's values on both ends.

By default, the parameter is bounded between 0 and 1:

```
param = pf.BoundedParameter()
param.posterior_plot(style='hist', ci=0.9)
```

But the upper and lower bounds can be set using the `max` and `min` keyword arguments:

```
param = pf.BoundedParameter(min=5, max=10)
param.posterior_plot(style='hist', ci=0.9)
```

### Positive Parameter

The *PositiveParameter* can be used to represent a parameter which must be positive, but has no upper bound. It uses a normal variational posterior and prior, with a softplus transformation.

```
param = pf.PositiveParameter()
param.posterior_plot(style='hist', ci=0.9)
```

### Deterministic Parameter

The *DeterministicParameter* can be used to represent a non-probabilistic parameter. That is, it has no variational posterior distribution because its "variational posterior" is a single point value.

Every random sample from a deterministic parameter has the same value (though the parameters are still trainable):

```
>>> param = pf.DeterministicParameter()
>>> param.posterior_sample(n=10)
array([[2.3372688],
       [2.3372688],
       [2.3372688],
       [2.3372688],
       [2.3372688]], dtype=float32)
```

### Multivariate Normal Parameter

The *MultivariateNormalParameter* uses a multivariate normal distribution (with full covariance) as the variational posterior, with a multivariate normal prior. This comes in handy when you want to model a potential correlation between parameter(s).

The `d` keyword argument sets the dimensionality of the multivariate normal variational posterior used:

```
param = pf.MultivariateNormalParameter(d=3)
param.posterior_plot()
```

Samples from this parameter's variational posterior are in `d` dimensions:

```
>>> param.posterior_sample().shape
(3,)
```

And this parameter models the full covariance structure of the multivariate distribution:

```
samples = param.posterior_sample(n=10000)

import seaborn as sb
sns.kdeplot(samples[:, 0], samples[:, 1])
```

### Centered Parameter

The *CenteredParameter* gets a vector or matrix of parameters which are constrained to have a mean of 0. This can be useful for making a model identifiable (e.g. when using as the weights of a multi-logit regression), or to induce hard centering on hierarchical parameters.

To create a length-5 parameter vector which always returns samples that have a mean of 0:

```
>>> param = pf.CenteredParameter([5, 1])
>>> np.mean(param.posterior_mean())
0.0
>>> np.mean(param.posterior_sample())
0.0
>>> np.mean(param.posterior_sample(3), axis=1)
[0.0, 0.0, 0.0]
```

To create a parameter matrix where the means of each column are always 0 (but the mean of all elements in the matrix is not necessarily 0), set the `center_by` keyword argument to `'column'`:

```
>>> param = pf.CenteredParameter([5, 3], center_by="column")
>>> np.mean(param.posterior_sample(), axis=0)
[0.0, 0.0, 0.0]
```

To create a parameter matrix where the means of each row are always 0 (but the mean of all elements in the matrix is not necessarily 0), set the `center_by` keyword argument to `'row'`:

```
>>> param = pf.CenteredParameter([5, 3], center_by="row")
>>> np.mean(param.posterior_sample(), axis=1)
[0.0, 0.0, 0.0, 0.0, 0.0]
```

## 1.5 Modules

TODO: what a module is

TODO: creating your own module

TODO: why it's useful to express things as modules (so you can hierarchically compose computations, give example of DenseLayer, then DenseNetwork, then MultiheadedNetwork)

TODO: included modules (Dense, Sequential, etc)

## 1.6 Models

TODO: what a model is (takes a Tensor (or nothing!) as input, and returns a probability distribution(s))

TODO: creating your own model (via a class which inherits pf.Model, implement __init__ and __call__, and sometimes it may be required to implement log_likelihood)

TODO: types of models and their ABCs (continuous, discrete, categorical)

### 1.6.1 Specifying the observation distribution

The return value of `Model.__call__` should be a *Distribution* object, which corresponds to the observation distribution. The model itself predicts the shape of this observation distribution. For example, for a linear regression the weights and the bias predict the mean of the observation distribution, and the standard deviation parameter predicts the standard deviation:

```
class LinearRegression(pf.ContinuousModel):

    def __init__(self):
        self.w = pf.Parameter()
        self.b = pf.Parameter()
        self.s = pf.ScaleParameter()

    def __call__(self, x):
        mean = x * self.w() + self.b()
        std = self.s()
        return pf.Normal(mean, std)  # returns predicted observation distribution
```

### 1.6.2 Bayesian updating

Bayesian updating consists of updating a model's parameters' priors to match their posterior distributions, after having observed some data. This is, after all, the main point of Bayesian inference! Because ProbFlow uses variational inference, both parameters' posteriors and priors have an analytical form (i.e. they're both known probability distributions with known parameters - not a set of MCMC samples or something), and so we can literally just set parameters' prior distribution variables to be equal to the current posterior distribution variables!

To perform a Bayesian update of all parameters in a model, use the *`Model.bayesian_update()`* method.

```
model = # your ProbFlow model
model.bayesian_updating()
```

This can be used for incremental model updates when we get more data, but don't want to have to retrain the model from scratch on all the historical data. For example,

```
# x, y = training data
model.fit(x, y)

# Perform the Bayesian updating!
model.bayesian_updating()

# x_new, y_new = new data
model.fit(x_new, y_new)
model.bayesian_update()

# x_new2, y_new2 = even more new data
model.fit(x_new2, y_new2)
model.bayesian_update()

# etc
```

### 1.6.3 Manually computing the log likelihood

The default loss function uses a log likelihood which is simply the log probability of the observed data according to the observation distribution (the distribution which was returned by the `__call__` method of the model, *see above*).

However, you can override this default by re-defining the *`Model.log_likelihood()`* method if you need more flexibility in how you're computing the log probability.

For example, if you want to limit the log probability of each datapoint to -10 (essentially, perform gradient clipping):

```
import probflow as pf
import tensorflow as tf

class LinearRegressionWithClipping(pf.ContinuousModel):

    def __init__(self):
        self.w = pf.Parameter()
        self.b = pf.Parameter()
        self.s = pf.ScaleParameter()

    def __call__(self, x):
        mean = x * self.w() + self.b()
        std = self.s()
        return pf.Normal(mean, std)
```

(continues on next page)

(continued from previous page)

```
    def log_likelihood(self, x, y):
        log_likelihoods = tf.math.maximum(self(x).log_prob(y), -10)
        return tf.reduce_sum(log_likelihoods)
```

Also see the *Censored Time-to-Event Model* example for an example using a custom `log_likelihood` method to handle censored data.

## 1.7 Fitting a Model

TODO:

- basic example of fitting w/ numpy arrays
- fitting changing the batch_size, epochs, lr, and shuffle
- fitting w/ a custom optimizer and/or optimizer kwargs
- fitting w/ or w/o flipout
- passing a pandas dataframe
- passing a DataGenerator to fit

### 1.7.1 Using multiple MC samples per batch

By default, ProbFlow uses only one Monte Carlo sample from the variational posteriors per batch. However, you can use more by passing the `n_mc` keyword argument to *`Model.fit()`*. For example, to use 10 MC samples during training:

```
model = pf.LinearRegression(x.shape[1])

model.fit(x, y, n_mc=10)
```

Using more MC samples will cause the fit to take longer, but the parameter optimization will be much more stable because the variance of the gradients will be less.

Note that *`Dense`* modules, which use the flipout estimator by default, will not use flipout when `n_mc` > 1.

### 1.7.2 Backend graph optimization during fitting

By default, ProbFlow uses tf.function (for TensorFlow) or tracing (for PyTorch) to optimize the gradient computations during training. This generally makes training faster.

```
N = 1024
D = 7
randn = lambda *a: np.random.randn(*a).astype('float32')
x = randn(N, D)
w = randn(D, 1)
y = x@w + 0.1*randn(N, 1)

model = pf.LinearRegression(D)
```

(continues on next page)

(continued from previous page)

```
model.fit(x, y)
# takes around 5s
```

But to disable autograph/tracing and use only eager execution during model fitting, just pass the `eager=True` kwarg to `fit`. This takes longer but can be more flexible in certiain situations that autograph/tracing can't handle.

```
model.fit(x, y, eager=True)
# takes around 28s
```

**Warning:** When inputs are `DataFrames` or `Series` it is not possible to use tracing or `tf.function`, so ProbFlow falls back on eager execution by defualt when the input data are `DataFrames` or `Series`

It's much easier to debug models in eager mode, since you can step through your own code using pdb, instead of trying to step through the tensorflow or pytorch compilation functions. So, if you're getting an error when fitting your model and want to debug the problem, try using `eager=True` when calling `fit`.

However, eager mode is used for all other ProbFlow functionality (e.g. *`Model.predict()`*, *`Model.predictive_sample()`*, *`Model.metric()`*, *`Model.posterior_sample()`*, etc). If you want an optimized version of one of ProbFlow's inference-time methods, for TensorFlow you can wrap it in a `tf.function`:

```
#model.fit(...)

@tf.function
def fast_predict(X):
    return model.predict(X)

fast_predict(x_test)
```

Or for PyTorch, use `torch.jit.trace`:

```
#model.fit(...)

def predict_fn(X):
    return model.predict(X)

fast_predict = torch.jit.trace(predict_fn, (example_x))

fast_predict(x_test)
```

## 1.8 Callbacks

Callbacks can be used to perform actions during the training process. Usually this involves running a function at the beginning or end of each epoch, or at the start or end of training. There are several different callbacks that come built-in to probflow, but you can pretty easily *create your own as well*!

### 1.8.1 Monitoring a metric

It's useful to be able to monitor the performance of your model on validation data over the course of training. This lets you know how quickly the model is learning, and whether the model has actually converged on a good solution (i.e. the metric has stopped improving).

To monitor a metric over the course of training with ProbFlow, use the *MonitorMetric* callback. This takes the name of a metric (see *Model.metric()* for a list of available metrics), and validation data.

For example, to record the mean absolute error of a model over the course of training:

```
#x_val and y_val are numpy arrays w/ validation data
import probflow as pf

monitor_mae = pf.MonitorMetric('mae', x_val, y_val)

model.fit(x_train, y_train, callbacks=[monitor_mae])
```

Then, after training has finished, you can view the value of the metric as a function of epoch:

```
monitor_mae.plot()
```

Or as a function of walltime:

```
monitor_mae.plot(x="time")
```

Additional keyword arguments to *MonitorMetric.plot()* are passed to `matplotlib.pyplot.plt`, so you can also specify labels, line styles, etc.

### 1.8.2 Monitoring the loss

Similarly, you can monitor the loss (the negative evidence lower bound, or ELBO loss) over the course of training. To monitor the ELBO loss, use the *MonitorELBO* callback. Note that this is capturing the ELBO loss on the *training* data, while *MonitorMetric* captured the metric of the model on validation data. However, *MonitorELBO* adds no additional time to training, while *MonitorMetric* does (because it has to evaluate the model on the validation data each epoch).

To record the ELBO loss for each epoch of training data over the course of training, we can create a *MonitorELBO* callback:

```
monitor_elbo = pf.MonitorELBO()

model.fit(x_train, y_train, callbacks=[monitor_elbo])
```

After training has finished, you can view the average value of the ELBO loss as a function of epoch:

```
monitor_elbo.plot()
```

Or as a function of walltime:

```
monitor_elbo.plot(x="time")
```

### 1.8.3 Ending training when a metric stops improving

Often, we may want to end training early when some metric (say, MSE on the validation data) stops improving. This is both to be more efficient - why keep training if it's already as good as it's gonna get? - but also to prevent overfitting (though TBH overfitting by running training for too long is less of a problem for Bayesian neural nets than it is for normal ones).

The *EarlyStopping* method can be used to stop training early when some metric has stopped improving. You can either use it in combination with *MonitorMetric* (*see above*), *MonitorELBO* (also *see above*), or with any arbitrary function you define. You can also perform early stopping with some patience - i.e., don't immediately stop training if your metric is worse than it was on the previous epoch, but only stop training when it has failed to improve after $n$ epochs.

#### Using a metric

To stop training when some accuracy metric stops improving, we can create a *EarlyStopping* callback which monitors the current value of that metric via a *MonitorMetric* callback. For example, to end training when the mean absolute error (MAE) on some validation data stops improving:

```
import probflow as pf

model = ...  # your ProbFlow model
x_train, y_train= ...  # your training data
x_val, y_val = ...  # your validation data

monitor_mae = pf.MonitorMetric('mae', x_val, y_val)
early_stopping = pf.EarlyStopping(monitor_mae)

model.fit(x_train, y_train, callbacks=[monitor_mae, early_stopping])
```

#### Using the loss

Instead of a predictive metric, you can stop training when the loss function (the evidence lower bound, "ELBO") stops improving. To do that, create an *EarlyStopping* callback which monitors the current value of the ELBO via a *MonitorELBO* callback:

```
monitor_elbo = MonitorELBO()
early_stopping = EarlyStopping(monitor_elbo)

model.fit(x_train, y_train, callbacks=[monitor_elbo, early_stopping])
```

#### Using any arbitrary function

You can also have *EarlyStopping* depend on any arbitrary function. For example, to manually compute, say, the symmetric mean absolute percentage error (SMAPE) and stop training when it stops improving,

```
# model = your ProbFlow model
# x_val, y_val = your validation data

def smape():
    y_pred = model.predict(x_val)
    return np.mean(
        2 *np.abs(y_pred - y_val) / (np.abs(y_pred) + np.abs(y_val))
```

(continues on next page)

(continued from previous page)

```
    )

early_stopping = EarlyStopping(smape)

model.fit(x_train, y_train, callbacks=[early_stopping])
```

#### Patience

Often it's a good idea to have some patience before stopping early - i.e. allow the metric being monitored to *not* improve for a few epochs. This gives the optimizer some time to get through any points on the loss surface at which it may temporarily get stuck. For example, to only stop training after the metric has not improved for 5 consecutive epochs, use the `patience` keyword argument to *EarlyStopping*:

```
monitor_mae = MonitorMetric('mae', x_val, y_val)
early_stopping = EarlyStopping(monitor_mae, patience=5)

model.fit(x_train, y_train, callbacks=[monitor_mae, early_stopping])
```

### 1.8.4 Ending training after a fixed amount of time

To end training after a set amount of time, use the *TimeOut* callback. This callback simply takes the number of seconds after which to stop.

For example, to stop training after one hour

```
time_out = pf.TimeOut(3600)

model.fit(x_train, y_train, callbacks=[time_out])
```

### 1.8.5 Changing the learning rate over training

Changing the learning rate over the course of training can lead to better fits. It's common to anneal the learning rate (i.e. decrease it over the course of training) or use cyclical learning rates to allow for faster optimization in the early stages of training, but better fine-tuning in the later stages.

To set the learning rate as a function of epoch, use the *LearningRateScheduler* callback. This callback simply requires a function which takes the epoch as input, and returns a learning rate.

For example, to anneal the learning rate from $10^{-3}$ to $10^{-4}$ over the course of 100 epochs, first define your function which computes the learning rate as a function of epoch:

```
def learning_rate_fn(epoch):
    return np.linspace(1e-3, 1e-4, 100)[epoch-1]
```

Then, define a *LearningRateScheduler* callback which takes that function, and use it to schedule the learning rate during training:

```
lr_scheduler = pf.LearningRateScheduler(learning_rate_fn)

model.fit(x_train, y_train, epochs=100, callbacks=[lr_scheduler])
```

You can plot the learning rate as a function of epoch:

```
lr_scheduler.plot()
```

Or, to use a cyclical annealing learning rate:

```
lr_scheduler = pf.LearningRateScheduler(
    lambda e: 0.1 * (1+np.cos(e/7)) * (1-0.01*e)
)

model.fit(x_train, y_train, epochs=100, callbacks=[lr_scheduler])

lr_scheduler.plot()
```

### 1.8.6 Changing the KL weight over training

It can also help to change the weight of the contribution to the loss of the KL divergence between the parameters' variational posteriors and their priors. Sometimes it is easier for the model to fit if the likelihood is allowed to dominate early in training, and then the regularization of the prior is introduced later on by increasing the weight of the KL contribution to the loss. A cyclical annealing schedule is sometimes even better (see Fu et al., 2019).

To change the weight of the KL contribution to the ELBO loss over the course of training, use the *KLWeightScheduler* callback.

For example, to increase the weight from 0 to 1 over the course of the first 50 epochs, and then keep at 1 for the last 50 epochs,

```
kl_scheduler = pf.KLWeightScheduler(lambda e: min(1.0, e/50.0))

model.fit(x_train, y_train, epochs=100, callbacks=[kl_scheduler])

kl_scheduler.plot()
```

Or, to use a cyclical annealing schedule like that used in Fu et al., 2019,

```
kl_scheduler = pf.KLWeightScheduler(lambda e: min(1.0, e%20/10.0))

model.fit(x_train, y_train, epochs=100, callbacks=[kl_scheduler])

kl_scheduler.plot()
```

### 1.8.7 Monitoring the value of parameter(s)

For diagnosing and debugging models, it can be useful to record the value of parameters over the course of training. This allows you to see where they were initialized, and how the optimization updated them over the course of training.

To record the mean of a parameter's variational posterior over the course of training, use the *MonitorParameter* callback.

For example, if we have a simple linear regression model:

```
class MyModel(pf.Model):

    def __init__(self):
        self.w = pf.Parameter(name="weight")
        self.b = pf.Parameter(name="bias")
        self.s = pf.Parameter(name="std")

    def __call__(self, x):
        return pf.Normal(x * self.w() + self.b(), self.s())

model = MyModel()
```

Then we can record the mean of the weight parameter's variational posterior over the course of training:

```
monitor_weight = pf.MonitorParameter("weight")

model.fit(x_train, y_train, callbacks=[monitor_weight])
```

And we can plot the weight over the course of training using the callback's *MonitorParameter.plot()* method:

```
monitor_weight.plot()
```

You can also monitor multiple parameters by passing a list to *MonitorParameter*:

```
monitor_params = pf.MonitorParameter(["weight", "bias"])

model.fit(x_train, y_train, callbacks=[monitor_params])

import matplotlib.pyplot as plt
monitor_params.plot("weight", label="weight")
monitor_params.plot("bias", label="bias")
plt.legend()
plt.ylabel("Parameter mean")
plt.show()
```

### 1.8.8 Rolling your own callback

It's pretty easy to make your own callback too! Just create a class which inherits from *Callback*, and implement any of its methods which you'd like to use:

- `__init__` - setup
- `on_train_start` - gets called once at the beginning of training
- `on_epoch_start` - gets called at the beginning each epoch
- `on_epoch_end` - gets called at the end each epoch
- `on_train_end` - gets called once at the end of training

For example, to implement a callback which records the time it takes to run each epoch,

```
import time

class TimeEpochs(pf.Callback):

    def __init__(self):
        self.epochs = []
        self.times = []
        self.current_epoch = 0

    def on_epoch_start(self):
        self.t0 = time.time()

    def on_epoch_end(self):
        self.current_epoch += 1
        self.epochs += [self.current_epoch]
        self.times += [time.time() - self.t0]

time_epochs = TimeEpochs()
```

Then you can pass that callback object to *Model.fit()* just as with the pre-built callbacks:

```
model.fit(x, y, callbacks=[time_epochs])
```

Or, to create a callback which records your model's mean absolute calibration error at the end of each epoch,

```
# model = your probflow model
import matplotlib.pyplot as plt

class MonitorMACE(pf.Callback):

    def __init__(self, x_val, y_val):
        self.x_val = x_val
        self.y_val = y_val
        self.epochs = []
        self.maces = []
        self.current_epoch = 0

    def on_epoch_end(self):
        self.current_epoch += 1
        self.epochs += [self.current_epoch]
        self.maces += [
            model.calibration_metric("mace", self.x_val, self.y_val)
        ]
```

(continues on next page)

(continued from previous page)

```
    def plot(self):
        plt.plot(self.epochs, self.maces)
        plt.xlabel("Epoch")
        plt.ylabel("Mean absolute calibration error")
```

## 1.9 Data Generators

TODO: use DataGenerators to generate data programmatically, which is handy when the dataset is too large to fit into memory

TODO: e.g. loading in images

## 1.10 Making Predictions with a Model

TODO: how to predict w/ MAP estimates and predictive distributions (predict, predictive_distribution, and plot_predictive_distribution)

### 1.10.1 Any Model

TODO: talk through how to use *Model*'s methods for prediction and sampling:

- predict
- log_prob / prob
- predictive_sample
- aleatoric_sample
- epistemic_sample
- pred_dist_plot

### 1.10.2 Continuous Model

TODO: talk through how to use

- predictive_interval
- aleatoric_interval
- epistemic_interval
- pred_dist_plot
- predictive_prc

## 1.11 Evaluating Model Performance

TODO: how to evaluate + criticize the model w/ . . .

### 1.11.1 Continuous Models

TODO:

- metric (only continuous ones)
- r_squared
- r_squared_plot
- residuals
- residuals_plot
- calibration_curve
- calibration_curve_plot
- calibration_metric
- sharpness
- coefficient_of_variation
- pred_dist_covered
- pred_dist_coverage
- coverage_by

### 1.11.2 Categorical Models

TODO:

- calibration_curve
- calibration_curve_plot
- calibration_metric

## 1.12 Inspecting a Fit Model

TODO: how to visualize posteriors and priors (and model structure?) with print(model), plot_posterior, plot_prior, predictive_interval, etc

## 1.13 Ready-made Models

TODO: the different ready-made models in the applications module and how to use them

## 1.14 Saving and Loading Models

Models and modules can be saved to a file or serialized to JSON-safe strings.

### 1.14.1 Saving a model to disk

The easiest way to save a ProbFlow *Model* (this will also work for a *Module* object) is to use the *save* method:

```
# ...
# model.fit(...)

model.save('my_model.pfm')

# Or, use pf.save, which does the same thing
# pf.save(model, 'my_model.pfm')
```

Then the file can be loaded with *pf.load*

```
model = pf.load('my_model.pfm')
# model is a pf.Model object
```

Note that the file extension doesn't matter - i.e. it doesn't have to be `.pfm`, that's just for "ProbFlow Model" (to give it an air of legitimacy haha). You could just as easily use `.pkl`, or `.dat` or whatever.

---

**Saving and loading only works between identical Python versions**

Currently, you can only load a ProbFlow model or module file using the exact same version of Python as was used to save it. This is because ProbFlow uses cloudpickle to serialize objects. Fancier/better storage might be supported in the future, but don't hold your breath!

---

### 1.14.2 Serializing a model to a JSON-safe string

Models and modules can also be serialized into JSON-safe strings. To do this, use the *dumps* method of *Model* and *Module*

```
model_str = model.dumps()

# Or, use pf.dumps, which does the same thing
# model_str = pf.dumps(model)
```

These are UTF-8 encoded strings, so they're JSON-safe. That is, you can do this:

```
import json
json.dumps({"model": model.dumps()})
```

Then the model can be loaded back from the string with *pf.loads*

```
model = pf.loads(model_str)
# model is a pf.Model object
```

## 1.15 Mathematical Details

ProbFlow fits Bayesian models to data using stochastic variational inference (Graves, 2011; Hoffman et al., 2013), specifically via "Bayes by Backprop" (Blundell et al., 2015).

Notation:

- Parameters: $\beta$
- Data: $\mathcal{D}$
- Prior: $p(\beta)$
- Likelihood: $p(\mathcal{D}|\beta)$
- Posterior: $p(\beta|\mathcal{D})$

With variational inference we approximate the posterior for each parameter with a "variational posterior distribution" $q$. That variational distribution has some variables $\theta$. For example, if we use a normal distribution as our variational distribution, it has two variables: $\mu$ and $\sigma$. So, $\theta = \{\mu, \sigma\}$ and

$$q(\beta|\theta) = q(\beta|\mu, \sigma) = \mathcal{N}(\beta|\mu, \sigma)$$

To "fit" a Bayesian model with this method, we want to find the values of $\theta$ such that the difference between $q(\beta|\theta)$ (the variational distribution) and $p(\beta|\mathcal{D})$ (the true posterior distribution) is as small as possible.

If we use Kullback-Leibler divergence as our measure of "difference", then we want to find the best values for our variational distribution variables ($\hat{\theta}$) which give the lowest KL divergence between the variational distribution and the true posterior:

$$\hat{\theta} = \arg\min_{\theta} \text{KL}(\ q(\beta|\theta)\ ||\ p(\beta|\mathcal{D})\ )$$

The problem is, we don't know what the true posterior looks like - that's what we're trying to solve! Luckily, this divergence between the variational and true posteriors can be broken down into the sum of three terms:

1. the divergence between the prior and the variational distribution
2. the (negative) expected log likelihood
3. the log model evidence (the probability of the data)

Because:

$$
\begin{aligned}
\text{KL}(\ q(\beta|\theta)\ ||\ p(\beta|\mathcal{D})\ ) &= \int q(\beta|\theta) \log \frac{q(\beta|\theta)}{p(\beta|\mathcal{D})} d\beta \\
&= \int q(\beta|\theta) \log \frac{q(\beta|\theta)\ p(\mathcal{D})}{p(\mathcal{D}|\beta)\ p(\beta)} d\beta \\
&= \int q(\beta|\theta) \left( \log \frac{q(\beta|\theta)}{p(\beta)} - \log p(\mathcal{D}|\beta) + \log p(\mathcal{D}) \right) d\beta \\
&= \int \left( q(\beta|\theta) \log \frac{q(\beta|\theta)}{p(\beta)} - q(\beta|\theta) \log p(\mathcal{D}|\beta) + q(\beta|\theta) \log p(\mathcal{D}) \right) d\beta \\
&= \int q(\beta|\theta) \log \frac{q(\beta|\theta)}{p(\beta)} d\beta - \int q(\beta|\theta) \log p(\mathcal{D}|\beta) d\beta + \int q(\beta|\theta) \log p(\mathcal{D}) d\beta \\
&= \int q(\beta|\theta) \log \frac{q(\beta|\theta)}{p(\beta)} d\beta - \int q(\beta|\theta) \log p(\mathcal{D}|\beta) d\beta + \log p(\mathcal{D}) \\
&= \text{KL}(\ q(\beta|\theta)\ ||\ p(\beta)\ ) - \int q(\beta|\theta) \log p(\mathcal{D}|\beta) d\beta + \log p(\mathcal{D}) \\
&= \text{KL}(\ q(\beta|\theta)\ ||\ p(\beta)\ ) - \mathbb{E}_{q(\beta|\theta)}[\ \log p(\mathcal{D}|\beta)\ ] + \log p(\mathcal{D})
\end{aligned}
$$

The model evidence ($\log p(\mathcal{D})$) is a constant, so in order to find the variational distribution parameters ($\hat{\theta}$) which minimize the divergence between the variational and true posteriors, we can just minimize the right-hand side of the equation, ignoring the model evidence:

$$
\hat{\theta} = \arg\min_{\theta} \text{KL}(\ q(\beta|\theta)\ ||\ p(\beta)\ ) - \mathbb{E}_{q(\beta|\theta)}[\ \log p(\mathcal{D}|\beta)\ ]
$$

These two terms are known as the "variational free energy", or the (negative) "evidence lower bound" (ELBO).

During optimization, we can analytically compute the divergence between the priors and the variational posteriors ( $\text{KL}(\ q(\beta|\theta)\ ||\ p(\beta)\ )$), assuming this is possible given the types of distributions we used for the prior and posterior (e.g. Normal distributions). We can estimate the expected log likelihood ($\mathbb{E}_{q(\beta|\theta)}[\ \log p(\mathcal{D}|\beta)\ ]$) by sampling parameter values from the variational distribution each minibatch, and then computing the average log likelihood for those samples. That is, we estimate it via Monte Carlo.

When creating a loss function to maximize the ELBO, we need to be careful about batching. The above minimization equation assumes all samples are being used, but when using stochastic gradient descent, we have only a subset of the samples at any given time (i.e. the minibatch). So, we need to ensure the contribution of the log likelihood and the KL divergence are scaled similarly. Since we're using a Monte Carlo estimation of the expected log likelihood anyway, with batching we can still just take the mean log likelihood of our samples as the contribution of the log likelihood term. However, the divergence term should be applied once per *pass through the data*, so we need to normalize it by the *total number of datapoints*, not by the number of datapoints in the batch. With TensorFlow, this looks like:

```
# kl_loss = sum of prior-posterior divergences
# log_likelihood = mean log likelihood of samples in batch
# N = number of samples in the entire dataset
elbo_loss = kl_loss/N - log_likelihood
```

### 1.15.1 References

Alex Graves. Practical Variational Inference for Neural Networks. In *Advances in Neural Information Processing Systems*, pages 2348–2356, 2011.

Matthew D. Hoffman, David M. Blei, Chong Wang, and John Paisley. Stochastic Variational Inference. *Journal of Machine Learning Research* 14:13031347, 2013.

Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks. *arXiv preprint*, 2015.

For a quick start, take a look at the *Examples*.

The user guide contains more detailed information about using ProbFlow, including:

- A brief description of *Bayesian modeling*,
- Using *Distributions*, *Parameters*, and *Modules* to create Bayesian *Models*,
- How to *fit those models* to data,
- How to *make predictions* with those models,
- How to *evaluate* the performance of a model,
- How to *inspect* a model's structure and the values of its parameters,
- How to use ProbFlow's *Ready-made Models*,
- How to perform actions mid-training with *Callbacks*,
- How to load data on-the-fly with *Data Generators*,
- How to *save and load models*,
- How to *choose your backend and default datatype*, and
- *Mathematical Details* about how ProbFlow fits Bayesian models.

CHAPTER

TWO

# EXAMPLES

Here are some examples of how to use ProbFlow to build, fit, and diagnose several different types of Bayesian models:

## 2.1 Linear Regression

**TLDR**

```
class LinearRegression(pf.ContinuousModel):

    def __init__(self, d):
        self.w = pf.Parameter([d, 1]) #weights
        self.b = pf.Parameter()       #bias
        self.s = pf.ScaleParameter()  #std dev

    def __call__(self, x):
        return pf.Normal(x @ self.w() + self.b(), self.s())

model = LinearRegression(x.shape[1])
```

or simply

```
model = pf.LinearRegression(x.shape[1])
```

and then

```
model.fit(x, y)
```

### 2.1.1 Simple Linear Regression

A simple linear regression is one of the simplest (discriminative) Bayesian models - it's just fitting a line to some data points! Suppose we have some data for which we have the $x$ values, and we want to predict the $y$ values:

```
# Imports
import probflow as pf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
```

(continues on next page)

(continued from previous page)

```
randn = lambda *x: np.random.randn(*x).astype('float32')

# Generate some data
x = randn(100)
y = 2*x -1 + randn(100)

# Plot it
plt.plot(x, y, '.')
```

With a Bayesian regression, we predict the expected value using two parameters: the slope ($w$) and the intercept ($b$) of the fit line. Then, we model the datapoints as being drawn from a normal distribution (with standard deviation $\sigma$) centered at that expected value.

$$y \sim \text{Normal}(wx + b,\ \sigma)$$

To create this model with ProbFlow, we'll create a class which inherits from the *ContinuousModel* class (because our target variable is continuous). In the `__init__` method, we'll define the parameters of the model. Then in the `__call__` method, we'll use samples from those parameters to generate probabilistic predictions.

TensorFlow

PyTorch

```
class SimpleLinearRegression(pf.ContinuousModel):

    def __init__(self):
        self.w = pf.Parameter(name='Weight')
        self.b = pf.Parameter(name='Bias')
        self.s = pf.ScaleParameter(name='Std')

    def __call__(self, x):
        return pf.Normal(x*self.w()+self.b(), self.s())
```

```
import torch

pf.set_backend('pytorch')

class SimpleLinearRegression(pf.ContinuousModel):

    def __init__(self):
        self.w = pf.Parameter(name='Weight')
        self.b = pf.Parameter(name='Bias')
        self.s = pf.ScaleParameter(name='Std')

    def __call__(self, x):
        x = torch.tensor(x)
        return pf.Normal(x*self.w()+self.b(), self.s())
```

After defining the model class, we just need to create an instance of the model, and then we can fit it to the data using stochastic variational inference!

```
model = SimpleLinearRegression()
model.fit(x, y)
```

Now that we've fit the model, we can use it to make predictions on some test data:

```
# Make predictions
x_test = np.array([-3, 3]).astype('float32')
preds = model.predict(x_test)

# Plot em
plt.plot(x_test, preds, 'r')
plt.plot(x, y, '.')
```

Or view the residuals of the model's predictions (the difference between the actual and predicted values):

```
model.residuals_plot(x, y)
```

Since this is a probabilistic model, we can also look at the posterior distributions of our parameters (that is, the probability distributions over what the model thinks the true values of the parameters are):

```
model.posterior_plot(ci=0.95)
```

We can also get the model's confidence intervals on its predictions:

```
# Compute 95% predictive confidence intervals
x_eval = np.linspace(-3, 3, 100).astype('float32')
lb, ub = model.predictive_interval(x_eval, ci=0.9)

# Plot em
plt.fill_between(x_eval, lb, ub, alpha=0.2)
plt.plot(x, y, '.')
```

Or on just a single datapoint:

```
model.pred_dist_plot(x_eval[:1], ci=0.95)
```

Or draw sample predictions from the model:

```
# Draw sample fits from the model
x_eval = np.array([-3, 3]).astype('float32')
samples = model.predictive_sample(x_eval, n=100)

# Plot em
x_plot = np.broadcast_to(x_eval[:, np.newaxis], samples.T.shape)
plt.plot(x_plot, samples.T, 'r', alpha=0.1)
plt.plot(x, y, '.')
```

And view the Bayesian R-squared distribution!

```
model.r_squared_plot(x, y)
```

### 2.1.2 Multiple Linear Regression

A multiple linear regression is when we have multiple features (independent variables), and a single target (dependent variable). It's just like a simple linear regression, except each feature gets its own weight:

$$y \sim \text{Normal}(x_1 w_1 + x_2 w_2 + \cdots + x_N w_N + b,\ \sigma)$$

or, if we represent the features and weights as vectors:

$$y \sim \text{Normal}(\mathbf{x}^\top \mathbf{w} + b,\ \sigma)$$

Let's generate a dataset with multiple features. Also, we'll store the data in a Pandas DataFrame to see how ProbFlow works with Pandas!

```
# Settings
D = 3   #number of dimensions
N = 100 #number of datapoints

# Generate some data
x = pd.DataFrame(randn(N, D))
weights = randn(D, 1)
y = x @ weights - 1 + 0.2*randn(N, 1)

# Plot em
for i in range(D):
    plt.subplot(1, D, i+1)
    sns.regplot(x[i], y[0])
```

We can create pretty much the same model as before, except we'll create $\mathbf{w}$ as a vector parameter, and because our input is now a pandas DataFrame, we'll call `df.values` to get the underlying numpy array. Also note that below we're using the `@` operator, which is the infix operator for matrix multiplication.

TensorFlow

PyTorch

```
class MultipleLinearRegression(pf.ContinuousModel):

    def __init__(self, dims):
        self.w = pf.Parameter([dims, 1], name='Weights')
        self.b = pf.Parameter(name='Bias')
        self.s = pf.ScaleParameter(name='Std')

    def __call__(self, x):
        return pf.Normal(x.values @ self.w() + self.b(), self.s())
```

```
class MultipleLinearRegression(pf.ContinuousModel):

    def __init__(self, dims):
        self.w = pf.Parameter([dims, 1], name='Weights')
        self.b = pf.Parameter(name='Bias')
        self.s = pf.ScaleParameter(name='Std')

    def __call__(self, x):
        x = torch.tensor(x.values)
        return pf.Normal(x @ self.w() + self.b(), self.s())
```

Again, just instantiate the model and fit it to the data. You can control the learning rate and the number of epochs used to fit the data with the `lr` and `epochs` keyword arguments:

```
model = MultipleLinearRegression(3)
model.fit(x, y, lr=0.1, epochs=300)
```

And again we can view the posterior distributions for our parameters, but this time our weight parameter is a vector with three elements, so each has an independent posterior:

```
model.posterior_plot(ci=0.95)
```

### 2.1.3 Using the Dense Module

The *Dense* module can also be used to build a linear regression:

```
class LinearRegression(pf.ContinuousModel):

    def __init__(self, dims):
        self.layer = pf.Dense(dims, 1)
        self.s = pf.ScaleParameter()

    def __call__(self, x):
        return pf.Normal(self.layer(x.values), self.s())

# Fit it!
model = LinearRegression(3)
model.fit(x, y)
```

### 2.1.4 Using the LinearRegression Model

But the easiest way to do a linear regression with ProbFlow is to use the pre-built *LinearRegression* model:

```
model = pf.LinearRegression(3)
model.fit(x, y)
```

### 2.1.5 Using the MultivariateNormalParameter

So far, our parameters have been completely independent. We might want to model the full joint distribution, to allow for a potential correlation between the parameters. For that, we can use *MultivariateNormalParameter*, which creates a parameter that has a multivariate normal posterior, with full covariance. We'll index the parameter in `__call__`, which automatically takes a slice from a sample of the parameter.

TensorFlow

PyTorch

```
class LinearRegression(pf.ContinuousModel):

    def __init__(self, dims):
        self.betas = pf.MultivariateNormalParameter(dims+2)
```

(continues on next page)

(continued from previous page)

```
    def __call__(self, x):
        w = self.betas[:-2]
        b = self.betas[-2]
        s = tf.nn.softplus(self.betas[-1])
        return pf.Normal(x.values @ w + b, s)
```

```
class LinearRegression(pf.ContinuousModel):

    def __init__(self, dims):
        self.betas = pf.MultivariateNormalParameter(dims+2)

    def __call__(self, x):
        x = torch.tensor(x.values)
        w = self.betas[:-2]
        b = self.betas[-2]
        s = torch.nn.Softplus()(self.betas[-1])
        return pf.Normal(x @ w + b, s)
```

Then we can instantiate the model and fit it:

```
model = LinearRegression(3)
model.fit(x, y, lr=0.1, epochs=300)
```

Now the covariance between parameters has also been modeled:

```
samples = model.betas.posterior_sample(n=10000)
sns.kdeplot(samples[:, 0, 0], samples[:, 1, 0])
```

## 2.2 Logistic Regression

**TLDR**

```
class LogisticRegression(pf.CategoricalModel):

    def __init__(self, d):
        self.w = pf.Parameter([d, 1]) #weights
        self.b = pf.Parameter([1, 1]) #bias

    def __call__(self, x):
        return pf.Bernoulli(x @ self.w() + self.b())

model = LogisticRegression(x.shape[1])
```

or simply

```
model = pf.LogisticRegression(x.shape[1])
```

and then

```
model.fit(x, y)
```

In the last example, both $x$ and $y$ were continuous variables (their values ranged from $-\infty$ to $\infty$). What if our output variable is binary? That is, suppose the output variable can take only one of two values, 0 or 1, and so we need a classification model.

Let's create a dataset which has 3 continuous features, and a target variable with 2 classes:

```
# Imports
import probflow as pf
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
randn = lambda *x: np.random.randn(*x).astype('float32')

# Settings
N = 1000 #number of datapoints
D = 3    #number of features

# Generate data
x = randn(N, D)
w = np.array([[2.], [0.1], [-2.]]).astype('float32')
noise = randn(N, 1)
y = np.round(1./(1.+np.exp(-(x@w+noise))))

# Plot it
for i in range(D):
    plt.subplot(1, D, i+1)
    sns.violinplot(x=y[:, 0], y=x[:, i])
```

### 2.2.1 Building a Logistic Regression Manually

A logistic regression is a model where our output variable is categorical. It's basically the same thing as a linear regression, except we pipe the linearly predicted value through a nonlinear function to get the probability of the output class:

$$p(y = 1) = f(\mathbf{x}^\top \mathbf{w} + b)$$

where $f$ is usually the logistic function. Or, with $> 2$ classes, a softmax.

If our target variable has only 2 possible classes, we can model this using a Bernoulli distribution:

$$y \sim \text{Bernoulli}(f(\mathbf{x}^\top \mathbf{w} + b))$$

To create this model in ProbFlow, we'll create a class which inherits *`CategoricalModel`*, because the target variable is categorical (either 0 or 1). Again, in the `__init__` method we define the parameters of the model, and in the `__call__` method we compute probabilistic predictions given the parameters and the input data:

TensorFlow

PyTorch

```
class LogisticRegression(pf.CategoricalModel):

    def __init__(self, dims):
        self.w = pf.Parameter([dims, 1], name='Weights')
```

(continues on next page)

(continued from previous page)

```
        self.b = pf.Parameter([1, 1], name='Bias')

    def __call__(self, x):
        return pf.Bernoulli(x @ self.w() + self.b())
```

```
import torch

pf.set_backend('pytorch')

class LogisticRegression(pf.CategoricalModel):

    def __init__(self, dims):
        self.w = pf.Parameter([dims, 1], name='Weights')
        self.b = pf.Parameter([1, 1], name='Bias')

    def __call__(self, x):
        x = torch.tensor(x)
        return pf.Bernoulli(x @ self.w() + self.b())
```

Note that by default, the *Bernoulli* distribution treats its inputs as logits (that is, it passes the inputs through a sigmoid function to get the output class probabilities). To force it to treat the inputs as raw probability values, use the `probs` keyword argument to the *Bernoulli* constructor.

Then we can instantiate our model class,

```
model = LogisticRegression(D)
```

And fit it to the data!

```
model.fit(x, y, lr=0.01)
```

Now we can plot the posterior distributions for the weights and the bias, and can see that the model recovered the values we used to generate the data:

```
model.posterior_plot(ci=0.9)
```

### 2.2.2 Using the LogisticRegression module

An even easier way to do a logistic regression with ProbFlow is to use the pre-built *LogisticRegression* model:

```
model = pf.LogisticRegression(D)
model.fit(x, y, lr=0.01)
model.posterior_plot(ci=0.9)
```

### 2.2.3 Multinomial Logistic Regression

The *LogisticRegression* model even handles when $y$ has multiple classes (that is, a Multinomial logistic regression). Let's generate some data with 4 features, where the target has 3 possible classes:

```
# Settings
N = 1000 #number of datapoints
D = 4    #number of features
K = 3    #number of target classes

# Generate data
x = randn(N, D)
w = randn(D, K)
noise = randn(N, 1)
y = np.argmax(x@w+noise, axis=1).astype('float32')

# Plot it
for i in range(D):
    plt.subplot(2, 2, i+1)
    sns.violinplot(x=y, y=x[:, i])
```

The `k` keyword argument to *LogisticRegression* sets the number of classes of the dependent variable.

```
model = pf.LogisticRegression(D, k=K)
model.fit(x, y, lr=0.01, epochs=200)
model.posterior_plot()
```

And we can predict the target class given the features:

```
>>> model.predict(x[:5, :])
array([0, 0, 1, 2, 1], dtype=int32)
```

Or even compute the posterior predictive probability of the target class for test datapoints:

```
x_test = randn(1, D)
model.pred_dist_plot(x_test)
```

## 2.3 Fully-connected Neural Network

**TLDR**

```
class DenseRegression(pf.ContinuousModel):

    def __init__(self, dims):
        self.net = pf.DenseNetwork(dims)
        self.s = pf.ScaleParameter()

    def __call__(self, x):
        return pf.Normal(self.net(x), self.s())
```

(continues on next page)

(continued from previous page)

```
units = [x.shape[1], 256, 128, 64, 32, 1] #units per layer
model = DenseRegression(units)
```

or simply

```
model = pf.DenseRegression(units)
```

and then

```
model.fit(x, y)
```

---

Purely linear regressions aren't able handle complex nonlinear relationships between predictors and target variables - for that kind of data we can use neural networks! Regular neural networks simply provide point estimates, but Bayesian neural networks (BNNs) give us both estimates and uncertainty information. BNNs also have strong regularization "built-in" (which comes from not only the priors, but also from from the sampling performed during stochastic variational inference). This makes it much harder for BNNs to overfit than regular neural networks.

Let's create some nonlinear data to test our neural networks on. Later we'll fit a network to some real-world data, but for now let's just use this toy dataset:

```
# Imports
import probflow as pf
import numpy as np
import matplotlib.pyplot as plt
rand = lambda *x: np.random.rand(*x).astype('float32')
randn = lambda *x: np.random.randn(*x).astype('float32')
zscore = lambda x: (x-np.mean(x, axis=0))/np.std(x, axis=0)

# Create the data
N = 1024
x = 10*rand(N, 1)-5
y = np.sin(x)/(1+x*x) + 0.05*randn(N, 1)

# Normalize
x = zscore(x)
y = zscore(y)

# Plot it
plt.plot(x, y, '.')
```

### 2.3.1 Building a Neural Network Manually

First we'll see how to manually create a Bayesian neural network with ProbFlow from "scratch", to illustrate how to use the *Module* class, and to see why it's so handy to be able to define components from which you can build a larger model. Then later we'll use ProbFlow's pre-built modules which make creating neural networks even easier.

Let's create a module which represents just a single fully-connected layer (aka a "dense" layer). This layer takes a vector $\mathbf{x}$ (of length $N_i$), and outputs a vector of length $N_o$. It multiplies the input by its weights ($\mathbf{W}$, a $N_i \times N_o$ matrix of learnable parameters), and adds a bias ($\mathbf{b}$, a $N_o$-length vector of learnable parameters).

$$\text{DenseLayer}(\mathbf{x}) = \mathbf{x}^\top \mathbf{W} + \mathbf{b}$$

To use ProbFlow to create a module which represents this layer and creates and keeps track of the weight and bias parameters, create a class which inherits *Module*:

TensorFlow

PyTorch

```
import tensorflow as tf

class DenseLayer(pf.Module):

    def __init__(self, d_in, d_out):
        self.w = pf.Parameter([d_in, d_out])
        self.b = pf.Parameter([1, d_out])

    def __call__(self, x):
        return x @ self.w() + self.b()
```

```
import torch

class DenseLayer(pf.Module):

    def __init__(self, d_in, d_out):
        self.w = pf.Parameter([d_in, d_out])
        self.b = pf.Parameter([1, d_out])

    def __call__(self, x):
        x = torch.tensor(x)
        return x @ self.w() + self.b()
```

Side note: we've used `@`, the infix operator for matrix multiplication.

Having defined a single layer, it's much easier to define another *Module* which stacks several of those layers together. This module will represent an entire sub-network of sequential fully connected layers, with ReLU activation functions in between each (but no activation after the final layer). In `__init__`, this new module creates and contains several of the `DenseLayer` modules we defined above.

TensorFlow

PyTorch

```
class DenseNetwork(pf.Module):

    def __init__(self, dims):
        Nl = len(dims)-1 #number of layers
        self.layers = [DenseLayer(dims[i], dims[i+1]) for i in range(Nl)]
        self.activations = (Nl-1)*[tf.nn.relu] + [lambda x: x]

    def __call__(self, x):
        for i in range(len(self.activations)):
            x = self.layers[i](x)
            x = self.activations[i](x)
        return x
```

```
class DenseNetwork(pf.Module):

    def __init__(self, dims):
        Nl = len(dims)-1 #number of layers
        self.layers = [DenseLayer(dims[i], dims[i+1]) for i in range(Nl)]
```

(continues on next page)

(continued from previous page)

```
        self.activations = (Nl-1)*[torch.nn.ReLU()] + [lambda x: x]

    def __call__(self, x):
        x = torch.tensor(x)
        for i in range(len(self.activations)):
            x = self.layers[i](x)
            x = self.activations[i](x)
        return x
```

The first thing to notice here is that *Modules* can contain other *Modules*! This allows you to construct models using hierarchical building blocks, making testing and debugging of your models much easier, and encourages code reuse.

Also note that we've used TensorFlow (or PyTorch) code within the model! ProbFlow lets you mix and match ProbFlow operations and objects with operations from the *backend you've selected*.

Finally, we can create an actual *Model* which uses the network Module we've just created. This model consists of a normal distribution whose mean is predicted by the neural network. Note that while the `__call__` methods of the Modules above returned tensors, the `__call__` method of the Model below returns a *probability distribution*!

```
class DenseRegression(pf.ContinuousModel):

    def __init__(self, dims):
        self.net = DenseNetwork(dims)
        self.s = pf.ScaleParameter([1, 1])

    def __call__(self, x):
        return pf.Normal(self.net(x), self.s())
```

Then we can instantiate the model. We'll create a fully-connected Bayesian neural network with two hidden layers, each having 32 units. The first element of the list passed to the constructor is the number of features (in this case just one: $x$), and the last element is the number of target dimensions (in this case also just one: $y$).

```
model = DenseRegression([1, 32, 32, 1])
```

Then we can fit the network to the data!

```
model.fit(x, y, epochs=1000, lr=0.02)
```

The fit network can make predictions:

```
# Test points to predict
x_test = np.linspace(min(x), max(x), 101).astype('float32').reshape(-1, 1)

# Predict them!
preds = model.predict(x_test)

# Plot it
plt.plot(x, y, '.', label='Data')
plt.plot(x_test, preds, 'r', label='Predictions')
```

And because this is a Bayesian neural network, it also gives us uncertainty estimates. For example, we can compute the 95% posterior predictive distribution intervals:

```
# Compute 95% confidence intervals
lb, ub = model.predictive_interval(x_test, ci=0.95)
```

(continues on next page)

(continued from previous page)

```
# Plot em!
plt.fill_between(x_test[:, 0], lb[:, 0], ub[:, 0],
                 alpha=0.2, label='95% ci')
plt.plot(x, y, '.', label='Data')
```

### 2.3.2 Using the Dense and Sequential Modules

ProbFlow comes with some ready-made modules for creating fully-connected neural networks. The *Dense* module handles creating the weight and bias parameters, and the *Sequential* module takes a list of modules or callables and pipes the output of each into the input of the next.

Using these two modules, we can define the same neural network as above much more easily:

TensorFlow

PyTorch

```
class DenseRegression(pf.Model):

    def __init__(self, d_in):
        self.net = pf.Sequential([
            pf.Dense(d_in, 32),
            tf.nn.relu,
            pf.Dense(32, 32),
            tf.nn.relu,
            pf.Dense(32, 1),
        ])
        self.s = pf.ScaleParameter()

    def __call__(self, x):
        return pf.Normal(self.net(x), self.s())
```

```
class DenseRegression(pf.Model):

    def __init__(self, d_in):
        self.net = pf.Sequential([
            pf.Dense(d_in, 32),
            torch.nn.ReLU(),
            pf.Dense(32, 32),
            torch.nn.ReLU(),
            pf.Dense(32, 1),
        ])
        self.s = pf.ScaleParameter()

    def __call__(self, x):
        x = torch.tensor(x)
        return pf.Normal(self.net(x), self.s())
```

Then we can instantiate and fit the network similarly to before:

```
model = DenseRegression(1)
model.fit(x, y)
```

### 2.3.3 Using the DenseNetwork Module

The *DenseNetwork* module can be used to automatically create sequential models of Dense layers with activations in between (by default, rectified linear activations). Just pass the number of dimensions per dense layer as a list, and *DenseNetwork* will create a fully-connected neural network with the corresponding number of units, rectified linear activation functions in between, and no activation function after the final layer.

For example, to create the same model as above with *DenseNetwork* (but without having to write the component modules yourself):

```
class DenseRegression(pf.Model):

    def __init__(self, dims):
        self.net = pf.DenseNetwork(dims)
        self.s = pf.ScaleParameter()

    def __call__(self, x):
        return pf.Normal(self.net(x), self.s())
```

### 2.3.4 Using the DenseRegression or DenseClassifier applications

The *DenseNetwork* module automatically creates sequential dense layers, but it doesn't include an observation distribuiton. To create the same model as before (a multilayer network which predicts the mean of a normally-distributed observation distribution), use the *DenseRegression* application:

```
model = pf.DenseRegression([1, 32, 32, 1])
model.fit(x, y)
```

To instead use a dense network to perform classification (where the observation distribution is a categorical distribtuion instead of a normal distribution), use the *DenseClassifier* application.

For example, to create a Bayesian neural network (with two hidden layers containing 32 units each) to perform classification between `Nc` categories:

```
# Nf = number of features
# Nc = number of categories in target

model = pf.DenseClassifier([Nf, 32, 32, Nc])
model.fit(x, y)
```

### 2.3.5 Fitting a large network to a large dataset

This is cool and all, but do Bayesian neural networks scale? Bayesian models have a reputation for taking a really long time to train - can we fit a Bayesian neural network to a large dataset in any reasonable amount of time?

Let's use ProbFlow to fit a Bayesian neural network to over 1 million samples from New York City's taxi trip records dataset, which is a publicly available BigQuery dataset.

First let's set some settings, including how many samples to use per batch, the number of epochs to train for, and how much of the data to use for validation.

```
# Batch size
BATCH_SIZE = 1024

# Number of training epochs
EPOCHS = 100

# Proportion of samples to hold out
VAL_SPLIT = 0.2
```

I'll skip the data loading and cleaning code here, though check out the colab if you want to see it! Once we've loaded the data, we have `x_taxi`, the normalized predictors, which includes the pickup and dropoff locations, and the time and date of the pickup:

```
x_taxi.head()
```

| pickup_longitude | pickup_latitude | dropoff_longitude | dropoff_latitude | min_of_day | day_of_week | day_of_year |
|---|---|---|---|---|---|---|
| -0.219310 | 0.745995 | 2.165016 | 4.864790 | -0.768917 | 1.507371 | 0.321964 |
| -0.079724 | -0.021950 | 3.029199 | 0.673444 | -0.241391 | -0.023185 | 1.348868 |
| -0.617643 | -0.531016 | 0.149039 | 0.158556 | -0.100370 | -0.023185 | 0.399466 |
| 3.054433 | 0.806405 | -0.242285 | 0.066294 | 0.087659 | 0.487000 | -0.937446 |
| 3.275032 | 0.662305 | -0.550953 | 0.044937 | 0.226069 | -1.553742 | -1.557464 |
| ... | ... | ... | ... | ... | ... | ... |

And `y_taxi`, the (normalized) trip durations which our model will be trying to predict:

```
>>> y_taxi.head()
0    0.890518
1    1.430678
2    2.711816
3    1.541603
4    1.976256
Name: trip_duration, dtype: float32
```

Let's split this data into training data and data to be used for validation.

```
# Train / validation split
train_N = int(x_taxi.shape[0]*(1.-VAL_SPLIT))
x_train = x_taxi.values[:train_N, :]
y_train = y_taxi.values[:train_N].reshape(-1, 1)
x_val = x_taxi.values[train_N:, :]
y_val = y_taxi.values[train_N:].reshape(-1, 1)
```

Now our training set has over 1.2M rows!

```
>>> print(x_train.shape, y_train.shape)
(1253485, 7) (1253485, 1)
```

To model this data, we'll use a 5-layer fully-connected Bayesian neural network. The first layer will have 256 units, then the second will have 128, and so on. The final layer will have a single unit whose activation corresponds to the network's prediction of the mean of the predicted distribution of the (normalized) trip duration.

We can create a Bayesian neural network with this structure pretty easily using ProbFlow's *DenseRegression* model:

```
model = pf.DenseRegression([7, 256, 128, 64, 32, 1])
```

To see how many parameters the model contains, use the `n_parameters` property of the model. This includes all the weights, the biases, and the standard deviation parameter:

```
>>> model.n_parameters
45314
```

You can also see the total number of underlying variables used for optimization. That is, all the variables used to parameterize the variational posteriors. Since all the parameters in our current model have 2 variables per parameter, this is exactly twice the number of parameters (the weights and biases have normally-distributed variational posteriors which have 2 variables: the mean and the standard deviation, and the noise standard deviation parameter has a Gamma-derived variational posterior and so also has two underlying variables, the concentration and the rate). However, this might not always be the case: certain variational posteriors could have only one variable per parameter (e.g. those with Deterministic or Exponential variational posteriors), or more (e.g. a Student's t-distribution, which has 3).

```
>>> model.n_variables
90628
```

The training of any ProbFlow model can be controlled using `callbacks`, which are similar to Keras callbacks. These can be used to do things such as anneal the learning rate, record the performance of the model over the course of training, record the values of parameters over the course of training, stop training early when some desired performance level is reached, and more!

We'll use three different callbacks during training. First, we'll make a callback to record the evidence lower bound (ELBO), the loss used for training the model, over the course of training. Also, we'll use a callback to record the predictive performance of our model on validation data, using the mean absolute error. Finally, we'll create a callback which anneals the learning rate: it starts at a high learning rate and decreases to ~0 over the course of training.

```
# Record the ELBO
monitor_elbo = pf.MonitorELBO()

# Record the mean absolute error on validation data
monitor_mae = pf.MonitorMetric('mae', x_val, y_val)

# Anneal the learning rate from ~2e-4 to ~0
lr_scheduler = pf.LearningRateScheduler(lambda e: 2e-4-2e-6*e)

# List of callbacks
callbacks = [monitor_elbo, monitor_mae, lr_scheduler]
```

Also, keep in mind that you can *build your own callback* pretty easily if you need to perform custom actions during training!

Then we can fit the model, using those callbacks!

```
model.fit(x_train, y_train,
          epochs=EPOCHS,
          batch_size=BATCH_SIZE,
          callbacks=callbacks)
```

Currently, this takes around 2.5 hrs, but that's using non-optimized eager execution. I'm working on adding support for autograph in TF and tracing in PyTorch, which makes things 3-10 times faster!

After the model has been fit, using it to predict new data is just as easy as before:

```
preds = model.predict(x_val)
```

We can also easily compute various *metrics* of our model's performance on validation data. For example, to compute the mean absolute error:

```
>>> model.metric('mae', x_val, y_val)
0.32324722
```

Also, since we used a callback to record the ELBO over the course of training, we can plot that and see that it has decreased over training and has mostly converged.

```
monitor_elbo.plot()
```

We also used a callback to record the mean absolute error of our model on validation data, and our error decreases over the course of training to reach pretty good performance!

```
monitor_mae.plot()
```

And we also used a callback to anneal the learning rate over the course of training, and we can plot that as well to validate that it actually decreased the learning rate:

```
lr_scheduler.plot()
```

Perhaps most importantly, since this is a Bayesian network, we have access to the uncertainty associated with the model's predictions:

```
# Plot predicted distributions
model.pred_dist_plot(x_val[:6, :], ci=0.95,
                     individually=True, cols=2)

# Show true values
for i in range(6):
    plt.subplot(3, 2, i+1)
    plt.axvline(y_val[i])
    plt.xlim([-4, 4])
    plt.xlabel('')
```

ProbFlow also has *DataGenerator* objects which make training models on large datasets easier (similar to Keras Sequences). Using DataGenerators, you can train models on datasets which are too large to fit into memory by loading data from disk (or from wherever!) batch-by-batch. You can also use them to do on-the-fly data augmentation like random rotation/scaling, random swap, or mixup!

For more complicated examples of using ProbFlow to build neural-network-based models, check out these other examples:

- *Censored Time-to-Event Model*
- *Mixture Density Network*
- *Neural Matrix Factorization*
- *Variational Autoencoder*

## 2.4 Poisson Regression (GLM)

**TLDR**

```
class PoissonRegression(pf.DiscreteModel):

    def __init__(self, d):
        self.w = pf.Parameter([d, 1]) #weights
        self.b = pf.Parameter([1, 1]) #bias

    def __call__(self, x):
        return pf.Poisson(tf.exp(x @ self.w() + self.b()))

model = PoissonRegression(x.shape[1])
model.fit(x, y)
```

In the *linear* and *logistic* regression examples, we created models which dealt with target variables which were continuous and categorical, respectively. But what if our target variable is, say, counts? In this case, the target variable is neither continuous (it can only take non-negative integer values) nor categorical (the number could theoretically be anywhere from 0 to $\infty$, and there is a definite ordering of the values).

With discrete data like this (counts of events during some period of time), we can model the target variable as being generated by a Poisson distribution. Let's generate a dataset which has 3 continuous features, and a Poisson-distributed target variable.

```
# Imports
import probflow as pf
import numpy as np
import matplotlib.pyplot as plt
rand = lambda *x: np.random.rand(*x).astype('float32')
randn = lambda *x: np.random.randn(*x).astype('float32')

# Settings
N = 512 #number of datapoints
D = 3   #number of features

# Data
x = rand(N, D)*4-2
w = np.array([[-1.], [0.], [1.]])
rate = np.exp(x@w + randn(N, D))
y = np.random.poisson(rate).astype('float32')

# Plot it
for i in range(D):
    plt.subplot(1, D, i+1)
    plt.plot(x[:, i], y[:, 0], '.')
```

We can use a Poisson regression to model this kind of data. Like a logistic regression, a Poisson regression is a type of generalized linear model (GLM). In a GLM, we use weight and bias parameters to compute a scalar prediction from the features, pipe that scalar through some function, and use the output as the mean of some observation distribution. With a Poisson regression, the function we use is the exponential function, and the observation distribution is the Poisson distribution.

$$y \sim \text{Poisson}(\exp(\mathbf{x}^\top \mathbf{w} + b))$$

Let's build that model with ProbFlow. Note that our model class below inherits *DiscreteModel*, because the target variable is discrete (it can only take integer values $\geq 0$), and we use the *Poisson* observation distribution.

TensorFlow

PyTorch

```
import tensorflow as tf

class PoissonRegression(pf.DiscreteModel):

    def __init__(self, dims):
        self.w = pf.Parameter([dims, 1], name='Weights')
        self.b = pf.Parameter([1, 1], name='Bias')

    def __call__(self, x):
        return pf.Poisson(tf.exp(x @ self.w() + self.b()))
```

```
import torch

class PoissonRegression(pf.DiscreteModel):

    def __init__(self, dims):
        self.w = pf.Parameter([dims, 1], name='Weights')
        self.b = pf.Parameter([1, 1], name='Bias')

    def __call__(self, x):
        x = torch.tensor(x)
        return pf.Poisson(torch.exp(x @ self.w() + self.b()))
```

Then, we can create an instance of our model and fit it to the data:

```
model = PoissonRegression(D)
model.fit(x, y, epochs=500)
```

We can see that the model recovers the paramers we used to generate the data (except for the bias . . . :/ )

```
model.posterior_plot(ci=0.9)
```

And we can make predictions using either the mean or the mode of the predicted distribution:

```
>>> x_test = randn(5, D)
>>> model.predict(x_test) #mean by default
array([[0.24128665],
       [0.80167174],
       [6.486554  ],
       [1.5575557 ],
       [6.112662  ]], dtype=float32)
>>> model.predict(x_test, method='mode')
array([[0.],
       [0.],
       [6.],
       [1.],
       [6.]], dtype=float32)
```

Also, when we plot the posterior predictive distribution for a test datapoint, we can see that the predictions are discrete (integer values $\geq 0$):

```
model.pred_dist_plot(x_test[2:4, :])
```

## 2.5 Robust Heteroscedastic Regression

**TLDR**

```
class RobustHeteroscedasticRegression(pf.ContinuousModel):

    def __init__(self, dims):
        self.m = pf.Dense(dims)
        self.s = pf.Dense(dims)

    def __call__(self, x):
        return pf.Cauchy(self.m(x), tf.exp(self.s(x)))

model = RobustHeteroscedasticRegression(x.shape[1])
model.fit(x, y)
```

With a *linear regression*, the amount of noise is usually assumed to be constant. But a lot of real data is heteroscedastic - that is, the amount of noise changes as a function of the independent variable(s). Also, most real data isn't perfectly normally distributed, and has some amount of outliers.

Let's generate some data which is heteroscedastic and also contains outliers.

```
# Imports
import probflow as pf
import numpy as np
import matplotlib.pyplot as plt
rand = lambda *x: np.random.rand(*x).astype('float32')
randn = lambda *x: np.random.randn(*x).astype('float32')

# Settings
N = 512 #number of datapoints
D = 1   #number of features

# Heteroscedastic data
x = 2*rand(N, D)-1
y = x*2. - 1. + np.exp(x)*randn(N, 1)

# Add some outliers
y[x<-0.95] += 10

# Plot it
plt.plot(x, y, '.')
```

### 2.5.1 The problem with a regular Linear Regression

Let's try fitting a regular linear regression to this data.

```
model = pf.LinearRegression(D)
model.fit(x, y, lr=0.02, epochs=500)
```

The first problem is that the regular linear regression systematically overestimates the target value for data points which have feature values similar to the outliers (around $x \approx -1$).

```
# Make predictions
x_test = np.linspace(-1, 1, 100).astype('float32').reshape((100, 1))
preds = model.predict(x_test)

# Plot em
plt.plot(x, y, '.')
plt.plot(x_test, preds, 'r')
```

Another problem is that the regular linear regression assumes the variance is constant over different feature values, and so the model can't capture the changing variance which is obviously present in the data. It systematically overestimates the variance for datapoints with low $x$ values, and underestimates the variance for datapoints with high $x$ values:

```
# Compute 95% predictive confidence intervals
lb, ub = model.predictive_interval(x_test, ci=0.9)

# Plot em
plt.fill_between(x_test[:, 0], lb[:, 0], ub[:, 0], alpha=0.2)
plt.plot(x, y, '.')
```

### 2.5.2 Robust Heteroscedastic Regression

To allow the model to capture this change in variance, we'll add to the model weights which use the features to predict the variance. Just like a regular linear regression, we'll use the features to predict the mean:

$$\mu = \mathbf{x}^\top \mathbf{w}_\mu + b_\mu$$

But unlike a regular linear regression, we'll use additional weights to predict the scale of the distribution (after passing it through an exponential function, because the scale must be positive):

$$\gamma = \exp(\mathbf{x}^\top \mathbf{w}_\gamma + b_\gamma)$$

Also, to perform a regression which is robust to outliers, we'll use a Cauchy distribution as the observation distribution, which has heavier tails than a Normal distribution (and so is less affected by outlier values).

$$y \sim \text{Cauchy}(\mu,\ \gamma)$$

Let's create this model with ProbFlow:

TensorFlow

PyTorch

```
import tensorflow as tf

class RobustHeteroscedasticRegression(pf.ContinuousModel):

    def __init__(self, dims):
        self.wm = pf.Parameter([dims, 1], name='Mean weights')
        self.bm = pf.Parameter([1, 1], name='Mean bias')
        self.ws = pf.Parameter([dims, 1], name='Scale weights')
        self.bs = pf.Parameter([1, 1], name='Scale bias')

    def __call__(self, x):
        means = x @ self.wm() + self.bm()
        stds = tf.exp(x @ self.ws() + self.bs())
        return pf.Cauchy(means, stds)
```

```
import torch

class RobustHeteroscedasticRegression(pf.ContinuousModel):

    def __init__(self, dims):
        self.wm = pf.Parameter([dims, 1], name='Mean weights')
        self.bm = pf.Parameter([1, 1], name='Mean bias')
        self.ws = pf.Parameter([dims, 1], name='Scale weights')
        self.bs = pf.Parameter([1, 1], name='Scale bias')

    def __call__(self, x):
        x = torch.tensor(x)
        means = x @ self.wm() + self.bm()
        stds = torch.exp(x @ self.ws() + self.bs())
        return pf.Cauchy(means, stds)
```

And fit it to the data:

```
model = RobustHeteroscedasticRegression(D)
model.fit(x, y, lr=0.02, epochs=500)
```

Taking a look at the parameter posteriors, we can see that the model has recovered the parameter values we used to generate the data:

```
model.posterior_plot(ci=0.9, cols=2)
```

More importantly, the robust heteroscedastic model neither under- or overestimates the target values:

```
# Make predictions
preds = model.predict(x_test)

# Plot em
plt.plot(x, y, '.')
plt.plot(x_test, preds, 'r')
```

And it is able to capture the changing variance as a function of the feature(s):

```
# Compute 95% predictive confidence intervals
lb, ub = model.predictive_interval(x_test, ci=0.9)

# Plot em
plt.fill_between(x_test[:, 0], lb[:, 0], ub[:, 0], alpha=0.2)
plt.plot(x, y, '.')
```

### 2.5.3 Using a Student's t-distribution

Note that with a robust regression, you could also use a Student's t-distribution and estimate the degrees of freedom as well (to model how heavy the tails of the distribution are):

TensorFlow

PyTorch

```
class RobustHeteroscedasticRegression(pf.ContinuousModel):

    def __init__(self, dims):
        self.mw = pf.Parameter([dims, 1], name='Mean weight')
        self.mb = pf.Parameter([1, 1], name='Mean bias')
        self.sw = pf.Parameter([dims, 1], name='Scale weight')
        self.sb = pf.Parameter([1, 1], name='Scale bias')
        self.df = pf.ScaleParameter(name='Degrees of freedom')

    def __call__(self, x):
        means = x @ self.wm() + self.bm()
        stds = tf.exp(x @ self.ws() + self.bs())
        return pf.StudentT(self.df(), means, stds)
```

```
class RobustHeteroscedasticRegression(pf.ContinuousModel):

    def __init__(self, dims):
        self.mw = pf.Parameter([dims, 1], name='Mean weight')
        self.mb = pf.Parameter([1, 1], name='Mean bias')
        self.sw = pf.Parameter([dims, 1], name='Scale weight')
        self.sb = pf.Parameter([1, 1], name='Scale bias')
        self.df = pf.ScaleParameter(name='Degrees of freedom')

    def __call__(self, x):
        x = torch.tensor(x)
        means = x @ self.wm() + self.bm()
        stds = torch.exp(x @ self.ws() + self.bs())
        return pf.StudentT(self.df(), means, stds)
```

## 2.6 Censored Time-to-Event Model

**TLDR**

```
class CensoredSurvivalModel(pf.ContinuousModel):

    def __init__(self, d):
        self.layer = pf.Dense(d)

    def __call__(self, x):
        return tfd.Exponential(tf.exp(self.layer(x)))

    def log_likelihood(self, x, y):
        """If y>=0, that's the time to the observed event.
        If y<0, it has not yet been observed after -y time!"""
        dist = self(x) #predicted distribution
        obs_ll = dist.log_prob(y)[y>=0] #observed likelihoods
        non_ll = dist.log_survival_function(-y)[y<0] #nonobserved
        return tf.reduce_sum(obs_ll) + tf.reduce_sum(non_ll)

model = CensoredSurvivalModel(x.shape[0])
model.fit(x, y)
```

The problem of predicting how long it will be until some event happens again is a pretty common one. For example, time-to-event prediction problems include:

- customer churn modeling (predicting how long it will be before a customer makes their next purchase)
- predictive maintenance (predicting how long it will be until some component of a vehicle or machine breaks)
- rare event prediction (like predicting how long it will be until the next earthquake or flood in some area)
- survival analysis (such as predicting how long a person will live given certian risk factors)
- various healthcare problems (like predicting how long it will be until a patient's next heart attack).

But a big problem with time-to-event prediction is that, depending on your data, much of your data could be censored . Censoring is when some of your data is only partially known. Suppose we're trying to predict the time to a customer's next purchase. We have a history of their purchases: they bought something in January, then again in March. If we're building a predictive model in September, then we have one datapoint from that customer which is *not* censored: in January their time to next purchase was 3 months (in March!). But we also have another datapoint which *is* censored: their time-to-purchase after the 2nd purchase. We know it must be at *least* 6 months (since they haven't bought anything since), but how much greater? 7 months? A year? Maybe they'll never buy again! This incomplete knowledge of a datapoint is called censoring, and that datapoint is "censored".

In this example we'll see how to build these time-to-event models using ProbFlow. First we'll first look at the drawbacks of using a naive model to predict the time to events, then we'll see how to build a time-to-event model which can handle censored data properly, and finally we'll extend that model to use deep Bayesian neural networks for prediction!

We'll be using these models to predict (simulated!) customer purchasing behavior, but basically the same model can be used for many of the other time-to-event prediction applications mentioned above.

First let's simulate some customer purchase data:

```
# Imports
import probflow as pf
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions
randn = lambda *x: np.random.randn(*x).astype('float32')

# Generate time-to-event data
N = 100 #number of customers
D = 2   #number of predictors
x = randn(N, D)
w = np.array([[1.], [-1.]]).astype('float32')
b = randn(1, 1)
y = tfd.Exponential(tf.nn.softplus(x@w + b)).sample().numpy()

# Simulate purchase times
t_stop = 10 #time of data collection
t1 = t_stop*np.random.rand(N)
t2 = t1 + y[:, 0]
cix = t2>t_stop
cid = np.arange(N) #customer id

# Plot em
plt.plot(np.row_stack((t1[~cix], t2[~cix])),
         np.row_stack((cid, cid))[:, ~cix], 'g')
plt.plot(np.row_stack((t1[cix], t_stop+0*t2[cix])),
         np.row_stack((cid, cid))[:, cix], 'r')
plt.plot(t1, cid, '.')
plt.plot(t2[~cix], cid[~cix], 'g.')
plt.axvline(t_stop, color='k', linestyle=':')
```

In the plot above, each row corresponds to a single customer. The blue dots are the customer's first purchase, and the green dots (and line in between) correspond to when that customer made a second purchase. The red lines correspond to customers who haven't made a second purchase before the time we collected the data to train the predictive model.

Our predictive model will be trying to predict the time until a customer's next purchase based on various features about the customer. We've just simulated two random features, but in reality there could be many more, and these features would correspond to things like number of purchases in the past month, number of ads for our site they've viewed recently, their average product review score for products they've bought from us, etc.

We can take a look at the relationship between each of our simulated features and the log time-to-next-purchase.

```
# Plot the data
for d in range(D):
    plt.subplot(1, 2, d+1)
    plt.plot(x[:, d], y, '.')
    plt.plot(x[cix, d], y[cix], 'ro')
    plt.yscale('log')
```

Some features (like $x_1$) make it more likely that the next purchase will take longer to occur (they make the customer more likely to "churn"), while other features (like $x_0$) make it more likely that the next purchase will happen sooner.

However, the points in red haven't actually occurred by the time we collect the data to train our model! The $y$ values correspond to what the actual time-to-events *would* be, but when fitting the models below, that data is censored - that is, we won't have access to the true $y$ values, we'll just know that the purchase has still not yet occurred after some

amount of time.

### 2.6.1 Regular Time-to-Event Model

One option is to just toss the data from customers who haven't made a second purchase, because we don't know when that second purchase would be made! With this method, we'll just fit a model to only the time-to-second-purchase datapoints which were actually observed.

Here's the time-to-next-purchase data for only customers who actually *have* made a second purchase.

```
# Use only datapoints which were observed
y_train = y[~cix, :]
x_train = x[~cix, :]

# Plot em
plt.plot(y_train, '.')
```

To model customers' time to their next purchase, we'll use a simple linear estimator with an Exponential observation distribution. The model predicts the expected time to the next event by multiplying the feature values ($\mathbf{x}$) by some learnable weights ($\mathbf{w}$), and then adding a bias ($b$). Then the likelihood of the actual time to event ($t$) is modeled by an exponential distribution with that predicted rate,

$$t \sim \text{Exponential}(f(\mathbf{x}^\top \mathbf{w} + b))$$

where the $f(\cdot)$ function is just some transform to ensure the predicted rate is positive (we'll use the softplus function, but you could instead use $\exp$ or something).

This model is pretty straightforward to build using ProbFlow:

```
class RegularSurvivalModel(pf.ContinuousModel):

    def __init__(self, d):
        self.w = pf.Parameter([d, 1], name='w')
        self.b = pf.Parameter([1, 1], name='b')

    def __call__(self, x):
        preds = tf.nn.softplus(x @ self.w() + self.b())
        return tfd.Exponential(preds)
```

Or equivalently, but more compactly, we can use a *Dense* module:

```
class RegularSurvivalModel(pf.ContinuousModel):

    def __init__(self, d):
        self.layer = pf.Dense(d)

    def __call__(self, x):
        return tfd.Exponential(tf.nn.softplus(self.layer(x)))
```

Then we can instantiate the model,

```
model1 = RegularSurvivalModel(D)
```

And fit it to the (subset of the) data!

```
model1.fit(x_train, y_train, lr=0.02, epochs=1000)
```

Let's take a look at how the model's estimate of the parameters match up with the actual values we used to generate the data. The distributions below are the model's posteriors, and the vertical lines are the actual values of each parameter which we used to generate the data.

```
# Show model posteriors
model1.posterior_plot(ci=0.9)

# True parameter values
plt.subplot(2, 1, 1)
plt.axvline(w[0], color='C0')
plt.axvline(w[1], color='C1')
plt.subplot(2, 1, 2)
plt.axvline(b, color='C0')
```

While the model gets the parameters in kind of the right direction (i.e., $w_0 > 0$, $w_1 < 0$, and $b < 0$), there is definitely a systematic bias happening which leads to the parameters not being estimated correctly!

Something we might do to try and use the censored data - at least a little bit - is to simply set the $y$ values for events which have not yet occurred to the time we've been waiting. That is, we just set the $y$ values for censored events to the time between the last event and the time at which we want to fit the model. As if we assume that all customers who haven't yet made a 2nd purchase make one the instant we collect the data. That would look like this:

```
# Set nonobserved datapoints to time waiting
x_train = x
y_train = y.copy()
y_train[cix, 0] = t_stop-t1[cix]

# Plot em
plt.plot(y_train, '.')
```

Then we can again try fitting the model to that quasi-censored data.

```
# Instantiate and fit a new model
model = RegularSurvivalModel(D)
model.fit(x_train, y_train, lr=0.02, epochs=1000)

# Show model posteriors
model.posterior_plot(ci=0.9)

# True parameter values
plt.subplot(2, 1, 1)
plt.axvline(w[0], color='C0')
plt.axvline(w[1], color='C1')
plt.subplot(2, 1, 2)
plt.axvline(b, color='C0')
```

That's just as bad!

### 2.6.2 Censored Time-to-Event Model

The bias in the previous model happens because we only used a specific subset of the data (the non-censored data) to train the model! Or, in the second case, because we systematically changed those values (we set the $y$ values of censored datapoints to the time between the first purchase and the time of model fitting).

With the first model, where we omitted censored data completely, we were simply computing the model likelihood based off the likelihood of the observed time-to-events along the predicted exponential distribution.

And for the second attempt above, where we assumed the second event occurred at the time of model-fitting, we used the likelihood of the time between the previous event and the time of model fitting along the predicted distribution.

However, it's possible to build a model which accounts for the fact that some of the data is censored, and uses information about the fact that an event has *not yet occurred* after some specific amount of time. This method takes advantage of the fact that we're using a probability distribution to model the time-to-event values.

The fact that the event has not yet been observed after $t$ time means that we know the true time-to-event is *at least* $t$. Therefore, we can use the predicted probability distribution to compute the probability under the curve after the time of data collection, which gives us the probability of $t$ occuring after that time, according to the predicted probability distribution. This allows our model to learn even from the censored datapoints, because it encourages the parameters to "push" the predicted time-to-event past the censoring time.

This probability density under the curve above some value can be computed using the survival function of a distribution $S(p, t)$ (aka the complementary cumulative distribution function), which is the cumulative probability of some probability distribution $p$ *after* a given value $t$,

$$S(p, t) = \int_t^\infty p(x)\, dx$$

So, if we have our predicted observation distribution as before:

$$\hat{p} = \text{Exponential}(f(\mathbf{x}^\top \mathbf{w} + b))$$

Then, depending on whether the event was observed by the time we collected the data or not, we can compute the likelihood using either the predicted distribution's likelihood or its survival function:

$$p(t \mid \mathbf{x}, \mathbf{w}, b) \;=\; \begin{cases} \hat{p}(t) & \text{if event was observed after } t \text{ time} \\ S(\hat{p}, t) & \text{if event not yet observed after } t \text{ time} \end{cases}$$

To indicate events which have not yet been observed, we'll set their $y$ values to be the negative of the time after which they've not yet been observed. (We could instead just use the positive value of the time after which the event hasn't yet been observed, and the add another column to $x$ which indicates whether an event was actually observed or is censored, but I found just negating the $y$ values to be simpler). Importantly, the new model will treat positive and negative $y$ values fundamentally differently.

```
# Use all the data!
x_train = x
y_train = y.copy()

# Neg. wait time w/ still no observation
y_train[cix, 0] = t1[cix]-t_stop

# Plot it
```

(continues on next page)

(continued from previous page)

```
plt.plot(y_train[cix], 'r.')
plt.plot(y_train[~cix], 'g.')
plt.xlabel('Customer ID')
plt.ylabel('Observed time to next purchase\n'
           'or neg time w/o repeat purchase')
plt.show()
```

Let's use ProbFlow to create a model which actually takes advantage of the censored information intelligently. The generation of the predictions will be exactly the same as before: we use a linear model to estimate the expected time to event, and use an exponential observation distribution. So, the `__init__` and `__call__` methods are the same as before.

However, we'll change how we compute the likelihood. With the previous model, ProbFlow automatically used the probability of y on the predicted exponential distribution as the likelihood. However, now we want to use that only for the observed datapoints, and use the survival probability (the area under the curve after the censoring time) as the likelihood for events which have not yet been observed! To acheive this, we'll override the `log_likelihood` method (which by default just returns `self(x).log_prob(y)`).

```
class CensoredSurvivalModel(pf.ContinuousModel):

    def __init__(self, d):
        self.layer = pf.Dense(d)

    def __call__(self, x):
        return tfd.Exponential(tf.nn.softplus(self.layer(x)))

    def log_likelihood(self, x, y):
        """If y>=0, that's the time to the observed event.
        If y<0, it has not yet been observed after -y time!"""

        # Predicted distributions
        dist = self(x)

        # Likelihoods of observed time-to-events
        obs_ll = dist.log_prob(y)[y>=0]

        # Likelihoods of events not yet observed
        non_ll = dist.log_survival_function(-y)[y<0]

        # Return the sum of log likelihoods
        return tf.reduce_sum(obs_ll) + tf.reduce_sum(non_ll)
```

After instantiating the model,

```
model2 = CensoredSurvivalModel(D)
```

We can fit it to the data - and this time we can use *all* the data!

```
model2.fit(x_train, y_train, lr=0.02, epochs=1000)
```

How do the posteriors look?

```
# Show model posteriors
model2.posterior_plot(ci=0.9)
```

(continues on next page)

(continued from previous page)

```
# True parameter values
plt.subplot(2, 1, 1)
plt.axvline(w[0], color='C0')
plt.axvline(w[1], color='C1')
plt.subplot(2, 1, 2)
plt.axvline(b, color='C0')
```

With this new model that accounts for censoring, the estimates are much closer to the values we actually used to generate the data!

Also, if we compare the predictions of this model with the previous model, the new model's predictions are much more likely to reflect the actual time-to-events.

```
# Make predictions with both models
y_pred1 = model1.predict(x)
y_pred2 = model2.predict(x)

# Compare predictions from two models
plt.plot(y, y_pred1, '.')
plt.plot(y, y_pred2, '.')
plt.plot([0.1, 100], [0.1, 100], 'k:')
plt.xscale('log')
plt.yscale('log')
```

Notice that the regular model (trained on only the data from events which were already observed) *severely* and systematically underestimates the time-to-event for datapoints with a long time-to-event. Like, by an order of magnitude!!

Obviously the severity of this problem depends on how much of your data is censored. For example if you're a credit card company, most of your customers probably use their credit cards on a regular basis, and so the fraction of your time-to-next-purchase events which are censored for a given time cutoff will probably be very small, and using a regular model will be almost as good as using a censored model. However if you're, say, an online retail company trying to predict churn, there will be a much larger contingent of infrequent buyers, which as in the above example would cause a regular model to be highly biased, and so using a censored model would be much more important.

But because the censored model is exactly equivalent to the regular model in the case of no censored events, and better in all other cases, it's probably a good idea to just always use a censored model!

### 2.6.3 Deep Time-to-Event Model

Sometimes though, a simple linear estimator like the one used in the previous model won't be enough to capture complex patterns in your data. We can build a much more expressive time-to-event model by using deep neural networks for estimation! ProbFlow makes adding neural network components to your model much simpler via the *DenseNetwork* module.

One easy way to do this is to just have a neural network predict the expected time to the next event. This is exactly the same as the earlier censored model, but instead of a simple linear estimator, we swap in a neural net.

```
class DeepSurvivalModel(pf.ContinuousModel):

    def __init__(self, dims):
        self.net = pf.DenseNetwork(dims)
```

(continues on next page)

(continued from previous page)

```
    def __call__(self, x):
        return tfd.Exponential(tf.nn.softplus(self.net(x)))

    def log_likelihood(self, x, y):
        """Same as before"""
        dist = self(x)
        obs_ll = dist.log_prob(y)[y>=0]
        non_ll = dist.log_survival_function(-y)[y<0]
        return tf.reduce_sum(obs_ll) + tf.reduce_sum(non_ll)
```

However, the Exponential distribution doesn't separately predict the estimated time-to-event and the uncertainty as to that estimate: the uncertainty is always proportional to the expected time-to-event. This limits the expressiveness of the model quite a bit, because there is no way for the model to express that it is highly confident that the time to event is some specific nonzero value (because the probability of the Exponential distribution is always greatest at $x = 0$ and decreases as $x$ increases).

```
# Plot some Exponential distributions
xx = np.linspace(0, 4, 500)
for p in [1, 2, 3, 4]:
    plt.plot(xx, tfd.Exponential(p).prob(xx))
```

To allow our network to predict both the expected time-to-event and the uncertainty, we can use a Gamma observation distribution (though other options we could use include the Weibull, Log-logistic, or Log-normal distributions). Together, the two parameters of the Gamma distribution (the concentration parameter $\alpha$ and the rate parameter $\beta$) control both the mean of the distribution and how broad the distribution is. By having our neural network predict these two parameters, we allow it to predict not only the expected time-to-event (as with the Exponential model), but also the uncertainty associated with that estimate.

Let's take a look at some Gamma distributions. Unlike the Exponential distribution, the Gamma distribution is able to have peaks at values greater than 0 (though with the right choice of parameters it can also behave just like an Exponential distribution).

```
# Plot some Gamma distributions
xx = np.linspace(0, 10, 500)
for p in [[1, 1], [5, 5], [10, 4], [15, 3]]:
    plt.plot(xx, tfd.Gamma(*p).prob(xx))
```

Unfortunately, using the Gamma distribution instead of the Exponential distribution causes an issue we need to work around (as does using any other positively bounded distribution that isn't the Exponential distribution).

TLDR: TensorFlow throws a hissy fit when there's a NaN or infinite value in the same tensor as the one you want to use to compute the gradient, so we have to do a hacky workaround.

Long version: This is kind of a mundane TensorFlow and TensorFlow Probability specific issue, but I'll elaborate in the astronomically unlikely event that anyone's interested: TensorFlow doesn't compute the gradients correctly when there are NaN/infinities in a vector used to compute those gradients, even when you select only the elements which aren't NaN/inf (using `tf.where`, `tf.boolean_mask`, or logical indexing). See this issue and this other issue for more info. The Exponential distribution didn't give us this issue since it has such a heavy tail, so it's hard to get over/underflow in the log of the survival function - but that's much easier with the Gamma distribution. Also, it just so happened that in a weird quirk of the TensorFlow Probability implementation, their Exponential distribution returns valid values with negative x values, so it wasn't causing NaN values either. To get around this, basically we have to first run the model to compute the log probabilities, figure out which ones are NaN or infinite, then re-run the model using only the datapoints which ended up not having NaN or infinite log probabilities, to ensure that TensorFlow

doesn't hurt its sensitive little eyes when it sees NaN values. That is, we use gradient clipping, but in a fantastically complicated way such that TensorFlow *never even has to see* the large/small gradients. (I'm totally not grumpy about it or anything haha)

That's why the `log_likelihood` function below is so much more complicated - but it's doing exactly the same thing. So, here's the model which uses two deep Bayesian neural networks: one to predict each of the parameters of the Gamma distribution which approximates the time to the next event.

```
class DeepSurvivalModel(pf.ContinuousModel):

    def __init__(self, dims):
        self.a_net = pf.DenseNetwork(dims) #net to predict alpha
        self.b_net = pf.DenseNetwork(dims) #net to predict beta

    def __call__(self, x):
        a = tf.nn.softplus(self.a_net(x))
        b = tf.nn.softplus(self.b_net(x))
        return tfd.Gamma(a, b)

    def log_likelihood(self, x, y):
        """If y>=0, that's the time to the observed event.
        If y<0, it has not yet been observed after -y time!"""

        def grad_safe_indexed_gamma_lp(x, y, lp_fn, ix_fn, min_lp=-100):
            a = tf.nn.softplus(self.a_net(x))
            b = tf.nn.softplus(self.b_net(x))
            ix = ix_fn(y[:, 0]) #indexes to use (observed vs not)
            y_ix = tf.boolean_mask(y, ix, axis=0)
            a_ix = tf.boolean_mask(a, ix, axis=0)
            b_ix = tf.boolean_mask(b, ix, axis=0)
            lp = lp_fn(tfd.Gamma(a_ix, b_ix), y_ix)
            safe = lp[:, 0]>min_lp #indexes with "safe" log probs
            y_safe = tf.boolean_mask(y_ix, safe, axis=0)
            a_safe = tf.boolean_mask(a_ix, safe, axis=0)
            b_safe = tf.boolean_mask(b_ix, safe, axis=0)
            return lp_fn(tfd.Gamma(a_safe, b_safe), y_safe)

        # Likelihoods of observed datapoints
        obs_ll = grad_safe_indexed_gamma_lp(x, y,
            lambda dist, y: dist.log_prob(y),
            lambda y: y>=0)

        # Likelihoods of datapoints which were not observed
        non_ll = grad_safe_indexed_gamma_lp(x, y,
            lambda dist, y: dist.log_survival_function(-y),
            lambda y: y<0)

        # Return the sum of log likelihoods
        return tf.reduce_sum(obs_ll) + tf.reduce_sum(non_ll)
```

Then we can instantiate the neural-network-based time-to-event model:

```
model = DeepSurvivalModel([D, 1])
```

TODO: fit it to some data

**See also**

- WTTE-RNN - Less hacky churn prediction by Egil Martinsson

- Modeling Censored Time-to-Event Data Using Pyro
- Survival analysis

**References**


- David Faraggi and Richard Simon, "A neural network model for survival data" *Statistics in Medicine*, vol. 14(1):73–82, 1995.
- Anny Xiang, Pablo Lapuerta, Alex Ryutov, Jonathan Buckley, and Stanley Azen, "Comparison of the performance of neural network methods and cox regression for censored survival data" *Computational Statistics & Data Analysis, 34(2):243–257, 2000.
- Jared L. Katzman, Uri Shaham, Alexander Cloninger, Jonathan Bates, Tingting Jiang, and Yuval Kluger, "Deepsurv: personalized treatment recommender system using a cox proportional hazards deep neural network" *BMC medical research methodology*, 18(1):24, 2018.
- Changhee Lee, Jinsung Yoon, and Mihaela van der Schaar, "Dynamic-deephit: A deep learning approach for dynamic survival analysis with competing risks based on longitudinal data" *IEEE Transactions on Biomedical Engineering*, 67(1):122-133, 2020.
- Chirag Nagpal, Xinyu Li, and Artur Dubrawski, "Deep Survival Machines: Fully Parametric Survival Regression and Representation Learning for Censored Data with Competing Risks", *arXiv preprint*, 2020.


## 2.7 Multivariate Regression

A multivariate regression is one where the target (dependent) variable has multiple dimensions instead of the usual 1. For example, predicting a location in 3D space, or a location on a 2D map.

Let's generate a dataset which has 7 features, and the target variable is 3-dimensional.

```
# Imports
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
import probflow as pf
np.random.seed(12345)
tf.random.set_seed(12345)
randn = lambda *x: np.random.randn(*x).astype('float32')

# Data with multiple output dimensions
N = 512
Di = 7
Do = 3
x = randn(N, Di)
w = randn(Di, Do)
y = x @ w + 0.3*randn(N, Do)
```

### 2.7.1 Homoscedastic Multivariate Regression

The simplest way to do a multivariate regression would be to predict the mean in each dimension, and assume the variance is constant in each dimension (but allow the variance to be different in each of the output dimensions). Then the observation distribution is a *MultivariateNormal* instead of just a *Normal* distribution:

```
class MultivariateRegression(pf.Model):

    def __init__(self, dims):
        self.net = pf.DenseNetwork(dims)
        self.std = pf.ScaleParameter(dims[-1])

    def __call__(self, x):
        loc = self.net(x)
        cov = tf.linalg.diag(self.std())
        return pf.MultivariateNormal(loc, cov)
```

Then we can create the model and fit it:

```
model = MultivariateRegression([Di, Do])
model.fit(x, y, epochs=1000)
```

Now our model's output/predictions are in multiple dimensions (note that the predictions are in 3 dimensions but we're only plotting the first 2):

```
for i in range(5):
    y_pred = model.predictive_sample(x[i:i+1, :])
    sns.kdeplot(y_pred[:, 0, 0], y_pred[:, 0, 1])
    plt.plot(y[i, 0], y[i, 1], 'r*')
    plt.show()
```

Note that the above is simply doing a multivariate linear regression - to use a basic feed-forward neural network instead, one would just add more elements to the `dims` list (to add hidden layers). E.g. the following would use a neural network with three hidden layers, the first w/ 128 units, the 2nd with 64 units, and the last w/ 32 units:

```
model = MultivariateRegression([Di, 128, 64, 32, Do])
```

### 2.7.2 Heteroscedastic Multivariate Regression

But, the previous model assumes the variance doesn't change as a function of the input variables! To allow for that, we can separately model the mean and the variance as a function of the predictors:

```
class MultivariateRegression(pf.Model):

    def __init__(self, dims):
        self.mean_net = pf.DenseNetwork(dims)
        self.std_net = pf.DenseNetwork(dims)
```

(continues on next page)

(continued from previous page)

```
    def __call__(self, x):
        loc = self.mean_net(x)
        cov = tf.linalg.diag(tf.exp(self.std_net(x)))
        return pf.MultivariateNormal(loc, cov)
```

Which can then be fit with:

```
model = MultivariateRegression([Di, Do])
model.fit(x, y, epochs=1000)
```

In this case to use a neural network, while you could just add elements to `dims` like before, this would just mean you're using two completely separate neural networks to predict the mean and the variances. Maybe a more efficient way to do it would be to use two separate heads of your network. So: there's a neural network, then a single linear layer on top of that neural network predicts the means, and another single linear layer predicts the variance, like this:

```
class MultivariateRegression(pf.Model):

    def __init__(self, dims, head_dims):
        self.net = pf.DenseNetwork(dims)
        self.mean = pf.DenseNetwork(head_dims)
        self.std = pf.DenseNetwork(head_dims)

    def __call__(self, x):
        z = self.net(x)
        loc = self.mean(z)
        cov = tf.linalg.diag(tf.exp(self.std(z)))
        return pf.MultivariateNormal(loc, cov)
```

Which could then be created like this (note that the last element of `dims` and the first element of `head_dims` should match):

```
model = MultivariateRegression([Di, 128, 64, 32], [32, Do])
model.fit(x, y, epochs=1000)
```

Note that, as long as long as you're not modeling the covariance structure of the target (i.e., the predicted covariance matrix only has nonzero elements along the diagonal), you can accomplish basically the same thing using *`LinearRegression`* or *`DenseRegression`* (note that this is using `Do` independent normal distributions, instead of a single multivariate normal distribution):

```
model = pf.LinearRegression(x.shape[1], y.shape[1])
```

or

```
model = pf.DenseRegression([x.shape[1], y.shape[1]])
```

## 2.8 Neural Linear Model

**TLDR**

```
import probflow as pf
import probflow.utils.ops as O

class NeuralLinear(pf.ContinuousModel):

    def __init__(self, dims):
        self.net = pf.DenseNetwork(dims, probabilistic=False)
        self.loc = pf.Dense(dims[-1], 1)
        self.std = pf.Dense(dims[-1], 1)

    def __call__(self, x):
        h = O.relu(self.net(x))
        return pf.Normal(self.loc(h), O.softplus(self.std(h)))

model = NeuralLinear([x.shape[1], 256, 128, 64, 32])
model.fit(x, y)
```

The neural linear model is an efficient way to get posterior samples and uncertainty out of a regular neural network. Basically, you just slap a Bayesian linear regression on top of the last hidden layer of a regular (non-Bayesian, non-probabilistic) neural network. It was first proposed for use with Bayesian optimization (Snoek et al., 2015), but has applications in reinforcement learning (see Riquelme et al., 2018 and Azizzadenesheli and Anandkumar, 2019), active learning, AutoML (Zhou and Precioso, 2019), and just Bayesian regression problems in general (Ober and Rasmussen, 2019).

Bayesian linear regressions have a closed form solution, so the usual approach for training a neural linear model is to first train the neural network to minimize the (for example) mean squared error. Then in a second step, compute the closed-form solution to the Bayesian linear regression regressing the last hidden layer's activations onto the target variable. Here we'll just use variational inference to train the neural network and Bayesian regression together end-to-end.

But first! Some imports:

```
import probflow as pf
import probflow.utils.ops as O
import matplotlib.pyplot as plt
```

### 2.8.1 Load data

We'll be training models to predict taxi fares given the origin, destination, and time of the trip. The data we'll use comes from the New York City Taxi and Limousine Commission dataset, which is a public BigQuery dataset. I'll skip the data loading and feature engineering here (see the colab for all of that!), but at the end of the day we're left with around 2.4 million training samples and little over half a million validation samples, in the form of four numpy arrays:

```
x_train  # training features, size (2399893, 9)
y_train  # training target, size (2399893, 1)
x_val  # validation features, size (600107, 9)
y_val  # validation target, size (600107, 1)
```

### 2.8.2 Fitting the Neural Linear Model

Let's create a neural linear model with ProbFlow. ProbFlow's *Dense*, *DenseNetwork*, and *DenseRegression* classes take a `probabilistic` keyword argument, which when `True` (the default), uses parameters which are probabilistic (i.e., they use Normal distributions as the variational posteriors). But when the `probabilistic` kwarg is set to `False`, then the parameters are totally non-probabilistic (i.e. a Deterministic distribution, aka a Dirac function, is used as the "variational posterior"). So, with `probabilistic = False`, ProbFlow won't model any uncertainty as to those parameters' values (like your run-of-the-mill non-Bayesian neural network would).

So, to create a neural linear model, we can just create a regular non-probabilistic neural network using *DenseNetwork* with `probabilistic = False`, and then perform a Bayesian linear regression on top of the final hidden layer (using the *Dense* class - we'll also predict the noise error to allow for heteroscedasticity).

```
class NeuralLinear(pf.ContinuousModel):

    def __init__(self, dims):
        self.net = pf.DenseNetwork(dims, probabilistic=False)
        self.loc = pf.Dense(dims[-1], 1)  # probabilistic=True by default
        self.std = pf.Dense(dims[-1], 1)  # probabilistic=True by default

    def __call__(self, x):
        h = O.relu(self.net(x))
        return pf.Normal(self.loc(h), O.softplus(self.std(h)))
```

However, one thing that really helps is to define what initialization we want for the head of the network which predicts the standard deviation. You don't *need* to, but a lot of random initializations can lead to bad fits otherwise. Basically we're just initializing the layer which predicts the noise standard deviation such that the standard deviation is small early in training. Otherwise the noise coming from the variational posteriors overwhelms the signal coming from the data, and so the network is unable to learn. Initializing the standard deviation head to output small values allows the network to learn the means early in training, and then later in training the variational posteriors can expand to capture the uncertainty properly.

So, to set the values at which our variables for that layer are initialized,

```
std_kwargs = {
    "weight_kwargs": {
        "initializer": {
            "loc": 0,
            "scale": -2,
        }
    },
    "bias_kwargs": {
        "initializer": {
            "loc": -2,
            "scale": -2,
        }
    }
}
```

And then we can re-define our neural linear model using that initialization (exactly the same as before, except we've added `**std_kwargs`):

```
class NeuralLinear(pf.ContinuousModel):

    def __init__(self, dims):
        self.net = pf.DenseNetwork(dims, probabilistic=False)
        self.loc = pf.Dense(dims[-1], 1)
```

(continues on next page)

(continued from previous page)

```
        self.std = pf.Dense(dims[-1], 1, **std_kwargs)

    def __call__(self, x):
        h = O.relu(self.net(x))
        return pf.Normal(self.loc(h), O.softplus(self.std(h)))
```

Then we can instantiate the model. We'll use a fully-connected sequential architecture, where the first hidden layer has 128 units, the second has 64, and the third has 32.

```
model = NeuralLinear([x.shape[1], 128, 64, 32])
```

We'll also use a *MonitorMetric* callback to monitor the mean absolute error of the model's predictions on the validation data over the course of training. Additionally, we'll anneal the learning rate from 0.0005 to near zero over the course of training using a *LearningRateScheduler* callback.

```
nlm_mae = pf.MonitorMetric('mae', x_val, y_val)
nlm_lrs = pf.LearningRateScheduler(lambda e: 5e-4 + e * 5e-6)
```

Then, we can fit the model!

```
nlm_model.fit(
    x_train,
    y_train,
    batch_size=2048,
    epochs=100,
    callbacks=[nlm_mae, nlm_lrs]
)
```

### 2.8.3 Fitting a fully Bayesian network

For comparison, let's also fit a fully Bayesian model (i.e., one where *all* the parameters of the network are modeled using Normal distributions as their variational posteriors). We'll define it in exactly the same way as the neural linear model, except that we'll use `probabilistic = True` (the default) when initializing the *DenseNetwork*.

```
class FullyBayesianNetwork(pf.ContinuousModel):

    def __init__(self, dims):
        self.net = pf.DenseNetwork(dims)
        self.loc = pf.Dense(dims[-1], 1)
        self.std = pf.Dense(dims[-1], 1, **std_kwargs)

    def __call__(self, x):
        h = O.relu(self.net(x))
        return pf.Normal(self.loc(h), O.softplus(self.std(h)))
```

And we'll initialize it using the exact same architecture:

```
bnn_model = FullyBayesianNetwork([x_train.shape[1], 128, 64, 32])
```

Again we'll monitor the MAE over the course of training, and anneal the learning rate:

```
bnn_mae = pf.MonitorMetric('mae', x_val, y_val)
bnn_lrs = pf.LearningRateScheduler(lambda e: 5e-4 + e * 5e-6)
```

And then we can fit it:

```
bnn_model.fit(
    x_train,
    y_train,
    batch_size=2048,
    epochs=100,
    callbacks=[bnn_mae, bnn_lrs]
)
```

### 2.8.4 Accuracy and time

Let's compare how long it took each network to reach similar levels of accuracy. We can use the *MonitorMetric* callback objects to plot the mean absolute error as a function of walltime.

```
# Plot wall time vs mean absolute error
nlm_mae.plot(x="time", label="Neural Linear")
bnn_mae.plot(x="time", label="Fully Bayesian")
plt.legend()
plt.show()
```

So the neural linear model is a lot faster to train! Both models seem to be able to reach similar levels of accuracy, but the neural linear model just gets to a given accuracy level more quickly.

### 2.8.5 Uncertainty calibration

But is the neural linear model - a mostly non-Bayesian network - at all well-calibrated in terms of its uncertainty estimates? We can measure the calibration of the model (how accurate its uncertainty estimates are) using calibration curves and the mean squared calibration error for regression (see Kuleshov et al., 2018 and Chung et al., 2020).

Essentially, the calibration curve plots the proportion of samples which our model predicts to have less than or equal to some cumulative probability, against the actual proportion of samples. In other words, 10% of samples should fall below the model's predicted lower 10th percentile confidence interval, 20% of samples should fall below the model's predicted lower 20th percentile confidence interval, and so on, all the way up to the 100th percentile. If the model is perfectly calibrated, these values will match, and we'll end up with a calibration curve which goes along the identity line from (0, 0) to (1, 1).

Then, the mean squared calibration error is just a metric of how far our model's calibration deviates from the ideal calibration curve (i.e., what the calibration curve would be if the model were perfectly calibrated - the identity line). Check out the docs for *ContinuousModel.calibration_metric()* for a more formal definition (and for other calibration metrics we could have used).

ProbFlow's *ContinuousModel* class has methods to compute both the calibration curve and various calibration metrics. So, all we need to do to view the calibration curve is:

```
# Plot each model's calibration curve
nlm_model.calibration_curve_plot(
    x_val, y_val, batch_size=10000, label="Neural Linear"
)
bnn_model.calibration_curve_plot(
    x_val, y_val, batch_size=10000, label="Fully Bayesian"
)
plt.legend()
plt.show()
```

Note that for *ContinuousModel.calibration_curve_plot()* above (and for *ContinuousModel.calibration_metric()* below) we're using the `batch_size` keyword argument to perform the computation in batches. Otherwise, if we tried to run the calculation using all ~600,000 samples in the validation data as one single ginormous batch, we'd run out of memory!

To view the mean squared calibration error (MSCE), we can just use *ContinuousModel.calibration_metric()* (lower is better):

```
>>> nlm_model.calibration_metric("msce", x_val, y_val, batch_size=10000)
0.00307
>>> bnn_model.calibration_metric("msce", x_val, y_val, batch_size=10000)
0.00412
```

Both the neural linear model and the fully Bayesian network have pretty good calibration errors! With different data splits and different random initializations, one or the other comes out technically on top, but I'd say they're about the same on average. Given that the neural linear model is so much faster to train, as long as you aren't specifically interested in the posteriors of the neural network parameters, there's definitely an upside to using this hybrid method to get the advantages of Bayesian modeling (uncertainty quantification, separation of epistemic and aleatoric uncertainty, etc) with most of the efficiency advantages of training a normal non-Bayesian neural network!

### 2.8.6 References


- Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Prabhat, Ryan P. Adams. Scalable Bayesian Optimization Using Deep Neural Networks, 2015
- Carlos Riquelme, George Tucker, and Jasper Snoek. Deep Bayesian Bandits Showdown, 2018
- Sebastian W. Ober and Carl Edward Rasmussen. Benchmarking the Neural Linear Model for Regression, 2019
- Kamyar Azizzadenesheli and Animashree Anandkumar. Efficient Exploration through Bayesian Deep Q-Networks, 2019
- Weilin Zhou and Frederic Precioso. Adaptive Bayesian Linear Regression for Automated Machine Learning, 2019
- Youngseog Chung, Willie Neiswanger, Ian Char, Jeff Schneider. Beyond Pinball Loss: Quantile Methods for Calibrated Uncertainty Quantification, 2020


## 2.9 Mixed Effects / Multilevel Models

TODO: description...

### 2.9.1 Linear Mixed Effects Model via Design Matrices

TODO: one way to do it is to manually encode the variables into design matrices...

$$\mathbf{y} \sim \text{Normal}(\mathbf{X}\mathbf{w} + \mathbf{Z}\mathbf{u},\ \sigma)$$

where

- $\mathbf{X}$ is the fixed variables' design matrix,
- $\mathbf{Z}$ is the random variables' design matrix,

- $\mathbf{w}$ is the vector of fixed-effects coefficients,
- $\mathbf{u}$ is the vector of random effects coefficients, and
- $\sigma$ is the noise standard deviation.

TensorFlow

PyTorch

```
import probflow as pf

class LinearMixedEffectsModel(pf.Model):

    def __init__(self, Nf, Nr):
        self.Nf = Nf
        self.w = pf.Parameter([Nf, 1])
        self.u = pf.Parameter([Nr, 1])
        self.sigma = pf.ScaleParameter([1, 1])

    def __call__(self, x):
        X = x[:, :self.Nf]
        Z = x[:, self.Nf:]
        return pf.Normal(X @ self.w() + Z @ self.u(), self.sigma())
```

```
import probflow as pf
import torch

class LinearMixedEffectsModel(pf.Model):

    def __init__(self, Nf, Nr):
        self.Nf = Nf
        self.w = pf.Parameter([Nf, 1])
        self.u = pf.Parameter([Nr, 1])
        self.sigma = pf.ScaleParameter([1, 1])

    def __call__(self, x):
        x = torch.tensor(x)
        X = x[:, :self.Nf]
        Z = x[:, self.Nf:]
        return pf.Normal(X @ self.w() + Z @ self.u(), self.sigma())
```

## 2.10 Autoregressive Models

TODO: description...

### 2.10.1 AR(K) Model

TODO: math

TODO: diagram

TensorFlow

PyTorch

```
import probflow as pf

class AutoregressiveModel(pf.Model):

    def __init__(self, k):
        self.beta = pf.Parameter([k, 1])
        self.mu = pf.Parameter()
        self.sigma = pf.ScaleParameter()

    def __call__(self, x):
        preds = x @ self.beta() + self.mu()
        return pf.Normal(preds, self.sigma())
```

```
import probflow as pf
import torch

class AutoregressiveModel(pf.Model):

    def __init__(self, k):
        self.beta = pf.Parameter([k, 1])
        self.mu = pf.Parameter()
        self.sigma = pf.ScaleParameter()

    def __call__(self, x):
        x = torch.tensor(x)
        preds = x @ self.beta() + self.mu()
        return pf.Normal(preds, self.sigma())
```

If we have a timeseries `x`,

```
N = 1000
x = np.linspace(0, 10, N) + np.random.randn(N)
```

Then we can pull out `k`-length windows into a feature matrix `X`, such that each row of `X` corresponds to a single time point for which we are trying to predict the next sample's value (in `y`).

```
k = 50

X = np.empty([N-k-1, k])
y = x[k:]
for i in range(N-k-1):
    X[i, :] = x[i:i+k]
```

Then, we can create and fit the model:

```
model = AutoregressiveModel(k)
model.fit(X, y)
```

Note that this is exactly equivalent to just doing a linear regression on the lagged feature matrix:

```
model = pf.LinearRegression(k)
model.fit(X, y)
```

### 2.10.2 ARIMA Model

TODO: math

TODO: diagram

TODO: code

## 2.11 Bayesian Correlation

```
import numpy as np
import matplotlib.pyplot as plt
randn = lambda *x: np.random.randn(*x).astype('float32')

import probflow as pf
```

The Pearson correlation coefficient, $\rho$, between two variables is the value that you need to multiply the individual variances of those variables to get the covariance between the two:

$$\rho_{x,y} = \frac{\text{cov}(x, y)}{\text{var}(x)\ \text{var}(y)}$$

To model a correlation probabilistically, we can model our datapoints as being drawn from a multivariate normal distribution, with covariance matrix $\Sigma$. The correlation coefficient is reflected in the off-diagonal elements of the correlation matrix (we'll just assume the data is normalized, and so all the diagonal elements are 1).

This is because (according to the definition of the correlation coefficient), the covariance matrix can be expressed using only the per-variable variances and the correlation coefficient:

$$\Sigma = \begin{bmatrix} \text{var}(x) & \text{cov}(x, y) \\ \text{cov}(x, y) & \text{var}(y) \end{bmatrix} = \begin{bmatrix} \text{var}(x) & \rho\ \text{var}(x)\text{var}(y) \\ \rho\ \text{var}(x)\text{var}(y) & \text{var}(y) \end{bmatrix}$$

Or, if we assume the input has been standardized:

$$\Sigma = \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}$$

So, basically what we're doing is optimizing the correlation coefficient such that it gives us a Gaussian distribution which best fits the data points.

This is our first example of a generative model, where we're not trying to *predict* $y$ given $x$. Instead, we're interested in the data-generating distribution itself, from which the correlation coefficient can be derived. With this type of

generative model, we'll treat all the data as the dependent variable, and so our `__call__` function will have no input. The only thing predicting the distribution of the data is the model and its parameters.

TensorFlow

PyTorch

```
import tensorflow as tf

class BayesianCorrelation(pf.Model):

    def __init__(self):
        self.rho = pf.BoundedParameter(min=-1, max=1)

    def __call__(self):
        cov = tf.eye(2) + self.rho()*tf.abs(tf.eye(2)-1)
        return pf.MultivariateNormal(tf.zeros([2]), cov)
```

```
import torch

class BayesianCorrelation(pf.Model):

    def __init__(self):
        self.rho = pf.BoundedParameter(min=-1, max=1)

    def __call__(self):
        cov = torch.eye(2) + self.rho()*torch.abs(torch.eye(2)-1)
        return pf.MultivariateNormal(torch.zeros([2]), cov)
```

Then we can instantiate the model.

```
model = BayesianCorrelation()
```

Let's generate some uncorrelated data.

```
X = np.random.randn(100, 2).astype('float32')
plt.plot(X[:, 0], X[:, 1], '.')
```

If we fit the model on some data which is uncorrelated, we can see that the posterior distribution for the correlation coefficient $\rho$ is centered around 0:

```
model.fit(X, lr=0.1)
model.posterior_plot(ci=0.95, style='hist')
```

On the other hand, if we fit the model to some data which is highly correlated,

```
X[:, 1] = X[:, 0] + 0.2*np.random.randn(100).astype('float32')
plt.plot(X[:, 0], X[:, 1], '.')
```

Then the posterior distribution for the correlation coefficient $\rho$ is considerably closer to 1:

```
model = BayesianCorrelation()
model.fit(X, lr=0.1)
model.posterior_plot(ci=0.95, style='hist')
```

Conversely, if we generate negatively correlated data,

```
X = np.random.randn(100, 2).astype('float32')
X[:, 1] = -X[:, 0] + 0.2*np.random.randn(100).astype('float32')
plt.plot(X[:, 0], X[:, 1], '.')
```

The model recovers the negative correlation coefficient:

```
model = BayesianCorrelation()
model.fit(X, lr=0.1)
model.posterior_plot(ci=0.95, style='hist')
```

## 2.12 Gaussian Mixture Model

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

import probflow as pf
```

Another type of generative model is a mixture model, where the distribution of datapoints is modeled as the combination ("mixture") of multiple individual distributions. A common type of mixture model is the Gaussian mixture model, where the data-generating distribution is modeled as the mixture of several Gaussian distributions.

Here's some data generated by sampling points from three two-dimensional Gaussian distributions:

```
# Generate some data
N = 3*1024
X = np.random.randn(N, 2).astype('float32')
X[:1024, :] += [2, 0]
X[1024:2048, :] -= [2, 4]
X[2048:, :] += [-2, 4]

# Plot the data
plt.plot(X[:, 0], X[:, 1], '.', alpha=0.2)
```

Let's model the data using a Bayesian Gaussian mixture model. The model has $k \in 1, \ldots, K$ mixture components - we'll use multivariate normal distributions. To match the data we generated, we'll use $K = 3$ mixture components in $D = 2$ dimensions.

Each of the $K$ normal distributions has a mean ($\mu$) and a standard deviation ($\sigma$) in each dimension. For simplicity we'll assume the covariance of the Gaussians are diagonal. Each of the mixture distributions also has a weight ($\theta$), where all the weights sum to 1.

The probability of a datapoint $i$ being generated by mixture component $k$ is modeled with a categorical distribution, according to the weights:

$$k_i \sim \text{Categorical}(\boldsymbol{\theta})$$

And then the likelihood of that datapoint's observed values are determined by the $k$-th mixture component's distribution:

$$\mathbf{y}_i \sim \mathcal{N}_D(\boldsymbol{\mu}_{k_i}, \boldsymbol{\sigma}_{k_i})$$

Let's make that model using ProbFlow. We'll use `DirichletParameter` for the weights, which uses a Dirichlet distribution as the variational posterior, because the weights must sum to 1. As with the *correlation model*, this is a generative model - we're not predicting $y$ given $x$, but rather are just fitting the data-generating distribution - and so the `__call__` method has no inputs.

```
class GaussianMixtureModel(pf.Model):

    def __init__(self, k, d):
        self.mu = pf.Parameter([k, d])
        self.sigma = pf.ScaleParameter([k, d])
        self.theta = pf.DirichletParameter(k)

    def __call__(self):
        dists = tfd.MultivariateNormalDiag(self.mu(), self.sigma())
        return pf.Mixture(dists, probs=self.theta())
```

Compare constructing the above model (using ProbFlow) to the complexity of constructing the model with "raw" TensorFlow and TensorFlow Probability.

Then, we can instantiate the model and fit it to the data!

```
model = GaussianMixtureModel(3, 2)
model.fit(X, lr=0.03, epochs=500, batch_size=1024)
```

To look at the fit mixture density over possible values of $X$, we can compute and plot the probability of the model over a grid:

```
# Compute log likelihood at each point on a grid
Np = 100 #number of grid points
xx = np.linspace(-6,6,Np)
Xp, Yp = np.meshgrid(xx, xx)
Pp = np.column_stack([Xp.ravel(), Yp.ravel()])
probs = model.prob(Pp.astype('float32'))
probs = np.reshape(probs, (Np, Np))

# Show the fit mixture density
plt.imshow(probs,
           extent=(-6, 6, -6, 6),
           origin='lower')
```

The density lines up well with the points used to fit the model!

```
# Plot the density and original points
plt.plot(X[:, 0], X[:, 1], ',', zorder=-1)
plt.contour(xx, xx, probs)
```

## 2.13 Stochastic Volatility Model

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
import probflow as pf
```

Stochastic volatility models are often used to model the variability of stock prices over time. The volatility is the standard deviation of the logarithmic returns over time. Instead of assuming that the volatility is constant, stochastic volatility models have latent parameters which model the volatility at each moment in time.

This example is pretty similar to the PyMC example stochastic volatility model, with a few differences specific to stochastic variational inference (as opposed to the PyMC example which uses MCMC).

### 2.13.1 Data

We'll fit a model to the volatility of the S&P 500 day-by-day returns. S&P 500 performance data can be found here. Let's load the data from the past three years.

```
df = pd.read_csv('PerformanceGraphExport.csv')
```

We can view the raw index values over time:

```
# Plot raw S&P 500 performance
plt.plot(df['S&P 500'])
plt.ylabel('S&P 500')
plt.xlabel('Days since 11/2/2016')
```

But we can also compute the difference in logarithmic returns, which we'll model to estimate the volatility. Note that at some time points the index is not very volatile (e.g. from time 100-250), while at other times the index is very volatile (e.g. from 300-400 and at ~500).

```
# Compute logarithmic returns
y = df['S&P 500'].values
y = np.log(y[1:]) - np.log(y[:-1])
y = y.reshape((1, y.shape[0])).astype('float32')
N = y.shape[1]

# Plot it
plt.plot(y.T)
plt.ylabel('Logarithmic Returns')
plt.xlabel('Days since 11/2/2016')
```

### 2.13.2 Model

At each timepoint ($i$) we'll model the logarithmic returns at that timepoint ($y_i$). The model allows the volatility to change over time, such that each time point has a volatility controlled by a parameter for that time point ($s_i$). We'll use a Student t-distribution to model the logarithmic returns, with degrees of freedom $\nu$ (a free parameter):

$$y_i \sim \text{StudentT}(\nu,\ 0,\ \exp(s_i))$$

However, we can't allow the scale parameters ($s_i$) at each timepoint to be completely independent, or the model will just overfit the data! So, we'll constrain the volatility at any timepoint $i$ to be similar to the volatility at the previous timepoint $i-1$, and add another parameter $\sigma$ which controls how quickly the volatility can change over time:

$$s_i \sim \text{Normal}(s_{i-1},\ \sigma)$$

We'll use normal distributions as the variational posteriors for each $s$ parameter, and log normal distributions for $\nu$ and $\sigma$, with the following priors:

$$\log(\sigma) \sim \text{Normal}(-3,\ 0.2)$$
$$\log(\nu) \sim \text{Normal}(0,\ 1)$$

Let's build this model with ProbFlow. There's one major difference between the PyMC version (which uses MCMC) and our model (which uses SVI): because the prior on each $s_i$ is Normal($s_{i-1}$, $\sigma$), in `__call__` we'll add the KL loss due to the divergence between those two distributions (as we would with a regular parameter's variational posterior and prior).

```
class StochasticVolatility(pf.Model):

    def __init__(self, N):
        self.s = pf.Parameter([1, N], prior=None)
        self.sigma = pf.Parameter(name='sigma',
                                  prior=pf.Normal(-3, 0.1),
                                  transform=tf.exp)
        self.nu = pf.Parameter(name='nu',
                               prior=pf.Normal(0, 1),
                               transform=tf.exp)

    def __call__(self):
        s_posteriors = pf.Normal(self.s.posterior().mean()[:, 1:],
                                 self.s.posterior().stddev()[:, 1:])
        s_priors = pf.Normal(self.s.posterior().mean()[:, :-1],
                             self.sigma())
        self.add_kl_loss(s_posteriors, s_priors)
        return pf.StudentT(self.nu(), 0, tf.exp(self.s()))
```

The we can instantiate the model,

```
model = StochasticVolatility(N)
```

And fit it to the data!

```
model.fit(y, batch_size=1, epochs=1000)
```

### 2.13.3 Inspecting the Fit

We can take a look at the posterior distributions for the $\sigma$ and $\nu$ parameters:

```
model.posterior_plot(['sigma', 'nu'], ci=0.9)
```

But more importantly, we can plot the MAP estimate of the volatility over time:

```
plt.plot(y.T)
plt.plot(np.exp(model.s.posterior_mean().T))
plt.legend(['S&P 500', 'Volatility'])
```

And since this is a Bayesian model, we also have access to uncertainty as to the amount of volatility at each time point:

```
# Sample from the posterior
Ns = 50
samples = model.s.posterior_sample(Ns).reshape((Ns, N))

# Plot the posterior over time
plt.plot(y.T)
plt.plot(np.exp(samples.T), 'r', alpha=0.05)
plt.legend(['S&P 500', 'Volatility'])
plt.show()
```

## 2.14 Normalizing Flows

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions
tfb = tfp.bijectors

import probflow as pf
```

TODO: description, math, diagram

---

**Note:** This example only works with the TensorFlow backend (using bijectors), but one could implement a similar model using PyTorch transforms.

---

Let's create some data which has a wonky shape, and which would be difficult to model with a standard probability distribution.

```
N = 512
x = 3*np.random.randn(N, 2)
x[:, 0] = 0.25*np.square(x[:, 1])
x[:, 0] += np.random.randn(N)
plt.plot(x[:, 0], x[:, 1], '.')
```

To create the normalizing flow, we'll first create a bijector to represent an invertible leaky rectified linear transformation.

```
class LeakyRelu(tfb.Bijector):

    def __init__(self, alpha=0.5):
        super().__init__(forward_min_event_ndims=0)
        self.alpha = alpha

    def _forward(self, x):
        return tf.where(x>=0, x, x*self.alpha)

    def _inverse(self, y):
        return tf.where(y>=0, y, y/self.alpha)

    def _inverse_log_det_jacobian(self, y):
        return tf.math.log(tf.where(y>=0, 1., 1./self.alpha))
```

The source distribution will be a standard multivariate normal distribution, and the affine transformations and "leakiness" of the rectified linear transformations will be parameterized by *DeterministicParameter* parameters, which are non-probabilistic (but still have priors).

```
class MlpNormalizingFlow(pf.Model):

    def __init__(self, Nl, d):
        self.base_dist = tfd.MultivariateNormalDiag([0., 0.])
        self.V = [pf.DeterministicParameter([d, d]) for _ in range(Nl)]
        self.s = [pf.DeterministicParameter([d]) for _ in range(Nl)]
        self.L = [pf.DeterministicParameter([int(d*(d+1)/2)]) for _ in range(Nl)]
        self.a = [pf.DeterministicParameter([1]) for _ in range(Nl-1)]

    def __call__(self, n_steps=None):
        n_steps = 2*len(self.V)-1 if n_steps is None else n_steps
        bijectors = []
        for i in range(len(self.V)):
            bijectors += [tfb.Affine(
                scale_tril=tfb.ScaleTriL().forward(self.L[i]()),
                scale_perturb_factor=self.V[i](),
                shift=self.s[i]())]
            if i < len(self.V)-1:
                bijectors += [LeakyRelu(alpha=tf.abs(self.a[i]()))]
        return tfd.TransformedDistribution(
            distribution=self.base_dist,
            bijector=tfb.Chain(bijectors[:n_steps]))
```

Then we can create and fit the model to the data:

```
Nl = 8 #number of layers
model = MlpNormalizingFlow(Nl, 2)
model.fit(x, epochs=1e4, lr=0.02)
```

Comparing the data points to samples from the base distribution transformed by the normalizing flow, we can see that the transformations have warped the base distribution to match the distribution of the data.

```
# Plot original points and samples from the model
plt.plot(x[:, 0], x[:, 1], '.')
```

(continues on next page)

(continued from previous page)

```
S = model().sample((512,))
plt.plot(S[:, 0], S[:, 1], '.')
```

And because normalizing flows are made up of invertible transformations, we can evaluate the probability of the distribution at any arbitrary point on a grid and compare that transformed probability distribution to the original data:

```
# Compute probabilities
res = 200
xx, yy = np.meshgrid(np.linspace(-5, 20, res), np.linspace(-10, 10, res))
xy = np.hstack([e.reshape(-1, 1) for e in [xx, yy]])
probs = model().prob(xy).numpy()

# Plot them
plt.imshow(probs.reshape((res, res)),
           origin='lower', extent=(-5, 20, -10, 10))
plt.plot(x[:, 0], x[:, 1], 'r,')
```

And we can use the `n_steps` kwarg to view samples after each successive transformation in the flow. We start with the base distribution (a standard multivariate Gaussian), and the flow stepwise distorts and transforms that distribution until it approximates the data.

```
for i in range(2*Nl):
    plt.subplot(4, 4, i+1)
    S = model(n_steps=i).sample((512,))
    plt.plot(S[:, 0], S[:, 1], '.')
```

Also see:

- Eric Jang's great tutorial on normalizing flows, which this example is based on.
- Danilo Jimenez Rezende & Shakir Mohamed. Variational Inference with Normalizing Flows. *PLMR*, 2015.

## 2.15 Probabilistic PCA

TODO: description...

TODO: math

TODO: diagram

TensorFlow

PyTorch

```
import probflow as pf
import tensorflow as tf

class PPCA(pf.Model):

```

(continues on next page)

(continued from previous page)

```
    def __init__(self, d, q):
        self.W = pf.Parameter(shape=[d, q])
        self.sigma = pf.ScaleParameter()

    def __call__(self):
        W = self.W()
        cov = W @ tf.transpose(W) + self.sigma()*tf.eye(W.shape[0])
        return pf.MultivariateNormal(tf.zeros(W.shape[0]), cov)
```

```
import probflow as pf
import torch

class PPCA(pf.Model):

    def __init__(self, d, q):
        self.W = pf.Parameter(shape=[d, q])
        self.sigma = pf.ScaleParameter()

    def __call__(self):
        W = self.W()
        cov = W @ torch.t(W) + self.sigma()*torch.eye(W.shape[0])
        return pf.MultivariateNormal(torch.zeros(W.shape[0]), cov)
```

## 2.16 Latent Dirichlet Allocation

TODO: intro, link to colab w/ these examples

TODO: math

$$
\begin{aligned}
N_t &= \text{number of topics} && (2.1)\\
N_w &= \text{number of words} && (2.2)\\
N_d &= \text{number of documents} && (2.3)\\
\varphi_{k=1\ldots N_t} &\sim \text{Dirichlet}_{N_w}(\beta) && (2.4)\\
\theta_{d=1\ldots N_d} &\sim \text{Dirichlet}_{N_t}(\alpha) && (2.5)\\
\mathbf{W} &\sim \text{OneHotCategorical}(\theta, \varphi) && (2.6)
\end{aligned}
$$

TODO: diagram

TODO: explain in terms of $\varphi$ and $\theta$ parameter matrixes

TensorFlow

PyTorch

```
import probflow as pf

class LDA(pf.Model):

    def __init__(self, Nt, Nd, Nw):
        self.phi = pf.DirichletParameter(Nw, Nt)    #per-topic word dists
        self.theta = pf.DirichletParameter(Nt, Nd) #per-document topic dists

    def __call__(self, x):
        probs = self.theta[x[:, 0]] @ self.phi()
        return pf.OneHotCategorical(probs=probs)
```

```
import probflow as pf
import torch

class LDA(pf.Model):

    def __init__(self, Nt, Nd, Nw):
        self.phi = pf.DirichletParameter(Nw, Nt)    #per-topic word dists
        self.theta = pf.DirichletParameter(Nt, Nd) #per-document topic dists

    def __call__(self, x):
        x = torch.tensor(x)
        probs = self.theta[x[:, 0]] @ self.phi()
        return pf.OneHotCategorical(probs=probs)
```

To fit the model in this way, `x` will be document IDs, and `y` will be a matrix of size `(Ndocuments, Nwords)`.

```
# Nt = number of topics to use
# Nd = number of documents
# Nw = number of words in the vocabulary
# W = (Nd, Nw)-size matrix of per-document word probabilities
doc_id = np.arange(W.shape[0])

model = LDA(Nt, Nd, Nw)
model.fit(doc_id, W)
```

TODO: Alternatively, when you have a LOT of documents, it's inefficient to try and infer that huge Nd-by-Nt matrix of parameters, so you can use a neural net to estimate the topic distribution from the word distributions (amortize). Its... kinda sorta like an autoencoder, where you're encoding documents into weighted mixtures of topics, and then decoding the word distributions from those topic distributions.

TensorFlow

PyTorch

```
class LdaNet(pf.Model):

    def __init__(self, dims):
        self.phi = pf.DirichletParameter(dims[0], dims[-1])
        self.net = pf.DenseNetwork(dims)

    def __call__(self, x):
        probs = self.net(x) @ self.phi()
        return pf.OneHotCategorical(probs=probs)
```

```
class LdaNet(pf.Model):

    def __init__(self, dims):
        self.phi = pf.DirichletParameter(dims[0], dims[-1])
        self.net = pf.DenseNetwork(dims)

    def __call__(self, x):
        x = torch.tensor(x)
        probs = self.net(x) @ self.phi()
        return pf.OneHotCategorical(probs=probs)
```

TODO: And then when fitting the model we'll use the per-document word frequency matrix as both `x` and `y`:

```
model = LdaNet([Nw, 128, 128, 128, Nt])
model.fit(W, W)
```

## 2.17 Entity Embeddings

TODO: explain embeddings, using the *Embedding* module

TODO: model, embedding of categorical features (1st column of x) and combining with continuous features (rest of x) and predicting with fully-connected neural net on top, which does binary classification

TensorFlow

PyTorch

```
import probflow as pf
import tensorflow as tf

class EmbeddingRegression(pf.Model):

    def __init__(self, k, Dcat, Dcon):
        self.emb = pf.Embedding(k, Dcat)
        self.net = pf.DenseNetwork([Dcat+Dcon, 1])

    def __call__(self, x):
        embeddings = self.emb(x[:, 0])
        logits = self.net(tf.concat([embeddings, x[:, 1:]], -1))
        return pf.Bernoulli(logits)
```

```
import probflow as pf
import torch

class EmbeddingRegression(pf.Model):

    def __init__(self, k, Dcat, Dcon):
        self.emb = pf.Embedding(k, Dcat)
        self.net = pf.DenseNetwork([Dcat+Dcon, 1])

    def __call__(self, x):
        x = torch.tensor(x)
        embeddings = self.emb(x[:, 0])
        logits = self.net(torch.cat([embeddings, x[:, 1:]], -1))
        return pf.Bernoulli(logits)
```

## 2.18 Neural Matrix Factorization

TODO: description...

### 2.18.1 Matrix Factorization

TODO: for a vanilla matrix factorization, description, diagram, math (with binary interactions)

TensorFlow

PyTorch

```
import probflow as pf
import tensorflow as tf

class MatrixFactorization(pf.Model):

    def __init__(self, Nu, Ni, Nd):
        self.user_emb = pf.Embedding(Nu, Nd)
        self.item_emb = pf.Embedding(Ni, Nd)

    def __call__(self, x):
        user_vec = self.user_emb(x['user_id'])
        item_vec = self.item_emb(x['item_id'])
        logits = user_vec @ tf.transpose(item_vec)
        return pf.Bernoulli(logits)
```

```
import probflow as pf
import torch

class MatrixFactorization(pf.Model):

    def __init__(self, Nu, Ni, Nd):
        self.user_emb = pf.Embedding(Nu, Nd)
        self.item_emb = pf.Embedding(Ni, Nd)

    def __call__(self, x):
        user_vec = self.user_emb(torch.tensor(x['user_id']))
        item_vec = self.item_emb(torch.tensor(x['item_id']))
        logits = user_vec @ torch.t(item_vec)
        return pf.Bernoulli(logits)
```

TODO: Then can instantiate the model

```
#df = DataFrame w/ 3 columns: 'user_id', 'item_id', and 'rating'
Nu = df['user_id'].nunique() #number of users
Ni = df['item_id'].nunique() #number of items
Nd = 50 #number of embedding dimensions
model = MatrixFactorization(Nu, Ni, Nd)
```

TODO: Then fit it;

```
model.fit(df[['user_id', 'item_id']], df['rating'])
```

### 2.18.2 Neural Collaborative Filtering

TODO: description, diagram, math TODO: cite https://arxiv.org/abs/1708.05031

TensorFlow

PyTorch

```
class MatrixFactorization(pf.Model):

    def __init__(self, Nu, Ni, Nd, dims):
        self.user_emb = pf.Embedding(Nu, Nd)
        self.item_emb = pf.Embedding(Ni, Nd)
        self.net = pf.DenseNetwork(dims)

    def __call__(self, x):
        user_vec = self.user_emb(x['user_id'])
        item_vec = self.item_emb(x['item_id'])
        logits = self.net(tf.concat([user_vec, item_vec], axis=1))
        return pf.Bernoulli(logits)
```

```
class MatrixFactorization(pf.Model):

    def __init__(self, Nu, Ni, Nd, dims):
        self.user_emb = pf.Embedding(Nu, Nd)
        self.item_emb = pf.Embedding(Ni, Nd)
        self.net = pf.DenseNetwork(dims)

    def __call__(self, x):
        user_vec = self.user_emb(torch.tensor(x['user_id']))
        item_vec = self.item_emb(torch.tensor(x['item_id']))
        logits = self.net(torch.cat([user_vec, item_vec], 1))
        return pf.Bernoulli(logits)
```

### 2.18.3 Neural Matrix Factorization

or for neural matrix factorization https://arxiv.org/abs/1708.05031

TensorFlow

PyTorch

```
class NeuralMatrixFactorization(pf.Model):

    def __init__(self, Nu, Ni, Nd, dims):
        self.user_mf = pf.Embedding(Nu, Nd)
        self.item_mf = pf.Embedding(Ni, Nd)
        self.user_ncf = pf.Embedding(Nu, Nd)
        self.item_ncf = pf.Embedding(Ni, Nd)
        self.net = pf.DenseNetwork(dims)
        self.linear = pf.Dense(dims[-1]+Nd)

    def __call__(self, x):
        user_mf = self.user_mf(x['user_id'])
        item_mf = self.item_mf(x['item_id'])
        user_ncf = self.user_ncf(x['user_id'])
        item_ncf = self.item_ncf(x['item_id'])
```

(continues on next page)

(continued from previous page)

```
        preds_mf = user_mf*item_mf
        preds_ncf = self.net(tf.concat([user_ncf, item_ncf], axis=1))
        logits = self.linear(tf.concat([preds_mf, preds_ncf], axis=1))
        return pf.Bernoulli(logits)
```

```
class NeuralMatrixFactorization(pf.Model):

    def __init__(self, Nu, Ni, Nd, dims):
        self.user_mf = pf.Embedding(Nu, Nd)
        self.item_mf = pf.Embedding(Ni, Nd)
        self.user_ncf = pf.Embedding(Nu, Nd)
        self.item_ncf = pf.Embedding(Ni, Nd)
        self.net = pf.DenseNetwork(dims)
        self.linear = pf.Dense(dims[-1]+Nd)

    def __call__(self, x):
        uid = torch.tensor(x['user_id'])
        iid = torch.tensor(x['item_id'])
        user_mf = self.user_mf(uid)
        item_mf = self.item_mf(iid)
        user_ncf = self.user_ncf(uid)
        item_ncf = self.item_ncf(iid)
        preds_mf = user_mf*item_mf
        preds_ncf = self.net(torch.cat([user_ncf, item_ncf], 1))
        logits = self.linear(torch.cat([preds_mf, preds_ncf], 1))
        return pf.Bernoulli(logits)
```

TODO: Then can instantiate the model

```
#df = DataFrame w/ 3 columns: 'user_id', 'item_id', and 'rating'
Nu = df['user_id'].nunique() #number of users
Ni = df['item_id'].nunique() #number of items
Nd = 50 #number of embedding dimensions
dims = [Nd*2, 128, 64, 32]
model = NeuralMatrixFactorization(Nu, Ni, Nd, dims)
```

TODO: Then fit it;

```
model.fit(df[['user_id', 'item_id']], df['rating'])
```

## 2.19 Batch Normalization

TODO: intro, math, diagram

Batch normalization can be performed using the *BatchNormalization* Module. For example, to add batch normalization to the dense neural network from the *previous example*:

TensorFlow

PyTorch

```
import probflow as pf
import tensorflow as tf
```

(continues on next page)

(continued from previous page)

```
class DenseRegression(pf.Model):

    def __init__(self):
        self.net = pf.Sequential([
            pf.Dense(5, 128),
            pf.BatchNormalization(128),
            tf.nn.relu,
            pf.Dense(128, 64),
            pf.BatchNormalization(64),
            tf.nn.relu,
            pf.Dense(64, 1),
        ])
        self.s = pf.ScaleParameter()

    def __call__(self, x):
        return pf.Normal(self.net(x), self.s())
```

```
import probflow as pf
import torch

class DenseRegression(pf.Model):

    def __init__(self):
        self.net = pf.Sequential([
            pf.Dense(5, 128),
            pf.BatchNormalization(128),
            torch.nn.ReLU(),
            pf.Dense(128, 64),
            pf.BatchNormalization(64),
            torch.nn.ReLU(),
            pf.Dense(64, 1),
        ])
        self.s = pf.ScaleParameter()

    def __call__(self, x):
        x = torch.tensor(x)
        return pf.Normal(self.net(x), self.s())
```

## 2.20 Mixture Density Network

TODO: note that last element of head_dims should be the desired number of mixture components

TensorFlow

PyTorch

```
import probflow as pf

class MixtureDensityNetwork(pf.Model):

    def __init__(self, dims, head_dims):
        self.core = pf.DenseNetwork(dims+[head_dims[0]])
        self.heads = [pf.DenseNetwork(head_dims) for _ in range(3)]
```

(continues on next page)

(continued from previous page)

```
    def __call__(self, x):
        x = self.core(x)
        preds = [h(x) for h in self.heads]
        return pf.Mixture(pf.Normal(preds[0], preds[1]), preds[2])
```

```
import probflow as pf
import torch

class MixtureDensityNetwork(pf.Model):

    def __init__(self, dims, head_dims):
        self.core = pf.DenseNetwork(dims+[head_dims[0]])
        self.heads = [pf.DenseNetwork(head_dims) for _ in range(3)]

    def __call__(self, x):
        x = torch.tensor(x)
        x = self.core(x)
        preds = [h(x) for h in self.heads]
        return pf.Mixture(pf.Normal(preds[0], preds[1]), preds[2])
```

TODO: cite Christopher M. Bishop, Mixture density networks, 1994

## 2.21 Variational Autoencoder

TODO: intro, link to colab w/ these examples

TODO: math

TODO: diagram

TODO: talk about how this model requires adding additional KL divergence term (for the latent representation posterior vs prior).

TensorFlow

PyTorch

```
import probflow as pf

class VariationalAutoencoder(pf.ContinuousModel):

    def __init__(self, dims)
        self.encoder = pf.DenseRegression(dims, heteroscedastic=True)
        self.decoder = pf.DenseRegression(dims[::-1], heteroscedastic=True)

    def __call__(self, x):
        z = self.encoder(x)
        self.add_kl_loss(z, pf.Normal(0, 1))
        return self.decoder(z.sample())
```

```
import probflow as pf
import torch

class VariationalAutoencoder(pf.ContinuousModel):

```

(continues on next page)

(continued from previous page)

```
    def __init__(self, dims)
        self.encoder = pf.DenseRegression(dims, heteroscedastic=True)
        self.decoder = pf.DenseRegression(dims[::-1], heteroscedastic=True)

    def __call__(self, x):
        x = torch.tensor(x)
        z = self.encoder(x)
        self.add_kl_loss(z, pf.Normal(0, 1))
        return self.decoder(z.sample())
```

Then we can create an instance of the model, defining the dimensionality of each layer of the network:

```
model = VariationalAutoencoder([7, 128, 64, 32, 3])
```

TODO: generate some data, and then fit

```
model.fit(x, x)
```

## 2.22 Generative Adversarial Network

ProbFlow isn't really built for the kind of flexibility you need to fit a GAN, so I wouldn't recommend fitting GANs with ProbFlow. That said, you can *technically* do it.

So... Just for fun...

TODO: description... (Bayesian GAN, cite https://arxiv.org/pdf/1705.09558.pdf)

TODO: math

TODO: diagram

TODO: talk about overriding the `Model.log_likelihood` method to compute the log likelihood due to the batch of acual data *and* a batch of the same size of data generated by the generator network.

First let's build a generator:

TensorFlow

PyTorch

```
import probflow as pf
import tensorflow as tf

class Generator(pf.Model):

    def __init__(self, dims):
        self.Dz = dims[0]
        self.G = pf.DenseNetwork(dims)
        self.D = None

    def __call__(self, x):
        z = tf.random.normal([x.shape[0], self.Dz])
        return self.G(z)

    def log_likelihood(self, _, x):
        labels = tf.ones([x.shape[0], 1])
```

(continues on next page)

(continued from previous page)

```
        true_ll = self.D(self(x)).log_prob(labels)
        return tf.reduce_sum(true_ll)
```

```
import probflow as pf
import torch

class Generator(pf.Model):

    def __init__(self, dims):
        self.Dz = dims[0]
        self.G = pf.DenseNetwork(dims)
        self.D = None

    def __call__(self, x):
        x = torch.tensor(x)
        z = torch.randn([x.shape[0], self.Dz])
        return self.G(z)

    def log_likelihood(self, _, x):
        labels = torch.ones([x.shape[0], 1])
        true_ll = self.D(self(x)).log_prob(labels)
        return torch.sum(true_ll)
```

Then a discriminator:

TensorFlow

PyTorch

```
class Discriminator(pf.Model):

    def __init__(self, dims):
        self.G = None
        self.D = pf.DenseNetwork(dims)

    def __call__(self, x):
        return pf.Bernoulli(self.D(x))

    def log_likelihood(self, _, x):
        labels = tf.ones([x.shape[0], 1])
        true_ll = self(x).log_prob(labels)
        fake_ll = self(self.G(x)).log_prob(0*labels)
        return tf.reduce_sum(true_ll + fake_ll)
```

```
class Discriminator(pf.Model):

    def __init__(self, dims):
        self.G = None
        self.D = pf.DenseNetwork(dims)

    def __call__(self, x):
        x = torch.tensor(x)
        return pf.Bernoulli(self.D(x))

    def log_likelihood(self, _, x):
        labels = torch.ones([x.shape[0], 1])
        true_ll = self(x).log_prob(labels)
```

(continues on next page)

(continued from previous page)

```
        fake_ll = self(self.G(x)).log_prob(0*labels)
        return torch.sum(true_ll + fake_ll)
```

And a callback to train the generator for an epoch at the end of each epoch of the discriminator's training:

```
class TrainGenerator(pf.Callback):

    def __init__(self, G, x):
        self.G = G
        self.x = x

    def on_epoch_end(self):
        self.G.fit(self.x, epochs=1)
```

Then, we can instantiate the networks, the callback, and fit the network:

```
# x is a numpy array or pandas DataFrame of real data
Nf = 7 #number of features / input dimensionality (x.shape[1])
Nz = 3 #number of latent dimensions

# Create the networks
G = Generator([Nz, 256, 128, Nf])
D = Discriminator([Nf, 256, 128, 1])

# Let them know about each other <3
G.D = lambda x: D(x)
D.G = lambda x: G(x)

# Create the callback which trains the generator
train_g = TrainGenerator(G, x)

# Fit both models by fitting the discriminator w/ the callback
D.fit(x, callbacks=[train_g])
```

Note that we use lambda functions instead of simply assigning the opposite net as an attribute of the each model instance. This is because ProbFlow recursively searches a model instance's attributes for *Module* and *Parameter* objects, and optimizes all found parameters with respect to the loss. However, we don't want the discriminator's parameters updated with the the generator's loss, or vice versa! ProbFlow even looks for parameters in attributes which are lists and dictionaries, but it doesn't look in lambda functions, so we can "hide" the parameters of one model from the other that way, while still allowing each model to `__call__` the other to compute its own loss.

CHAPTER

THREE

# API

## 3.1 Distributions

The *distributions* module contains classes to instantiate probability distributions, which describe the likelihood of either a parameter or a datapoint taking any given value. Distribution objects are used to represent both the predicted probability distribution of the data, and also the parameters' posteriors and priors.

### 3.1.1 Continuous Distributions

- *Deterministic*
- *Normal*
- *MultivariateNormal*
- *StudentT*
- *Cauchy*
- *Gamma*
- *InverseGamma*

### 3.1.2 Discrete Distributions

- *Bernoulli*
- *Categorical*
- *OneHotCategorical*
- *Poisson*
- *Dirichlet*

### 3.1.3 Other

- *Mixture*
- *HiddenMarkovModel*

---

**class** probflow.distributions.**Bernoulli**(*logits=None*, *probs=None*)
Bases: *probflow.utils.base.BaseDistribution*

The Bernoulli distribution.

The Bernoulli distribution is a discrete distribution defined over only two integers: 0 and 1. It has one parameter:

- a probability parameter ($0 \leq p \leq 1$).

A random variable $x$ drawn from a Bernoulli distribution

$$x \sim \text{Bernoulli}(p)$$

takes the value 1 with probability $p$, and takes the value 0 with probability $p - 1$.

TODO: example image of the distribution

TODO: specifying either logits or probs

**Parameters**

- **logits** (int, float, ndarray, or Tensor) – Logit-transformed probability parameter of the Bernoulli distribution ()
- **probs** (int, float, ndarray, or Tensor) – Logit-transformed probability parameter of the Bernoulli distribution ()

**cdf**(*y*)
Cumulative probability of some data along this distribution

**log_prob**(*y*)
Compute the log probability of some data given this distribution

**mean**()
Compute the mean of this distribution

Note that this uses the mode of distributions for which the mean is undefined (for example, a categorical distribution)

**mode**()
Compute the mode of this distribution

**prob**(*y*)
Compute the probability of some data given this distribution

**sample**(*n=1*)
Generate a random sample from this distribution

**class** probflow.distributions.**Categorical**(*logits=None*, *probs=None*)
Bases: *probflow.utils.base.BaseDistribution*

The Categorical distribution.

The Categorical distribution is a discrete distribution defined over $N$ integers: 0 through $N - 1$. A random variable $x$ drawn from a Categorical distribution

$$x \sim \text{Categorical}(\theta)$$

has probability

$$p(x = i) = p_i$$

TODO: example image of the distribution

TODO: logits vs probs

**Parameters**

- **logits** (int, float, ndarray, or Tensor) – Logit-transformed category probabilities ($\frac{\theta}{1-\theta}$)
- **probs** (int, float, ndarray, or Tensor) – Raw category probabilities ($\theta$)

**prob**(*y*)
Doesn't broadcast correctly when logits/probs and y are same dims

**cdf**(*y*)
Cumulative probability of some data along this distribution

**log_prob**(*y*)
Doesn't broadcast correctly when logits/probs and y are same dims

**mean**()
Compute the mean of this distribution

Note that this uses the mode of distributions for which the mean is undefined (for example, a categorical distribution)

**mode**()
Compute the mode of this distribution

**sample**(*n=1*)
Generate a random sample from this distribution

**class** probflow.distributions.**Cauchy**(*loc=0*, *scale=1*)
Bases: *probflow.utils.base.BaseDistribution*

The Cauchy distribution.

The Cauchy distribution is a continuous distribution defined over all real numbers, and has two parameters:

- a location parameter (`loc` or $\mu$) which determines the median of the distribution, and
- a scale parameter (`scale` or $\gamma > 0$) which determines the spread of the distribution.

A random variable $x$ drawn from a Cauchy distribution

$$x \sim \text{Cauchy}(\mu, \gamma)$$

has probability

$$p(x) = \frac{1}{\pi\gamma\left[1 + \left(\frac{x-\mu}{\gamma}\right)^2\right]}$$

The Cauchy distribution is equivalent to a Student's t-distribution with one degree of freedom.

TODO: example image of the distribution

**Parameters**

- **loc** (int, float, ndarray, or Tensor) – Median of the Cauchy distribution ($\mu$). Default = 0
- **scale** (int, float, ndarray, or Tensor) – Spread of the Cauchy distribution ($\gamma$). Default = 1

**mean**()
Compute the mean of this distribution.

Note that the mean of a Cauchy distribution is technically undefined.

**cdf**(*y*)
Cumulative probability of some data along this distribution

**log_prob**(*y*)
Compute the log probability of some data given this distribution

**mode**()
Compute the mode of this distribution

**prob**(*y*)
Compute the probability of some data given this distribution

**sample**(*n=1*)
Generate a random sample from this distribution

**class** probflow.distributions.**Deterministic**(*loc=0*)
Bases: *probflow.utils.base.BaseDistribution*

A deterministic distribution.

A deterministic distribution is a continuous distribution defined over all real numbers, and has one parameter:

- a location parameter (`loc` or $k_0$) which determines the mean of the distribution.

A random variable $x$ drawn from a deterministic distribution has probability of 1 at its location parameter value, and zero elsewhere:

$$p(x) = \begin{cases} 1, & \text{if } x = k_0 \\ 0, & \text{otherwise} \end{cases}$$

TODO: example image of the distribution

**Parameters** **loc** (int, float, ndarray, or Tensor) – Mean of the deterministic distribution ($k_0$). Default = 0

**cdf**(*y*)
Cumulative probability of some data along this distribution

**log_prob**(*y*)
Compute the log probability of some data given this distribution

**mean**()
Compute the mean of this distribution

Note that this uses the mode of distributions for which the mean is undefined (for example, a categorical distribution)

**mode**()
Compute the mode of this distribution

**prob**(*y*)
Compute the probability of some data given this distribution

**sample**(*n=1*)
Generate a random sample from this distribution

**class** probflow.distributions.**Dirichlet**(*concentration*)
Bases: *probflow.utils.base.BaseDistribution*

The Dirichlet distribution.

The Dirichlet distribution is a continuous distribution defined over the $k$-simplex, and has one vector of parameters:

- concentration parameters (`concentration` or $\boldsymbol{\alpha} \in \mathbb{R}^k_{>0}$), a vector of positive numbers which determine the relative likelihoods of different categories represented by the distribution.

A random variable (a vector) $\mathbf{x}$ drawn from a Dirichlet distribution

$$\mathbf{x} \sim \text{Dirichlet}(\boldsymbol{\alpha})$$

has probability

$$p(\mathbf{x}) = \frac{1}{\text{B}(\boldsymbol{\alpha})} \prod_{i=1}^{K} x_i^{\alpha_i - 1}$$

where B is the multivariate beta function.

TODO: example image of the distribution

**Parameters** **concentration** (ndarray, or Tensor) – Concentration parameter of the Dirichlet distribution ($\alpha$).

**cdf**(*y*)
Cumulative probability of some data along this distribution

**log_prob**(*y*)
Compute the log probability of some data given this distribution

**mean**()
Compute the mean of this distribution

Note that this uses the mode of distributions for which the mean is undefined (for example, a categorical distribution)

**mode**()
Compute the mode of this distribution

**prob**(*y*)
Compute the probability of some data given this distribution

**sample**(*n=1*)
Generate a random sample from this distribution

**class** probflow.distributions.**Gamma**(*concentration*, *rate*)
Bases: *probflow.utils.base.BaseDistribution*

The Gamma distribution.

The Gamma distribution is a continuous distribution defined over all positive real numbers, and has two parameters:

- a shape parameter (`shape` or $\alpha > 0$, a.k.a. "concentration"), and
- a rate parameter (`rate` or $\beta > 0$).

The ratio of $\frac{\alpha}{\beta}$ determines the mean of the distribution, and the ratio of $\frac{\alpha}{\beta^2}$ determines the variance.

A random variable $x$ drawn from a Gamma distribution

$$x \sim \text{Gamma}(\alpha, \beta)$$

has probability

$$p(x) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha - 1} \exp(-\beta x)$$

Where $\Gamma$ is the Gamma function.

TODO: example image of the distribution

**Parameters**

- **shape** (int, float, ndarray, or Tensor) – Shape parameter of the gamma distribution ($\alpha$).
- **rate** (int, float, ndarray, or Tensor) – Rate parameter of the gamma distribution ($\beta$).

**cdf**(*y*)
Cumulative probability of some data along this distribution

**log_prob**(*y*)
Compute the log probability of some data given this distribution

**mean**()
Compute the mean of this distribution

Note that this uses the mode of distributions for which the mean is undefined (for example, a categorical distribution)

**mode**()
Compute the mode of this distribution

**prob**(*y*)
Compute the probability of some data given this distribution

**sample**(*n=1*)
Generate a random sample from this distribution

**class** probflow.distributions.**HiddenMarkovModel**(*initial*, *transition*, *observation*, *steps*)
Bases: *probflow.utils.base.BaseDistribution*

A hidden Markov model distribution

TODO: docs

$$p(X_0) \text{initial probability}$$

TODO: example image of the distribution

**Parameters** **initial** (ndarray, or Tensor) – Concentration parameter of the Dirichlet distribution ($\alpha$).

**cdf**(*y*)
Cumulative probability of some data along this distribution

**log_prob**(*y*)
Compute the log probability of some data given this distribution

**mean**()
Compute the mean of this distribution

Note that this uses the mode of distributions for which the mean is undefined (for example, a categorical distribution)

**mode**()
Compute the mode of this distribution

**prob**(*y*)
Compute the probability of some data given this distribution

**sample**(*n=1*)
Generate a random sample from this distribution

**class** probflow.distributions.**InverseGamma**(*concentration*, *scale*)
Bases: *probflow.utils.base.BaseDistribution*

The Inverse-gamma distribution.

The Inverse-gamma distribution is a continuous distribution defined over all positive real numbers, and has two parameters:

- a shape parameter (`shape` or $\alpha > 0$, a.k.a. "concentration"), and
- a rate parameter (`rate` or $\beta > 0$, a.k.a. "scale").

The ratio of $\frac{\beta}{\alpha-1}$ determines the mean of the distribution, and for $\alpha > 2$, the variance is determined by:

$$\frac{\beta^2}{(\alpha-1)^2(\alpha-2)}$$

A random variable $x$ drawn from an Inverse-gamma distribution

$$x \sim \text{InvGamma}(\alpha, \beta)$$

has probability

$$p(x) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{-\alpha-1} \exp(-\frac{\beta}{x})$$

Where $\Gamma$ is the Gamma function.

TODO: example image of the distribution

**Parameters**

- **concentration** (int, float, `ndarray`, or Tensor) – Shape parameter of the inverse gamma distribution ($\alpha$).
- **scale** (int, float, `ndarray`, or Tensor) – Rate parameter of the inverse gamma distribution ($\beta$).

**cdf**(*y*)
Cumulative probability of some data along this distribution

**log_prob**(*y*)
Compute the log probability of some data given this distribution

**mean**()
Compute the mean of this distribution

Note that this uses the mode of distributions for which the mean is undefined (for example, a categorical distribution)

**mode**()
Compute the mode of this distribution

**prob**(*y*)
Compute the probability of some data given this distribution

**sample**(*n=1*)
Generate a random sample from this distribution

**class** probflow.distributions.**Mixture**(*distributions*, *logits=None*, *probs=None*)
Bases: *probflow.utils.base.BaseDistribution*

A mixture distribution.

TODO

TODO: example image of the distribution w/ 2 gaussians

**Parameters**

- **distributions** (*Distribution*) – Distributions to mix.
- **logits** (|Tensor|) – Logit probabilities of the mixture weights. Either this or *probs* must be specified.
- **probs** (|Tensor|) – Raw probabilities of the mixture weights. Either this or *probs* must be specified. Must sum to 1 along the last axis.

**cdf** (*y*)
Cumulative probability of some data along this distribution

**log_prob** (*y*)
Compute the log probability of some data given this distribution

**mean** ()
Compute the mean of this distribution

Note that this uses the mode of distributions for which the mean is undefined (for example, a categorical distribution)

**mode** ()
Compute the mode of this distribution

**prob** (*y*)
Compute the probability of some data given this distribution

**sample** (*n=1*)
Generate a random sample from this distribution

**class** probflow.distributions.**MultivariateNormal** (*loc*, *cov*)
Bases: *probflow.utils.base.BaseDistribution*

The multivariate Normal distribution.

The multivariate normal distribution is a continuous distribution in $d$-dimensional space, and has two parameters:

- a location vector (`loc` or $\boldsymbol{\mu} \in \mathbb{R}^d$) which determines the mean of the distribution, and
- a covariance matrix (`scale` or $\boldsymbol{\Sigma} \in \mathbb{R}^{d \times d}_{>0}$) which determines the spread and covariance of the distribution.

A random variable $\mathbf{x} \in \mathbb{R}^d$ drawn from a multivariate normal distribution

$$\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

has probability

$$p(\mathbf{x}) = (2\pi)^{-\frac{d}{2}} \det(\boldsymbol{\Sigma})^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)$$

TODO: example image of the distribution

**Parameters**

- **loc** (ndarray, or Tensor) – Mean of the multivariate normal distribution ($\boldsymbol{\mu}$).
- **cov** (ndarray, or Tensor) – Covariance matrix of the multivariate normal distribution ($\boldsymbol{\Sigma}$).

**cdf** (*y*)
Cumulative probability of some data along this distribution

**log_prob** (*y*)
Compute the log probability of some data given this distribution

**mean**()
Compute the mean of this distribution

Note that this uses the mode of distributions for which the mean is undefined (for example, a categorical distribution)

**mode**()
Compute the mode of this distribution

**prob**(*y*)
Compute the probability of some data given this distribution

**sample**(*n=1*)
Generate a random sample from this distribution

**class** probflow.distributions.**Normal**(*loc=0*, *scale=1*)
Bases: *probflow.utils.base.BaseDistribution*

The Normal distribution.

The normal distribution is a continuous distribution defined over all real numbers, and has two parameters:

- a location parameter (`loc` or $\mu$) which determines the mean of the distribution, and
- a scale parameter (`scale` or $\sigma > 0$) which determines the standard deviation of the distribution.

A random variable $x$ drawn from a normal distribution

$$x \sim \mathcal{N}(\mu, \sigma)$$

has probability

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

TODO: example image of the distribution

**Parameters**

- **loc** (int, float, `ndarray`, or Tensor) – Mean of the normal distribution ($\mu$). Default = 0
- **scale** (int, float, `ndarray`, or Tensor) – Standard deviation of the normal distribution ($\sigma$). Default = 1

**cdf**(*y*)
Cumulative probability of some data along this distribution

**log_prob**(*y*)
Compute the log probability of some data given this distribution

**mean**()
Compute the mean of this distribution

Note that this uses the mode of distributions for which the mean is undefined (for example, a categorical distribution)

**mode**()
Compute the mode of this distribution

**prob**(*y*)
Compute the probability of some data given this distribution

**sample**(*n=1*)
Generate a random sample from this distribution

**class** probflow.distributions.**OneHotCategorical**(*logits=None*, *probs=None*)
Bases: *probflow.utils.base.BaseDistribution*

The Categorical distribution, parameterized by categories-len vectors.

TODO: explain

TODO: example image of the distribution

TODO: logits vs probs

**Parameters**

- **logits** (int, float, ndarray, or Tensor) – Logit-transformed category probabilities
- **probs** (int, float, ndarray, or Tensor) –

**cdf**(*y*)
Cumulative probability of some data along this distribution

**log_prob**(*y*)
Compute the log probability of some data given this distribution

**mean**()
Compute the mean of this distribution

Note that this uses the mode of distributions for which the mean is undefined (for example, a categorical distribution)

**mode**()
Compute the mode of this distribution

**prob**(*y*)
Compute the probability of some data given this distribution

**sample**(*n=1*)
Generate a random sample from this distribution

**class** probflow.distributions.**Poisson**(*rate*)
Bases: *probflow.utils.base.BaseDistribution*

The Poisson distribution.

The Poisson distribution is a discrete distribution defined over all non-negativve real integers, and has one parameter:

- a rate parameter (`rate` or $\lambda$) which determines the mean of the distribution.

A random variable $x$ drawn from a Poisson distribution

$$x \sim \text{Poisson}(\lambda)$$

has probability

$$p(x) = \frac{\lambda^x e^{-\lambda}}{x!}$$

TODO: example image of the distribution

**Parameters** **rate** (int, float, ndarray, or Tensor) – Rate parameter of the Poisson distribution ($\lambda$).

**cdf**(*y*)
Cumulative probability of some data along this distribution

**log_prob**(*y*)
Compute the log probability of some data given this distribution

**mean**()
Compute the mean of this distribution

Note that this uses the mode of distributions for which the mean is undefined (for example, a categorical distribution)

**mode**()
Compute the mode of this distribution

**prob**(*y*)
Compute the probability of some data given this distribution

**sample**(*n=1*)
Generate a random sample from this distribution

**class** probflow.distributions.**StudentT**(*df=1*, *loc=0*, *scale=1*)
Bases: *probflow.utils.base.BaseDistribution*

The Student-t distribution.

The Student's t-distribution is a continuous distribution defined over all real numbers, and has three parameters:

- a degrees of freedom parameter (`df` or $\nu > 0$), which determines how many degrees of freedom the distribution has,
- a location parameter (`loc` or $\mu$) which determines the mean of the distribution, and
- a scale parameter (`scale` or $\sigma > 0$) which determines the standard deviation of the distribution.

A random variable $x$ drawn from a Student's t-distribution

$$x \sim \text{StudentT}(\nu, \mu, \sigma)$$

has probability

$$p(x) = \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu x}\Gamma(\frac{\nu}{2})}\left(1 + \frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}}$$

Where $\Gamma$ is the Gamma function.

TODO: example image of the distribution

**Parameters**

- **df** (int, float, `ndarray`, or Tensor) – Degrees of freedom of the t-distribution ($\nu$). Default = 1
- **loc** (int, float, `ndarray`, or Tensor) – Median of the t-distribution ($\mu$). Default = 0
- **scale** (int, float, `ndarray`, or Tensor) – Spread of the t-distribution ($\sigma$). Default = 1

**mean**()
Compute the mean of this distribution.

Note that the mean of a StudentT distribution is technically undefined when df=1.

**cdf**(*y*)
Cumulative probability of some data along this distribution

**log_prob**(*y*)
Compute the log probability of some data given this distribution

**mode**()
Compute the mode of this distribution

**prob**(*y*)
Compute the probability of some data given this distribution

**sample**(*n=1*)
Generate a random sample from this distribution

## 3.2 Parameters

Parameters are values which characterize the behavior of a model. When fitting a model, we want to find the values of the parameters which best allow the model to explain the data. However, with Bayesian modeling we want not only to find the single *best* value for each parameter, but a probability distribution which describes how likely any given value of a parameter is to be the best or true value.

Parameters have both priors (probability distributions which describe how likely we think different values for the parameter are *before* taking into consideration the current data), and posteriors (probability distributions which describe how likely we think different values for the parameter are *after* taking into consideration the current data). The prior is set to a specific distribution before fitting the model. While the *type* of distribution used for the posterior is set before fitting the model, the shape of that distribution (the value of the parameters which define the distribution) is optimized while fitting the model. See the *Mathematical Details* section for more info.

The `Parameter` class can be used to create any probabilistic parameter.

For convenience, ProbFlow also includes some classes which are special cases of a `Parameter`:

- `ScaleParameter` - standard deviation parameter
- `CategoricalParameter` - categorical parameter
- `DirichletParameter` - parameter with a Dirichlet posterior
- `BoundedParameter` - parameter which is bounded between 0 and 1
- `PositiveParameter` - parameter which is always greater than 0
- `DeterministicParameter` - a non-probabilistic parameter
- `MultivariateNormalParameter` - parameter with a multivariate Normal posterior
- `CenteredParameter` - parameter array with a mean of 0

See the *user guide* for more information on Parameters.

---

**class** probflow.parameters.**BoundedParameter**(*shape=1*, *posterior=<class 'probflow.distributions.normal.Normal'>*, *prior=<probflow.distributions.normal.Normal object>*, *transform=None*, *initializer={'loc': <function xavier>, 'scale': <function scale_xavier>}*, *var_transform={'loc': None, 'scale': <function softplus>}*, *min: float = 0.0*, *max: float = 1.0*, *name='BoundedParameter'*)

Bases: `probflow.parameters.parameter.Parameter`

A parameter bounded on either side

This is a convenience class for creating a parameter $\beta$ bounded on both sides. It uses a logit-normal posterior distribution:

$$\text{Logit}(\beta) = \log\left(\frac{\beta}{1-\beta}\right) \sim \text{Normal}(\mu, \sigma)$$

**Parameters**

- **shape** (*int or List[int]*) – Shape of the array containing the parameters. Default = 1
- **posterior** (*Distribution* class) – Probability distribution class to use to approximate the posterior. Default = *Normal*
- **prior** (*Distribution* object) – Prior probability distribution function which has been instantiated with parameters. Default = *Normal* (0, 1)
- **transform** (*callable*) – Transform to apply to the random variable. Default is to use a sigmoid transform.
- **initializer** (*Dict[str, callable]*) – Initializer functions to use for each variable of the variational posterior distribution. Keys correspond to variable names (arguments to the distribution), and values contain functions to initialize those variables given `shape` as the single argument.
- **var_transform** (*Dict[str, callable]*) – Transform to apply to each variable of the variational posterior.
- **min** (*float*) – Minimum value the parameter can take. Default = 0.
- **max** (*float*) – Maximum value the parameter can take. Default = 1.
- **name** (*str*) – Name of the parameter(s). Default = `'BoundedParameter'`

### Examples

TODO

**bayesian_update**()
Update priors to match the current posterior

**kl_loss**()
Compute the sum of the Kullback–Leibler divergences between this parameter's priors and its variational posteriors.

**property n_parameters**
Get the number of independent parameters

**property n_variables**
Get the number of underlying variables

**property posterior**
This Parameter's variational posterior distribution

**posterior_ci**(*ci: float = 0.95*, *n: int = 10000*)
Posterior confidence intervals

**Parameters**

- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95

- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals. Default = 10,000

**Returns**

- **lb** (*float or |ndarray|*) – Lower bound(s) of the confidence interval(s)
- **ub** (*float or |ndarray|*) – Upper bound(s) of the confidence interval(s)

**posterior_mean**()
Get the mean of the posterior distribution(s)

**posterior_plot**(*n: int = 10000*, *style: str = 'fill'*, *bins: Union[int, list, numpy.ndarray] = 20*, *ci: float = 0.0*, *bw: float = 0.075*, *alpha: float = 0.4*, *color=None*, ***kwargs*)
Plot distribution of samples from the posterior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each posterior distribution for estimating the density. Default = 10000
- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram
- **bins** (int or list or `ndarray`) – Number of bins to use for the posterior density histogram (if `style='hist'`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075
- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle
- **kwargs** – Additional keyword arguments are passed to `utils.plotting.plot_dist()`

**posterior_sample**(*n: int = 1*)
Sample from the posterior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the posterior distribution. Default = 1

**Returns** Samples from the parameter's posterior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** `ndarray`

**prior_plot**(*n: int = 10000*, *style: str = 'fill'*, *bins: Union[int, list, numpy.ndarray] = 20*, *ci: float = 0.0*, *bw: float = 0.075*, *alpha: float = 0.4*, *color=None*)
Plot distribution of samples from the prior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each prior distribution for estimating the density. Default = 1000

- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram
- **bins** (int or list or `ndarray`) – Number of bins to use for the prior density histogram (if `style='hist'`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075
- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle

**prior_sample**(*n: int = 1*)

Sample from the prior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the prior distribution. Default = 1

**Returns** Samples from the parameter prior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** `ndarray`

**property trainable_variables**

Get a list of trainable variables from the backend

**property variables**

Variables after applying their respective transformations

**class** probflow.parameters.**CategoricalParameter**(*k: int = 2, shape: Union[int, List[int]] = [], posterior=<class 'probflow.distributions.categorical.Categorical'>, prior=None, transform=None, initializer={'probs': <function xavier>}, var_transform={'probs': <function additive_logistic_transform>}, name='CategoricalParameter'*)

Bases: `probflow.parameters.parameter.Parameter`

Categorical parameter.

This is a convenience class for creating a categorical parameter $\beta$ with a Categorical posterior:

$$\beta \sim \text{Categorical}(\theta)$$

By default, a uniform prior is used.

TODO: explain that a sample is an int in [0, k-1]

**Parameters**

- **k** (*int > 2*) – Number of categories.

- **shape** (*int or List[int]*) – Shape of the array containing the parameters. Default = 1
- **posterior** (*Distribution* class) – Probability distribution class to use to approximate the posterior. Default = *Categorical*
- **prior** (*Distribution* object) – Prior probability distribution function which has been instantiated with parameters. Default = *Categorical* (1/k)
- **transform** (*callable*) – Transform to apply to the random variable. Default is to use no transform.
- **initializer** (*Dict[str, callable]*) – Initializer functions to use for each variable of the variational posterior distribution. Keys correspond to variable names (arguments to the distribution), and values contain functions to initialize those variables given `shape` as the single argument.
- **var_transform** (*Dict[str, callable]*) – Transform to apply to each variable of the variational posterior.
- **name** (*str*) – Name of the parameter(s). Default = `'CategoricalParameter'`

### Examples

TODO: creating variable

**bayesian_update**()
Update priors to match the current posterior

**kl_loss**()
Compute the sum of the Kullback–Leibler divergences between this parameter's priors and its variational posteriors.

**property n_parameters**
Get the number of independent parameters

**property n_variables**
Get the number of underlying variables

**property posterior**
This Parameter's variational posterior distribution

**posterior_ci**(*ci: float = 0.95*, *n: int = 10000*)
Posterior confidence intervals

**Parameters**

- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95
- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals. Default = 10,000

**Returns**

- **lb** (*float or |ndarray|*) – Lower bound(s) of the confidence interval(s)
- **ub** (*float or |ndarray|*) – Upper bound(s) of the confidence interval(s)

**posterior_mean**()
Get the mean of the posterior distribution(s)

**posterior_plot**(*n: int = 10000*, *style: str = 'fill'*, *bins: Union[int, list, numpy.ndarray] = 20*, *ci: float = 0.0*, *bw: float = 0.075*, *alpha: float = 0.4*, *color=None*, ***kwargs*)

Plot distribution of samples from the posterior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each posterior distribution for estimating the density. Default = 10000
- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram
- **bins** (int or list or `ndarray`) – Number of bins to use for the posterior density histogram (if `style='hist'`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075
- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle
- **kwargs** – Additional keyword arguments are passed to `utils.plotting.plot_dist()`

**posterior_sample**(*n: int = 1*)

Sample from the posterior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the posterior distribution. Default = 1

**Returns** Samples from the parameter's posterior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** `ndarray`

**prior_plot**(*n: int = 10000*, *style: str = 'fill'*, *bins: Union[int, list, numpy.ndarray] = 20*, *ci: float = 0.0*, *bw: float = 0.075*, *alpha: float = 0.4*, *color=None*)

Plot distribution of samples from the prior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each prior distribution for estimating the density. Default = 1000
- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram
- **bins** (int or list or `ndarray`) – Number of bins to use for the prior density histogram (if `style='hist'`), or a list or vector of bin edges.

- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075
- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle

**prior_sample**(*n: int = 1*)

Sample from the prior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the prior distribution. Default = 1

**Returns** Samples from the parameter prior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** ndarray

**property trainable_variables**

Get a list of trainable variables from the backend

**property variables**

Variables after applying their respective transformations

**class** probflow.parameters.**CenteredParameter**(*shape: Union[int, List[int]]*, *center_by: str = 'all'*, *name='CenteredParameter'*)

Bases: probflow.parameters.parameter.Parameter

A vector of parameters centered at 0.

Uses a QR decomposition to transform a vector of $K-1$ unconstrained parameters into a vector of $K$ variables centered at zero (i.e. the mean of the elements in the vector is 0). It starts with a $K \times K$ matrix $A$ which has 1-1`s along the bottom - except for the bottom-right element which is 0:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \dots & 1 & 0 \\ -1 & -1 & \dots & -1 & 0 \end{bmatrix}$$

The $QR$ decomposition is performed on this matrix such that

$$\mathbf{A} = \mathbf{QR}$$

A vector of $K-1$ unconstrained variables $\mathbf{u}$ is then transformed into a vector $\mathbf{v}$ of $K$ centered variables by

$$\mathbf{v} = \mathbf{Q}_{1:K,1:K-1}\mathbf{u}$$

The prior on the untransformed variables is

$$\mathbf{u} \sim \text{Normal}(0, \frac{1}{\sqrt{1-\frac{1}{K}}})$$

Such that the effective prior on the transformed parameters works out to be

$$\mathbf{v} \sim \text{Normal}(0, 1)$$

**Prior is fixed!**

Note that the prior on the parameters is fixed at Normal$(0, 1)$. This is because the true prior is being placed on the untransformed parameters (see above).

**Parameters**

- **d** (*int or list*) – Length of the parameter vector, or shape of the parameter matrix.
- **center_by** (*str {'all', 'column', 'row'}*) – If `all` (the default), the sum of all parameters in the resulting vector or matrix will be 0. If `column`, the sum of each column will be 0 (but the sum across rows will not necessarily be 0). If `row`, the sum of each row will be 0 (but the sum across columns will not necessarily be 0).
- **name** (*str*) – Name of the parameter(s). Default = `'CenteredParameter'`

### Examples

TODO

**bayesian_update**()
Update priors to match the current posterior

**kl_loss**()
Compute the sum of the Kullback–Leibler divergences between this parameter's priors and its variational posteriors.

**property n_parameters**
Get the number of independent parameters

**property n_variables**
Get the number of underlying variables

**property posterior**
This Parameter's variational posterior distribution

**posterior_ci**(*ci: float = 0.95*, *n: int = 10000*)
Posterior confidence intervals

**Parameters**

- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95
- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals. Default = 10,000

**Returns**

- **lb** (*float or |ndarray|*) – Lower bound(s) of the confidence interval(s)
- **ub** (*float or |ndarray|*) – Upper bound(s) of the confidence interval(s)

**posterior_mean**()
Get the mean of the posterior distribution(s)

**posterior_plot**(*n: int = 10000*, *style: str = 'fill'*, *bins: Union[int, list, numpy.ndarray] = 20*, *ci: float = 0.0*, *bw: float = 0.075*, *alpha: float = 0.4*, *color=None*, ***kwargs*)
Plot distribution of samples from the posterior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each posterior distribution for estimating the density. Default = 10000
- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram
- **bins** (int or list or `ndarray`) – Number of bins to use for the posterior density histogram (if `style='hist'`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075
- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle
- **kwargs** – Additional keyword arguments are passed to `utils.plotting.plot_dist()`

**posterior_sample**(*n: int = 1*)

Sample from the posterior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the posterior distribution. Default = 1

**Returns** Samples from the parameter's posterior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** `ndarray`

**prior_plot**(*n: int = 10000, style: str = 'fill', bins: Union[int, list, numpy.ndarray] = 20, ci: float = 0.0, bw: float = 0.075, alpha: float = 0.4, color=None*)

Plot distribution of samples from the prior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each prior distribution for estimating the density. Default = 1000
- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram
- **bins** (int or list or `ndarray`) – Number of bins to use for the prior density histogram (if `style='hist'`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075
- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram

- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle

**prior_sample**(*n: int = 1*)

Sample from the prior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the prior distribution. Default = 1

**Returns** Samples from the parameter prior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** `ndarray`

**property trainable_variables**

Get a list of trainable variables from the backend

**property variables**

Variables after applying their respective transformations

**class** probflow.parameters.**DeterministicParameter**(*shape=1*, *posterior=<class 'probflow.distributions.deterministic.Deterministic'>*, *prior=<probflow.distributions.normal.Normal object>*, *transform=None*, *initializer={'loc': <function xavier>}*, *var_transform={'loc': None}*, *name='DeterministicParameter'*)

Bases: `probflow.parameters.parameter.Parameter`

A parameter which takes only a single value (i.e., the posterior is a single point value, not a probability distribution).

**Parameters**

- **shape** (*int or List[int]*) – Shape of the array containing the parameters. Default = `1`
- **posterior** (*Distribution* class) – Probability distribution class to use to approximate the posterior. Default = *Deterministic*
- **prior** (*Distribution* object) – Prior probability distribution function which has been instantiated with parameters. Default = *Normal* `(0, 1)`
- **transform** (*callable*) – Transform to apply to the random variable. Default is to use no transformation.
- **initializer** (*Dict[str, callable]*) – Initializer functions to use for each variable of the variational posterior distribution. Keys correspond to variable names (arguments to the distribution), and values contain functions to initialize those variables given `shape` as the single argument.
- **var_transform** (*Dict[str, callable]*) – Transform to apply to each variable of the variational posterior.
- **name** (*str*) – Name of the parameter(s). Default = `'PositiveParameter'`

### Examples

TODO

**bayesian_update**()
Update priors to match the current posterior

**kl_loss**()
Compute the sum of the Kullback–Leibler divergences between this parameter's priors and its variational posteriors.

**property n_parameters**
Get the number of independent parameters

**property n_variables**
Get the number of underlying variables

**property posterior**
This Parameter's variational posterior distribution

**posterior_ci**(*ci: float = 0.95*, *n: int = 10000*)
Posterior confidence intervals

**Parameters**

- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95
- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals. Default = 10,000

**Returns**

- **lb** (*float or |ndarray|*) – Lower bound(s) of the confidence interval(s)
- **ub** (*float or |ndarray|*) – Upper bound(s) of the confidence interval(s)

**posterior_mean**()
Get the mean of the posterior distribution(s)

**posterior_plot**(*n: int = 10000*, *style: str = 'fill'*, *bins: Union[int, list, numpy.ndarray] = 20*, *ci: float = 0.0*, *bw: float = 0.075*, *alpha: float = 0.4*, *color=None*, ***kwargs*)
Plot distribution of samples from the posterior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each posterior distribution for estimating the density. Default = 10000
- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram
- **bins** (int or list or `ndarray`) – Number of bins to use for the posterior density histogram (if `style='hist'`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075

- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle
- **kwargs** – Additional keyword arguments are passed to `utils.plotting.plot_dist()`

**posterior_sample**(*n: int = 1*)

Sample from the posterior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the posterior distribution. Default = 1

**Returns** Samples from the parameter's posterior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** ndarray

**prior_plot**(*n: int = 10000*, *style: str = 'fill'*, *bins: Union[int, list, numpy.ndarray] = 20*, *ci: float = 0.0*, *bw: float = 0.075*, *alpha: float = 0.4*, *color=None*)

Plot distribution of samples from the prior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each prior distribution for estimating the density. Default = 1000
- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram
- **bins** (int or list or ndarray) – Number of bins to use for the prior density histogram (if `style='hist'`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075
- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle

**prior_sample**(*n: int = 1*)

Sample from the prior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the prior distribution. Default = 1

**Returns** Samples from the parameter prior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** ndarray

**property trainable_variables**

Get a list of trainable variables from the backend

**property variables**
Variables after applying their respective transformations

**class** probflow.parameters.**DirichletParameter**(*k: int = 2*, *shape: Union[int, List[int]] = []*, *posterior=<class 'probflow.distributions.dirichlet.Dirichlet'>*, *prior=None*, *transform=None*, *initializer={'concentration': <function pos_xavier>}*, *var_transform={'concentration': <function softplus>}*, *name='DirichletParameter'*)

Bases: probflow.parameters.parameter.Parameter

Dirichlet parameter.

This is a convenience class for creating a parameter $\theta$ with a Dirichlet posterior:

$$\theta \sim \text{Dirichlet}(\alpha)$$

By default, a uniform Dirichlet prior is used:

$$\theta \sim \text{Dirichlet}_K(\mathbf{1}/K)$$

TODO: explain that a sample is a categorical prob dist (as compared to CategoricalParameter, where a sample is a single value)

**Parameters**

- **k** (*int > 2*) – Number of categories.
- **shape** (*int or List[int]*) – Shape of the array containing the parameters. Default = 1
- **posterior** (*Distribution* class) – Probability distribution class to use to approximate the posterior. Default = *Dirichlet*
- **prior** (*Distribution* object) – Prior probability distribution function which has been instantiated with parameters. Default = *Dirichlet* (1)
- **transform** (*callable*) – Transform to apply to the random variable. Default is to use no transform.
- **initializer** (*Dict[str, callable]*) – Initializer functions to use for each variable of the variational posterior distribution. Keys correspond to variable names (arguments to the distribution), and values contain functions to initialize those variables given shape as the single argument.
- **var_transform** (*Dict[str, callable]*) – Transform to apply to each variable of the variational posterior.
- **name** (*str*) – Name of the parameter(s). Default = 'DirichletParameter'

### Examples

TODO: creating variable

**bayesian_update**()
Update priors to match the current posterior

**kl_loss**()
Compute the sum of the Kullback–Leibler divergences between this parameter's priors and its variational posteriors.

**property n_parameters**
Get the number of independent parameters

**property n_variables**
Get the number of underlying variables

**property posterior**
This Parameter's variational posterior distribution

**posterior_ci**(*ci: float = 0.95*, *n: int = 10000*)
Posterior confidence intervals

**Parameters**

- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95
- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals. Default = 10,000

**Returns**

- **lb** (*float or |ndarray|*) – Lower bound(s) of the confidence interval(s)
- **ub** (*float or |ndarray|*) – Upper bound(s) of the confidence interval(s)

**posterior_mean**()
Get the mean of the posterior distribution(s)

**posterior_plot**(*n: int = 10000*, *style: str = 'fill'*, *bins: Union[int, list, numpy.ndarray] = 20*, *ci: float = 0.0*, *bw: float = 0.075*, *alpha: float = 0.4*, *color=None*, ***kwargs*)
Plot distribution of samples from the posterior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each posterior distribution for estimating the density. Default = 10000
- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram
- **bins** (int or list or `ndarray`) – Number of bins to use for the posterior density histogram (if `style='hist'`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075

- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle
- **kwargs** – Additional keyword arguments are passed to `utils.plotting.plot_dist()`

**posterior_sample**(*n: int = 1*)

Sample from the posterior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the posterior distribution. Default = 1

**Returns** Samples from the parameter's posterior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** ndarray

**prior_plot**(*n: int = 10000*, *style: str = 'fill'*, *bins: Union[int, list, numpy.ndarray] = 20*, *ci: float = 0.0*, *bw: float = 0.075*, *alpha: float = 0.4*, *color=None*)

Plot distribution of samples from the prior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each prior distribution for estimating the density. Default = 1000
- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram
- **bins** (int or list or ndarray) – Number of bins to use for the prior density histogram (if `style='hist'`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075
- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle

**prior_sample**(*n: int = 1*)

Sample from the prior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the prior distribution. Default = 1

**Returns** Samples from the parameter prior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** ndarray

**property trainable_variables**

Get a list of trainable variables from the backend

**property variables**
Variables after applying their respective transformations

**class** probflow.parameters.**MultivariateNormalParameter**(*d: int = 1*, *prior=None*, *expand_dims: int = -1*, *name='MultivariateNormalParameter'*)

Bases: probflow.parameters.parameter.Parameter

A parameter with a multivariate normal posterior, with full covariance.

TODO: uses the log-Cholesky parameterization (Pinheiro & Bates, 1996).

TODO: support shape?

**Parameters**

- **d** (*int*) – Number of dimensions
- **prior** (*Distribution* object) – Prior probability distribution function which has been instantiated with parameters. Default = *MultivariateNormal* (0, I)
- **expand_dims** (*int or None*) – Dimension to expand output samples along.
- **name** (*str*) – Name of the parameter(s). Default = 'MultivariateNormalParameter'

### Examples

TODO

### References


- Jose C. Pinheiro & Douglas M. Bates. Unconstrained Parameterizations for Variance-Covariance Matrices *Statistics and Computing*, 1996.


**bayesian_update**()
Update priors to match the current posterior

**kl_loss**()
Compute the sum of the Kullback–Leibler divergences between this parameter's priors and its variational posteriors.

**property n_parameters**
Get the number of independent parameters

**property n_variables**
Get the number of underlying variables

**property posterior**
This Parameter's variational posterior distribution

**posterior_ci**(*ci: float = 0.95*, *n: int = 10000*)
Posterior confidence intervals

**Parameters**

- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95
- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals. Default = 10,000

**Returns**

- **lb** (*float or |ndarray|*) – Lower bound(s) of the confidence interval(s)
- **ub** (*float or |ndarray|*) – Upper bound(s) of the confidence interval(s)

**posterior_mean**()
Get the mean of the posterior distribution(s)

**posterior_plot**(*n: int = 10000*, *style: str = 'fill'*, *bins: Union[int, list, numpy.ndarray] = 20*, *ci: float = 0.0*, *bw: float = 0.075*, *alpha: float = 0.4*, *color=None*, ***kwargs*)
Plot distribution of samples from the posterior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each posterior distribution for estimating the density. Default = 10000
- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram
- **bins** (int or list or `ndarray`) – Number of bins to use for the posterior density histogram (if `style='hist'`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075
- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle
- **kwargs** – Additional keyword arguments are passed to `utils.plotting.plot_dist()`

**posterior_sample**(*n: int = 1*)
Sample from the posterior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the posterior distribution. Default = 1

**Returns** Samples from the parameter's posterior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** `ndarray`

**prior_plot**(*n: int = 10000*, *style: str = 'fill'*, *bins: Union[int, list, numpy.ndarray] = 20*, *ci: float = 0.0*, *bw: float = 0.075*, *alpha: float = 0.4*, *color=None*)
Plot distribution of samples from the prior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each prior distribution for estimating the density. Default = 1000
- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)

- `'line'` - line density plot
- `'hist'` - histogram

- **bins** (int or list or `ndarray`) – Number of bins to use for the prior density histogram (if `style='hist'`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075
- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle

**prior_sample**(*n: int = 1*)

Sample from the prior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the prior distribution. Default = 1

**Returns** Samples from the parameter prior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** `ndarray`

**property trainable_variables**

Get a list of trainable variables from the backend

**property variables**

Variables after applying their respective transformations

**class** probflow.parameters.**Parameter**(*shape: Union[int, List[int]] = 1*, *posterior: Type[probflow.utils.base.BaseDistribution] = <class 'probflow.distributions.normal.Normal'>*, *prior: probflow.utils.base.BaseDistribution = <probflow.distributions.normal.Normal object>*, *transform: Optional[Callable] = None*, *initializer: Dict[str, Callable] = {'loc': <function xavier>, 'scale': <function scale_xavier>}*, *var_transform: Dict[str, Callable] = {'loc': None, 'scale': <function softplus>}*, *name: str = 'Parameter'*)

Bases: *probflow.utils.base.BaseParameter*

Probabilistic parameter(s).

A probabilistic parameter $\beta$. The default posterior distribution is the *Normal* distribution, and the default prior is a *Normal* distribution with a mean of 0 and a standard deviation of 1.

The prior for a *Parameter* can be set to any *Distribution* object (via the `prior` argument), and the type of distribution to use for the posterior can be set to any *Distribution* class (using the `posterior` argument).

The parameter can be given a specific name using the `name` argument. This makes it easier to access specific parameters after fitting the model (e.g. in order to view the posterior distribution).

The number of independent parameters represented by this *Parameter* object can be set using the `shape` argument. For example, to create a vector of 5 parameters, set `shape=5`, or to create a 20x7 matrix of parameters set `shape=[20,7]`.

**Parameters**

- **shape** (*int or List[int]*) – Shape of the array containing the parameters. Default = 1
- **posterior** (*Distribution* class) – Probability distribution class to use to approximate the posterior. Default = *Normal*
- **prior** (*Distribution* object) – Prior probability distribution function which has been instantiated with parameters. Default = *Normal* (0,1)
- **transform** (*callable*) – Transform to apply to the random variable. For example, to create a parameter with an inverse gamma posterior, use `posterior``=:class:`.Gamma` and `transform = lambda x: tf.reciprocal(x)` Default is to use no transform.
- **initializer** (*Dict[str, callable] or Dict[str, int] or Dict[str, float]*) – Initializer functions to use for each variable of the variational posterior distribution. Keys correspond to variable names (arguments to the distribution), and values contain functions to initialize those variables given `shape` as the single argument. Or, keys can be a float or an int, in which case all elements will be initialized to that value.
- **var_transform** (*Dict[str, callable]*) – Transform to apply to each variable of the variational posterior. For example to transform the standard deviation parameter from untransformed space to transformed, positive, space, use `initializer={'scale': tf.random.randn}` and `var_transform={'scale': tf.nn.softplus}`
- **name** (*str*) – Name of the parameter(s). Default = `'Parameter'`

**property n_parameters**
Get the number of independent parameters

**property n_variables**
Get the number of underlying variables

**property trainable_variables**
Get a list of trainable variables from the backend

**property variables**
Variables after applying their respective transformations

**property posterior**
This Parameter's variational posterior distribution

**kl_loss**()
Compute the sum of the Kullback–Leibler divergences between this parameter's priors and its variational posteriors.

**bayesian_update**()
Update priors to match the current posterior

**posterior_mean**()
Get the mean of the posterior distribution(s)

**posterior_sample**(*n: int = 1*)
Sample from the posterior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the posterior distribution. Default = 1

**Returns** Samples from the parameter's posterior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** `ndarray`

**prior_sample**(*n: int = 1*)
Sample from the prior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the prior distribution. Default = 1

**Returns** Samples from the parameter prior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** `ndarray`

**posterior_ci**(*ci: float = 0.95*, *n: int = 10000*)
Posterior confidence intervals

**Parameters**

- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95
- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals. Default = 10,000

**Returns**

- **lb** (*float or |ndarray|*) – Lower bound(s) of the confidence interval(s)
- **ub** (*float or |ndarray|*) – Upper bound(s) of the confidence interval(s)

**posterior_plot**(*n: int = 10000*, *style: str = 'fill'*, *bins: Union[int, list, numpy.ndarray] = 20*, *ci: float = 0.0*, *bw: float = 0.075*, *alpha: float = 0.4*, *color=None*, ***kwargs*)
Plot distribution of samples from the posterior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each posterior distribution for estimating the density. Default = 10000
- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram
- **bins** (int or list or `ndarray`) – Number of bins to use for the posterior density histogram (if `style='hist'`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075
- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle
- **kwargs** – Additional keyword arguments are passed to `utils.plotting.plot_dist()`

**prior_plot**(*n: int = 10000*, *style: str = 'fill'*, *bins: Union[int, list, numpy.ndarray] = 20*, *ci: float = 0.0*, *bw: float = 0.075*, *alpha: float = 0.4*, *color=None*)
Plot distribution of samples from the prior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each prior distribution for estimating the density. Default = 1000
- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram
- **bins** (int or list or `ndarray`) – Number of bins to use for the prior density histogram (if `style='hist'`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075
- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle

**class** probflow.parameters.**PositiveParameter**(*shape=1*, *posterior=<class 'probflow.distributions.normal.Normal'>*, *prior=<probflow.distributions.normal.Normal object>*, *transform=<function softplus>*, *initializer={'loc': <function xavier>, 'scale': <function scale_xavier>}*, *var_transform={'loc': None, 'scale': <function softplus>}*, *name='PositiveParameter'*)

Bases: `probflow.parameters.parameter.Parameter`

A parameter which takes only positive values.

This is a convenience class for creating a parameter $\beta$ which can only take positive values. It uses a normal variational posterior distribution and a softplus transform:

$$\log(1 + \exp(\beta)) \sim \text{Normal}(\mu, \sigma)$$

**Parameters**

- **shape** (*int or List[int]*) – Shape of the array containing the parameters. Default = `1`
- **posterior** (*Distribution* class) – Probability distribution class to use to approximate the posterior. Default = *Normal*
- **prior** (*Distribution* object) – Prior probability distribution function which has been instantiated with parameters. Default = *Normal* `(0, 1)`
- **transform** (*callable*) – Transform to apply to the random variable. Default is to use a softplus transform.
- **initializer** (*Dict[str, callable]*) – Initializer functions to use for each variable of the variational posterior distribution. Keys correspond to variable names (arguments to the distribution), and values contain functions to initialize those variables given `shape` as the single argument.

- **var_transform** (*Dict[str, callable]*) – Transform to apply to each variable of the variational posterior.
- **min** (*float*) – Minimum value the parameter can take. Default = 0.
- **max** (*float*) – Maximum value the parameter can take. Default = 1.
- **name** (*str*) – Name of the parameter(s). Default = `'PositiveParameter'`

### Examples

TODO

**bayesian_update**()
Update priors to match the current posterior

**kl_loss**()
Compute the sum of the Kullback–Leibler divergences between this parameter's priors and its variational posteriors.

**property n_parameters**
Get the number of independent parameters

**property n_variables**
Get the number of underlying variables

**property posterior**
This Parameter's variational posterior distribution

**posterior_ci**(*ci: float = 0.95*, *n: int = 10000*)
Posterior confidence intervals

**Parameters**

- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95
- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals. Default = 10,000

**Returns**

- **lb** (*float or |ndarray|*) – Lower bound(s) of the confidence interval(s)
- **ub** (*float or |ndarray|*) – Upper bound(s) of the confidence interval(s)

**posterior_mean**()
Get the mean of the posterior distribution(s)

**posterior_plot**(*n: int = 10000*, *style: str = 'fill'*, *bins: Union[int, list, numpy.ndarray] = 20*, *ci: float = 0.0*, *bw: float = 0.075*, *alpha: float = 0.4*, *color=None*, ***kwargs*)
Plot distribution of samples from the posterior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each posterior distribution for estimating the density. Default = 10000
- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram

- **bins** (int or list or ndarray) – Number of bins to use for the posterior density histogram (if `style='hist'`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075
- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle
- **kwargs** – Additional keyword arguments are passed to `utils.plotting.plot_dist()`

**posterior_sample**(*n: int = 1*)

Sample from the posterior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the posterior distribution. Default = 1

**Returns** Samples from the parameter's posterior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** ndarray

**prior_plot**(*n: int = 10000*, *style: str = 'fill'*, *bins: Union[int, list, numpy.ndarray] = 20*, *ci: float = 0.0*, *bw: float = 0.075*, *alpha: float = 0.4*, *color=None*)

Plot distribution of samples from the prior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each prior distribution for estimating the density. Default = 1000
- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram
- **bins** (int or list or ndarray) – Number of bins to use for the prior density histogram (if `style='hist'`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075
- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle

**prior_sample**(*n: int = 1*)

Sample from the prior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the prior distribution. Default = 1

**Returns** Samples from the parameter prior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** `ndarray`

**property trainable_variables**
Get a list of trainable variables from the backend

**property variables**
Variables after applying their respective transformations

**class** probflow.parameters.**ScaleParameter**(*shape=1*, *posterior=<class 'probflow.distributions.gamma.Gamma'>*, *prior=<probflow.distributions.gamma.Gamma object>*, *transform=<function ScaleParameter.<lambda>>*, *initializer={'concentration': <function full_of.<locals>.init>, 'rate': <function full_of.<locals>.init>}*, *var_transform={'concentration': <function exp>, 'rate': <function exp>}*, *name='ScaleParameter'*)

Bases: `probflow.parameters.parameter.Parameter`

Standard deviation parameter.

This is a convenience class for creating a standard deviation parameter ($\sigma$). It is created by first constructing a variance parameter ($\sigma^2$) which uses an inverse gamma distribution as the variational posterior.

$$\frac{1}{\sigma^2} \sim \text{Gamma}(\alpha, \beta)$$

Then the variance is transformed into the standard deviation:

$$\sigma = \sqrt{\sigma^2}$$

By default, an inverse gamma prior is used:

$$\frac{1}{\sigma^2} \sim \text{Gamma}(5, 5)$$

**Parameters**

- **shape** (*int or List[int]*) – Shape of the array containing the parameters. Default = `1`
- **posterior** (*Distribution* class) – Probability distribution class to use to approximate the posterior. Default = *Gamma*
- **prior** (*Distribution* object or `None`) – Prior probability distribution function which has been instantiated with parameters, or `None` for a uniform prior. Default = `None`
- **transform** (*callable*) – Transform to apply to the random variable. Default is to use an inverse square root transform (`sqrt(1/x)`)
- **initializer** (*Dict[str, callable]*) – Initializer functions to use for each variable of the variational posterior distribution. Keys correspond to variable names (arguments to the distribution), and values contain functions to initialize those variables given `shape` as the single argument.
- **var_transform** (*Dict[str, callable]*) – Transform to apply to each variable of the variational posterior.
- **name** (*str*) – Name of the parameter(s). Default = `'ScaleParameter'`

### Examples

Use *ScaleParameter* to create a standard deviation parameter for a *Normal* distribution:

TODO

**bayesian_update**()
Update priors to match the current posterior

**kl_loss**()
Compute the sum of the Kullback–Leibler divergences between this parameter's priors and its variational posteriors.

**property n_parameters**
Get the number of independent parameters

**property n_variables**
Get the number of underlying variables

**property posterior**
This Parameter's variational posterior distribution

**posterior_ci**(*ci: float = 0.95*, *n: int = 10000*)
Posterior confidence intervals

**Parameters**

- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95
- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals. Default = 10,000

**Returns**

- **lb** (*float or |ndarray|*) – Lower bound(s) of the confidence interval(s)
- **ub** (*float or |ndarray|*) – Upper bound(s) of the confidence interval(s)

**posterior_mean**()
Get the mean of the posterior distribution(s)

**posterior_plot**(*n: int = 10000*, *style: str = 'fill'*, *bins: Union[int, list, numpy.ndarray] = 20*, *ci: float = 0.0*, *bw: float = 0.075*, *alpha: float = 0.4*, *color=None*, ***kwargs*)
Plot distribution of samples from the posterior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each posterior distribution for estimating the density. Default = 10000
- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram
- **bins** (int or list or `ndarray`) – Number of bins to use for the posterior density histogram (if `style='hist'`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)

- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075
- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle
- **kwargs** – Additional keyword arguments are passed to `utils.plotting.plot_dist()`

**posterior_sample**(*n: int = 1*)

Sample from the posterior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the posterior distribution. Default = 1

**Returns** Samples from the parameter's posterior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** ndarray

**prior_plot**(*n: int = 10000*, *style: str = 'fill'*, *bins: Union[int, list, numpy.ndarray] = 20*, *ci: float = 0.0*, *bw: float = 0.075*, *alpha: float = 0.4*, *color=None*)

Plot distribution of samples from the prior distribution.

**Parameters**

- **n** (*int*) – Number of samples to take from each prior distribution for estimating the density. Default = 1000
- **style** (*str*) – Which style of plot to show. Available types are:
  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram
- **bins** (int or list or ndarray) – Number of bins to use for the prior density histogram (if `style='hist'`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075
- **alpha** (*float between 0 and 1*) – Transparency of fill/histogram
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle

**prior_sample**(*n: int = 1*)

Sample from the prior distribution.

**Parameters** **n** (*int > 0*) – Number of samples to draw from the prior distribution. Default = 1

**Returns** Samples from the parameter prior distribution. If `n>1` of size `(n, self.prior.shape)`. If `n==1`, of size `(self.prior.shape)`.

**Return type** ndarray

**property trainable_variables**
Get a list of trainable variables from the backend

**property variables**
Variables after applying their respective transformations

## 3.3 Modules

Modules are objects which take Tensor(s) as input, perform some computation on that Tensor, and output a Tensor. Modules can create and contain *Parameters*. For example, neural network layers are good examples of a *Module*, since they store parameters, and use those parameters to perform a computation (the forward pass of the data through the layer).

- *Module* - abstract base class for all modules
- *Dense* - fully-connected neural network layer
- *DenseNetwork* - a multi-layer dense neural network module
- *Sequential* - apply a list of modules sequentially
- *BatchNormalization* - normalize data per batch
- *Embedding* - embed categorical data in a lower-dimensional space

---

**class** probflow.modules.**Module**(**args*)
Bases: *probflow.utils.base.BaseModule*

Abstract base class for Modules.

TODO

**property parameters**
A list of *Parameters* in this *Module* and its sub-Modules.

**property modules**
A list of sub-Modules in this *Module*, including itself.

**property trainable_variables**
A list of trainable backend variables within this *Module*

**property n_parameters**
Get the number of independent parameters of this module

**property n_variables**
Get the number of underlying variables in this module

**bayesian_update**()
Perform a Bayesian update of all *Parameters* in this module. Sets the prior to the current variational posterior for all parameters.

**kl_loss**()
Compute the sum of the Kullback-Leibler divergences between priors and their variational posteriors for all *Parameters* in this *Module* and its sub-Modules.

**kl_loss_batch**()
Compute the sum of additional Kullback-Leibler divergences due to data in this batch

**reset_kl_loss**()
Reset additional loss due to KL divergences

**add_kl_loss**(*loss*, *d2=None*)
Add additional loss due to KL divergences.

**dumps**()
Serialize module object to bytes

**save**(*filename: str*)
Save module object to file

**Parameters** **filename** (*str*) – Filename for file to which to save this object

**class** probflow.modules.**Dense**(*d_in: int*, *d_out: int = 1*, *probabilistic: bool = True*, *flipout: bool = True*, *weight_kwargs: dict = {}*, *bias_kwargs: dict = {}*, *name: str = 'Dense'*)
Bases: probflow.modules.module.Module

Dense neural network layer.

TODO

---

**Will not use flipout when n_mc>1**

Note that this module uses the flipout estimator by default, but will not use the flipout estimator when we are taking multiple monte carlo samples per batch (when *n_mc* > 1). See *Model.fit()* for more info on setting the value of *n_mc*.

---

**Parameters**

- **d_in** (*int*) – Number of input dimensions.
- **d_out** (*int*) – Number of output dimensions (number of "units").
- **probabilistic** (*bool*) – Whether variational posteriors for the weights and biases should be probabilistic. If True (the default), will use Normal distributions for the variational posteriors. If False, will use Deterministic distributions.
- **flipout** (*bool*) – Whether to use the flipout estimator for this layer. Default is True. Usually, when the global flipout setting is set to True, will use flipout during training but not during inference. If this kwarg is set to False, will not use flipout even during training.
- **weight_kwargs** (*dict*) – Additional kwargs to pass to the Parameter constructor for the weight parameters. Default is an empty dict.
- **bias_kwargs** (*dict*) – Additional kwargs to pass to the Parameter constructor for the bias parameters. Default is an empty dict.
- **name** (*str*) – Name of this layer

**add_kl_loss**(*loss*, *d2=None*)
Add additional loss due to KL divergences.

**bayesian_update**()
Perform a Bayesian update of all *Parameters* in this module. Sets the prior to the current variational posterior for all parameters.

**dumps**()
Serialize module object to bytes

**kl_loss**()
Compute the sum of the Kullback-Leibler divergences between priors and their variational posteriors for all *Parameters* in this *Module* and its sub-Modules.

**kl_loss_batch**()
Compute the sum of additional Kullback-Leibler divergences due to data in this batch

**property modules**
A list of sub-Modules in this *Module*, including itself.

**property n_parameters**
Get the number of independent parameters of this module

**property n_variables**
Get the number of underlying variables in this module

**property parameters**
A list of *Parameters* in this *Module* and its sub-Modules.

**reset_kl_loss**()
Reset additional loss due to KL divergences

**save**(*filename: str*)
Save module object to file

**Parameters** **filename** (*str*) – Filename for file to which to save this object

**property trainable_variables**
A list of trainable backend variables within this *Module*

**class** probflow.modules.**DenseNetwork**(*d: List[int], activation: Callable = <function relu>, batch_norm: bool = False, batch_norm_loc: str = 'after', name: str = 'DenseNetwork', batch_norm_kwargs: dict = {}, **kwargs*)

Bases: probflow.modules.module.Module

A multilayer dense neural network

TODO: explain, math, diagram, examples, etc

**Parameters**

- **d** (*List[int]*) – Dimensionality (number of units) for each layer. The first element should be the dimensionality of the independent variable (number of features).
- **activation** (*callable*) – Activation function to apply to the outputs of each layer. Note that the activation function will not be applied to the outputs of the final layer. Default = $\max(0, x)$
- **probabilistic** (*bool*) – Whether variational posteriors for the weights and biases should be probabilistic. If True (the default), will use Normal distributions for the variational posteriors. If False, will use Deterministic distributions.
- **batch_norm** (*bool*) – Whether or not to use batch normalization in between layers of the network. Default is False.
- **batch_norm_loc** (*str {'after' or 'before'}*) – Where to apply the batch normalization. If `'after'`, applies the batch normalization after the activation. If `'before'`, applies the batch normalization before the activation. Default is `'after'`.
- **batch_norm_kwargs** (*dict*) – Additional parameters to pass to *BatchNormalization* for each layer.
- **kwargs** – Additional parameters are passed to *Dense* for each layer.

**layers**
List of *Dense* neural network layers to be applied

**Type** List[*Dense*]

**activations**
Activation function for each layer

**Type** List[callable]

**batch_norms**
Batch normalization layers

**Type** Union[None, List[*BatchNormalization*]]

**add_kl_loss**(*loss*, *d2=None*)
Add additional loss due to KL divergences.

**bayesian_update**()
Perform a Bayesian update of all *Parameters* in this module. Sets the prior to the current variational posterior for all parameters.

**dumps**()
Serialize module object to bytes

**kl_loss**()
Compute the sum of the Kullback-Leibler divergences between priors and their variational posteriors for all *Parameters* in this *Module* and its sub-Modules.

**kl_loss_batch**()
Compute the sum of additional Kullback-Leibler divergences due to data in this batch

**property modules**
A list of sub-Modules in this *Module*, including itself.

**property n_parameters**
Get the number of independent parameters of this module

**property n_variables**
Get the number of underlying variables in this module

**property parameters**
A list of *Parameters* in this *Module* and its sub-Modules.

**reset_kl_loss**()
Reset additional loss due to KL divergences

**save**(*filename: str*)
Save module object to file

**Parameters** **filename** (*str*) – Filename for file to which to save this object

**property trainable_variables**
A list of trainable backend variables within this *Module*

**class** probflow.modules.**Sequential**(*steps: List[Callable]*, *name: str = 'Sequential'*)
Bases: probflow.modules.module.Module

Apply a series of modules or functions sequentially.

TODO

**Parameters**

- **steps** (list of *Modules* or callables) – Steps to apply
- **name** (*str*) – Name of this module

**add_kl_loss**(*loss*, *d2=None*)
Add additional loss due to KL divergences.

**bayesian_update**()
Perform a Bayesian update of all *Parameters* in this module. Sets the prior to the current variational posterior for all parameters.

**dumps**()
Serialize module object to bytes

**kl_loss**()
Compute the sum of the Kullback-Leibler divergences between priors and their variational posteriors for all *Parameters* in this *Module* and its sub-Modules.

**kl_loss_batch**()
Compute the sum of additional Kullback-Leibler divergences due to data in this batch

**property modules**
A list of sub-Modules in this *Module*, including itself.

**property n_parameters**
Get the number of independent parameters of this module

**property n_variables**
Get the number of underlying variables in this module

**property parameters**
A list of *Parameters* in this *Module* and its sub-Modules.

**reset_kl_loss**()
Reset additional loss due to KL divergences

**save**(*filename: str*)
Save module object to file

**Parameters** **filename** (*str*) – Filename for file to which to save this object

**property trainable_variables**
A list of trainable backend variables within this *Module*

**class** probflow.modules.**BatchNormalization**(*shape: Union[int, List[int]], weight_posterior: Type[probflow.utils.base.BaseDistribution] = <class 'probflow.distributions.deterministic.Deterministic'>, bias_posterior: Type[probflow.utils.base.BaseDistribution] = <class 'probflow.distributions.deterministic.Deterministic'>, weight_prior: probflow.utils.base.BaseDistribution = <probflow.distributions.normal.Normal object>, bias_prior: probflow.utils.base.BaseDistribution = <probflow.distributions.normal.Normal object>, weight_initializer: Dict[str, Callable] = {'loc': <function xavier>}, bias_initializer: Dict[str, Callable] = {'loc': <function xavier>}, name='BatchNormalization'*)

Bases: probflow.modules.module.Module

A layer which normalizes its inputs.

Batch normalization is a technique which normalizes, re-scales, and offsets the output of one layer before passing it on to another layer[1]. It often leads to faster training of neural networks, and better generalization error

[1] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv preprint*, 2015. http://arxiv.org/abs/1502.03167

by stabilizing the change in the layers' input distributions, or perhaps by smoothing the optimization landscape[2].

Given a set of tensors for this batch, where $x_{ij}$ is the $i$-th element of the $j$-th sample in this batch, this layer returns an elementwise transformation of the input tensors according to:

$$\text{BatchNorm}(x_{ij}) = \gamma_i \left( \frac{x_{ij} - \mu_i}{\sigma_i} \right) + \beta_i$$

Where $\mu_i$ is the mean of the $i$-th element across the batch:

$$\mu_i = \frac{1}{N} \sum_{k=1}^{N} x_{ik}$$

and $\sigma_i$ is the standard deviation of the $i$-th element across the batch:

$$\sigma_i = \frac{1}{N} \sum_{k=1}^{N} (x_{ik} - \mu_i)^2$$

and $\gamma$ and $\beta$ are two free parameters for each element.

**Parameters**

- **shape** (int or list of int or `ndarray`) – Shape of the tensor to be batch-normalized.
- **name** (*str*) – Name for this layer. Default = 'BatchNormalization'
- **weight_posterior** (*Distribution*) – Probability distribution class to use to approximate the posterior for the weight parameter(s) ($\gamma$). Default = *Deterministic*
- **bias_posterior** (*Distribution*) – Probability distribution class to use to approximate the posterior for the bias parameter(s) ($\beta$). Default = *Deterministic*
- **weight_prior** (`None` or a *Distribution* object) – Prior probability distribution for the weight parameter(s) ($\gamma$). `None` or a *Distribution* function which has been instantiated with parameters. Default = *Normal* `(0,1)`
- **bias_prior** (`None` or a *Distribution* object) – Prior probability distribution for the bias parameter(s) ($\beta$). `None` or a *Distribution* function which has been instantiated with parameters. Default = *Normal* `(0,1)`
- **weight_initializer** (*dict of callables*) – Initializer functions to use for each variable of the variational posterior distribution for the weights ($\gamma$). Keys correspond to variable names (arguments to the distribution), and values contain functions to initialize those variables given `shape` as the single argument.
- **bias_initializer** (*dict of callables*) – Initializer functions to use for each variable of the variational posterior distribution for the biases ($\beta$). Keys correspond to variable names (arguments to the distribution), and values contain functions to initialize those variables given `shape` as the single argument.

[2] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How Does Batch Normalization Help Optimization? *arXiv preprint*, 2018. http://arxiv.org/abs/1805.11604

### Examples

Batch normalize the output of a *Dense* layer:

```
import probflow as pf

network = pf.Sequential([
    pf.Dense(d_in=7, d_out=100, bias=False),
    pf.BatchNormalization(100),
    tf.nn.relu,
    pf.Dense(d_in=100, d_out=1)
])
...
```

### References

**add_kl_loss**(*loss*, *d2=None*)
Add additional loss due to KL divergences.

**bayesian_update**()
Perform a Bayesian update of all *Parameters* in this module. Sets the prior to the current variational posterior for all parameters.

**dumps**()
Serialize module object to bytes

**kl_loss**()
Compute the sum of the Kullback-Leibler divergences between priors and their variational posteriors for all *Parameters* in this *Module* and its sub-Modules.

**kl_loss_batch**()
Compute the sum of additional Kullback-Leibler divergences due to data in this batch

**property modules**
A list of sub-Modules in this *Module*, including itself.

**property n_parameters**
Get the number of independent parameters of this module

**property n_variables**
Get the number of underlying variables in this module

**property parameters**
A list of *Parameters* in this *Module* and its sub-Modules.

**reset_kl_loss**()
Reset additional loss due to KL divergences

**save**(*filename: str*)
Save module object to file

**Parameters** **filename** (*str*) – Filename for file to which to save this object

**property trainable_variables**
A list of trainable backend variables within this *Module*

**class** probflow.modules.**Embedding**(*k: Union[int, List[int]]*, *d: Union[int, List[int]]*, *probabilistic: bool = False*, *name: str = 'Embedding'*, ***kwargs*)
Bases: probflow.modules.module.Module

A categorical embedding layer.

Maps an input variable containing non-negative integers to dense vectors. The length of the vectors (the dimensionality of the embedding) can be set with the `dims` keyword argument. The embedding is learned over the course of training: if there are N unique integers in the input, and the embedding dimensionality is M, a matrix of NxM free parameters is created and optimized to minimize the loss.

By default, a *Deterministic* distribution is used for the embedding variables' posterior distributions, with *Normal* `(0, 1)` priors. This corresponds to normal non-probabilistic embedding with L2 regularization.

The embeddings can be non-probabilistic (each integer corresponds to a single point in M-dimensional space, the default), or probabilistic (each integer corresponds to a M-dimensional multivariate distribution). Set the *probabilistic* kwarg to True to use probabilistic embeddings.

**Parameters**

- **k** (*int > 0 or List[int]*) – Number of categories to embed.
- **d** (*int > 0 or List[int]*) – Number of embedding dimensions.
- **posterior** (*Distribution* class) – Probability distribution class to use to approximate the posterior. Default = *Deterministic*
- **prior** (*Distribution* object) – Prior probability distribution which has been instantiated with parameters. Default = *Normal* `(0,1)`
- **initializer** (*dict of callables*) – Initializer functions to use for each variable of the variational posterior distribution. Keys correspond to variable names (arguments to the distribution), and values contain functions to initialize those variables given `shape` as the single argument.
- **probabilistic** (*bool*) – Whether variational posteriors for the weights and biases should be probabilistic. If False (the default), will use *Deterministic* distributions for the variational posteriors. If True, will use *Normal* distributions.
- **name** (*str*) – Name for this layer. Default = 'Embeddings'
- **kwargs** – Additional keyword arguments are passed to the *Parameter* constructor which creates the embedding variables.

### Examples

Embed 10k word IDs into a 50-dimensional space:

```
emb = Embedding(k=10000, d=50)

ids = tf.random.uniform([1000000], minval=1, maxval=10000,
                        dtype=tf.dtypes.int64)

embeddings = emb(ids)
```

TODO: fuller example

**add_kl_loss**(*loss*, *d2=None*)
Add additional loss due to KL divergences.

**bayesian_update**()
Perform a Bayesian update of all *Parameters* in this module. Sets the prior to the current variational posterior for all parameters.

**dumps**()
Serialize module object to bytes

**kl_loss**()
Compute the sum of the Kullback-Leibler divergences between priors and their variational posteriors for all *Parameters* in this *Module* and its sub-Modules.

**kl_loss_batch**()
Compute the sum of additional Kullback-Leibler divergences due to data in this batch

**property modules**
A list of sub-Modules in this *Module*, including itself.

**property n_parameters**
Get the number of independent parameters of this module

**property n_variables**
Get the number of underlying variables in this module

**property parameters**
A list of *Parameters* in this *Module* and its sub-Modules.

**reset_kl_loss**()
Reset additional loss due to KL divergences

**save**(*filename: str*)
Save module object to file

**Parameters** **filename** (*str*) – Filename for file to which to save this object

**property trainable_variables**
A list of trainable backend variables within this *Module*

## 3.4 Models

Models are objects which take Tensor(s) as input, perform some computation on those Tensor(s), and output probability distributions.

TODO: more...

- *Model*
- *ContinuousModel*
- *DiscreteModel*
- *CategoricalModel*

---

**class** probflow.models.**Model**(**args*)
Bases: probflow.modules.module.Module

Abstract base class for probflow models.

TODO

**This class inherits several methods and properties from :class:`.Module`:**

*** :attr:`~parameters`**

*** :attr:`~modules`**

*** :attr:`~trainable_variables`**

*** :attr:`~n_parameters`**

```
\* :attr:`~n_variables`
\* :meth:`~bayesian_update`
\* :meth:`~kl_loss`
\* :meth:`~kl_loss_batch`
\* :meth:`~reset_kl_loss`
\* :meth:`~add_kl_loss`
\* :meth:`~dumps`
\* :meth:`~save`
and adds model-specific methods:
\* :meth:`~log_likelihood`
\* :meth:`~train_step`
\* :meth:`~fit`
\* :meth:`~stop_training`
\* :meth:`~set_learning_rate`
\* :meth:`~predictive_sample`
\* :meth:`~aleatoric_sample`
\* :meth:`~epistemic_sample`
\* :meth:`~predict`
\* :meth:`~metric`
\* :meth:`~posterior_mean`
\* :meth:`~posterior_sample`
\* :meth:`~posterior_ci`
\* :meth:`~prior_sample`
\* :meth:`~posterior_plot`
\* :meth:`~prior_plot`
\* :meth:`~log_prob`
\* :meth:`~log_prob_by`
\* :meth:`~prob`
\* :meth:`~prob_by`
\* :meth:`~save`
\* :meth:`~summary`
```

### Example

See the user guide section on *Models*.

**log_likelihood**(*x_data*, *y_data*)
Compute the sum log likelihood of the model given a batch of data

**elbo_loss**(*x_data*, *y_data*, *n: int*, *n_mc: int*)
Compute the negative ELBO, scaled to a single sample.

**Parameters**

- **x_data** – The independent variable values (or None if this is a generative model)
- **y_data** – The dependent variable values
- **n** (*int*) – Total number of datapoints in the dataset
- **n_mc** (*int*) – Number of MC samples we're taking from the posteriors

**get_elbo**()
Get the current ELBO on training data

**train_step**(*x_data*, *y_data*)
Perform one training step

**fit**(*x*, *y=None*, *batch_size: int = 128*, *epochs: int = 200*, *shuffle: bool = False*, *optimizer=None*, *optimizer_kwargs: dict = {}*, *lr: Optional[float] = None*, *flipout: bool = True*, *num_workers: Optional[int] = None*, *callbacks: List[*probflow.utils.base.BaseCallback*] = []*, *eager: bool = False*, *n_mc: int = 1*)
Fit the model to data

TODO

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values (or, if fitting a generative model, the dependent variable values). Should be of shape (Nsamples,...)
- **y** (None or ndarray or DataFrame or Series) – Dependent variable values (or, if fitting a generative model, None). Should be of shape (Nsamples,...). Default = None
- **batch_size** (*int*) – Number of samples to use per minibatch. Default = 128
- **epochs** (*int*) – Number of epochs to train the model. Default = 200
- **shuffle** (*bool*) – Whether to shuffle the data each epoch. Note that this is ignored if x is a *DataGenerator* Default = True
- **optimizer** (None or a backend-specific optimizer) – What optimizer to use for optimizing the variational posterior distributions' variables. When the backend is TensorFlow the default is to use adam (tf.keras.optimizers.Adam). When the backend is PyTorch the default is to use TODO
- **optimizer_kwargs** (*dict*) – Keyword arguments to pass to the optimizer. Default is an empty dict.
- **lr** (*float*) – Learning rate for the optimizer. Note that the learning rate can be updated during training using the set_learning_rate method. Default is $\exp(-\log_{10}(N_p N_b))$, where $N_p$ is the number of parameters in the model, and $N_b$ is the number of samples per batch (batch_size).
- **flipout** (*bool*) – Whether to use flipout during training where possible Default = True

- **num_workers** (*None or int > 0*) – Number of parallel processes to run for loading the data. If `None`, will not use parallel processes. If an integer, will use a process pool with that many processes. Note that this parameter is ignored if a *DataGenerator* is passed as `x`. Default = None
- **callbacks** (*List[BaseCallback]*) – List of callbacks to run while training the model. Default is `[]`, i.e. no callbacks.
- **eager** (*bool*) – Whether to use eager execution. If False, will use `tf.function` (for TensorFlow) or tracing (for PyTorch) to optimize the model fitting. Note that even if eager=True, you can still use eager execution when using the model after it is fit. Default = False
- **n_mc** (*int*) – Number of monte carlo samples to take from the variational posteriors per minibatch. The default is to just take one per batch. Using a smaller number of MC samples is faster, but using a greater number of MC samples will decrease the variance of the gradients, leading to more stable parameter optimization.

### Example

See the user guide section on *Fitting a Model*.

**stop_training**()
Stop the training of the model

**set_learning_rate**(*lr*)
Set the learning rate used by this model's optimizer

**set_kl_weight**(*w*)
Set the weight of the KL term's contribution to the ELBO loss

**predictive_sample**(*x=None*, *n=1000*, *batch_size=None*)
Draw samples from the posterior predictive distribution given x

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predictive distribution. Size (num_samples, x.shape[0], ...)

**Return type** `ndarray`

**aleatoric_sample**(*x=None*, *n=1000*, *batch_size=None*)
Draw samples of the model's estimate given x, including only aleatoric uncertainty (uncertainty due to noise)

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.

- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples,x.shape[0],...)

**Return type** ndarray

**epistemic_sample**(*x=None*, *n=1000*, *batch_size=None*)

Draw samples of the model's estimate given x, including only epistemic uncertainty (uncertainty due to uncertainty as to the model's parameter values)

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples, x.shape[0], ...)

**Return type** ndarray

**predict**(*x=None*, *method='mean'*, *batch_size=None*)

Predict dependent variable using the model

TODO... using maximum a posteriori param estimates etc

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **method** (*str*) – Method to use for prediction. If `'mean'`, uses the mean of the predicted target distribution as the prediction. If `'mode'`, uses the mode of the distribution.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Predicted y-value for each sample in `x`. Of size (x.shape[0], y.shape[0], ..., y.shape[-1])

**Return type** ndarray

### Examples

TODO: Docs...

**metric**(*metric*, *x*, *y=None*, *batch_size=None*)

Compute a metric of model performance

TODO: docs

TODO: note that this doesn't work w/ generative models

**Parameters**

- **metric** (*str or callable*) – Metric to evaluate. Available metrics:
  - 'lp': log likelihood sum
  - 'log_prob': log likelihood sum

- 'accuracy': accuracy
- 'acc': accuracy
- 'mean_squared_error': mean squared error
- 'mse': mean squared error
- 'sum_squared_error': sum squared error
- 'sse': sum squared error
- 'mean_absolute_error': mean absolute error
- 'mae': mean absolute error
- 'r_squared': coefficient of determination
- 'r2': coefficient of determination
- 'recall': true positive rate
- 'sensitivity': true positive rate
- 'true_positive_rate': true positive rate
- 'tpr': true positive rate
- 'specificity': true negative rate
- 'selectivity': true negative rate
- 'true_negative_rate': true negative rate
- 'tnr': true negative rate
- 'precision': precision
- 'f1_score': F-measure
- 'f1': F-measure
- callable: a function which takes (y_true, y_pred)

**x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* to generate both x and y.

**y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").

**batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

**Return type** TODO

**posterior_mean**(*params=None*)

Get the mean of the posterior distribution(s)

TODO: Docs... params is a list of strings of params to plot

**Parameters** **params** (*str or List[str] or None*) – Parameter name(s) for which to compute the means. Default is to get the mean for all parameters in the model.

**Returns** Means of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the posterior means. The `ndarrays` are the same size as each parameter. Or just the `ndarray` if `params` was a str.

**Return type** dict

**posterior_sample** (*params=None*, *n=10000*)

Draw samples from parameter posteriors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get a sample for all parameters in the model.
- **num_samples** (*int*) – Number of samples to take from each posterior distribution. Default = 1000

**Returns** Samples from the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the posterior samples. The `ndarrays` are of size (`num_samples`, param.shape). Or just the `ndarray` if `params` was a str.

**Return type** dict

**posterior_ci** (*params=None*, *ci=0.95*, *n=10000*)

Posterior confidence intervals

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get the confidence intervals for all parameters in the model.
- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95
- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals Default = 10,000

**Returns** Confidence intervals of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain tuples. The first element of each tuple is the lower bound, and the second element is the upper bound. Or just a single tuple if params was a str

**Return type** dict

**prior_sample** (*params=None*, *n=10000*)

Draw samples from parameter priors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*list*) – List of parameter names to sample. Each element should be a str. Default is to sample priors of all parameters in the model.
- **n** (*int*) – Number of samples to take from each prior distribution. Default = 10000

**Returns** Samples from the parameter prior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the prior samples. The `ndarrays` are of size (`n`,param.shape).

**Return type** dict

**posterior_plot** (*params=None*, *cols=1*, ***kwargs*)
Plot posterior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the posterior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.posterior_plot()*

**prior_plot** (*params=None*, *cols=1*, ***kwargs*)
Plot prior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the prior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.prior_plot()*

**log_prob** (*x*, *y=None*, *individually=True*, *distribution=False*, *n=1000*, *batch_size=None*)
Compute the log probability of *y* given the model

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor) – Independent variable values of the dataset to evaluate (aka the "features").
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **individually** (*bool*) – If `individually` is True, returns log probability for each sample individually, so return shape is `(x.shape[0], ?)`. If `individually` is False, returns sum of all log probabilities, so return shape is `(1, ?)`.
- **distribution** (*bool*) – If `distribution` is True, returns log probability posterior distribution (`n` samples from the model), so return shape is `(?, n)`. If `distribution` is False, returns log posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is `(?, 1)`.
- **n** (*int*) – Number of samples to draw for each distribution if `distribution=True`.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns log_probs** – Log probabilities. Shape is determined by `individually`, `distribution`, and `n` kwargs.

**Return type** ndarray

**prob** (*x*, *y=None*, ***kwargs*)
Compute the probability of `y` given the model

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **individually** (*bool*) – If `individually` is True, returns probability for each sample individually, so return shape is `(x.shape[0], ?)`. If `individually` is False, returns product of all probabilities, so return shape is `(1, ?)`.
- **distribution** (*bool*) – If `distribution` is True, returns posterior probability distribution (`n` samples from the model), so return shape is `(?, n)`. If `distribution` is False, returns posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is `(?, 1)`.
- **n** (*int*) – Number of samples to draw for each distribution if `distribution=True`.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** **probs** – Probabilities. Shape is determined by `individually`, `distribution`, and `n` kwargs.

**Return type** ndarray

**summary**()
Show a summary of the model and its parameters.

TODO

TODO: though maybe this should be a method of module... model would have to add to it the observation dist

**add_kl_loss**(*loss*, *d2=None*)
Add additional loss due to KL divergences.

**bayesian_update**()
Perform a Bayesian update of all *Parameters* in this module. Sets the prior to the current variational posterior for all parameters.

**dumps**()
Serialize module object to bytes

**kl_loss**()
Compute the sum of the Kullback-Leibler divergences between priors and their variational posteriors for all *Parameters* in this *Module* and its sub-Modules.

**kl_loss_batch**()
Compute the sum of additional Kullback-Leibler divergences due to data in this batch

**property modules**
A list of sub-Modules in this *Module*, including itself.

**property n_parameters**
Get the number of independent parameters of this module

**property n_variables**
Get the number of underlying variables in this module

**property parameters**
A list of *Parameters* in this *Module* and its sub-Modules.

**reset_kl_loss**()
Reset additional loss due to KL divergences

**save**(*filename: str*)
Save module object to file

**Parameters** **filename** (*str*) – Filename for file to which to save this object

**property trainable_variables**
A list of trainable backend variables within this *Module*

**class** probflow.models.**ContinuousModel**(**args*)
Bases: probflow.models.model.Model

Abstract base class for probflow models where the dependent variable (the target) is continuous and 1-dimensional.

The only advantage to using this over the more general *Model* is that *ContinuousModel* also includes several methods specific to continuous models, for tasks such as getting the predictive intervals, coverage, R-squared value, or calibration metrics (see below for the full list of methods).

---

**Only supports scalar dependent variables**

Note that the methods of *ContinuousModel* only support scalar, continuous dependent variables (not *any* continuous model, as the name might suggest). For models which have a multidimensional output, just use the more general *Model*; for models with categorical output (i.e., classifiers), use *CategoricalModel*; and for models which have a discrete output (e.g. a Poisson regression), use *DiscreteModel*.

---

This class inherits several methods from *Module*:

- *parameters*
- *modules*
- *trainable_variables*
- *kl_loss()*
- *kl_loss_batch()*
- *reset_kl_loss()*
- *add_kl_loss()*

as well as several methods from *Model*:

- *log_likelihood()*
- *train_step()*
- *fit()*
- *stop_training()*
- *set_learning_rate()*
- *predictive_sample()*
- *aleatoric_sample()*
- *epistemic_sample()*

- *predict()*
- *metric()*
- *posterior_mean()*
- *posterior_sample()*
- *posterior_ci()*
- *prior_sample()*
- *posterior_plot()*
- *prior_plot()*
- *log_prob()*
- *prob()*
- *save()*
- *summary()*

and adds the following continuous-model-specific methods:

- *predictive_interval()*
- *aleatoric_interval()*
- *epistemic_interval()*
- *pred_dist_plot()*
- *predictive_prc()*
- *pred_dist_covered()*
- *pred_dist_coverage()*
- *coverage_by()*
- *r_squared()*
- *r_squared_plot()*
- *residuals()*
- *residuals_plot()*
- *calibration_curve()*
- *calibration_curve_plot()*
- *calibration_metric()*
- *sharpness()*
- coefficient_of_variation()

### Example

TODO

**predictive_interval**(*x*, *ci=0.95*, *side='both'*, *n=1000*, *batch_size=None*)

Compute confidence intervals on the model's estimate of the target given `x`, including all sources of uncertainty.

TODO: docs

TODO: using side= both, upper, vs lower

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *`DataGenerator`*) – Independent variable values of the dataset to evaluate (aka the "features").
- **ci** (*`float between 0 and 1`*) – Inner proportion of predictive distribution to use a the confidence interval. Default = 0.95
- **side** (*`str {'lower', 'upper', 'both'}`*) – Whether to get the one- or two-sided interval, and which side to get. If `'both'` (default), gets the upper and lower bounds of the central `ci` interval. If `'lower'`, gets the lower bound on the one-sided `ci` interval. If `'upper'`, gets the upper bound on the one-sided `ci` interval.
- **n** (*`int`*) – Number of samples from the posterior predictive distribution to take to compute the confidence intervals. Default = 1000
- **batch_size** (*`None or int`*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **lb** (|*ndarray*|) – Lower bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='upper'`.
- **ub** (|*ndarray*|) – Upper bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='lower'`.

**aleatoric_interval**(*x*, *ci=0.95*, *side='both'*, *n=1000*, *batch_size=None*)

Compute confidence intervals on the model's estimate of the target given `x`, including only aleatoric uncertainty (uncertainty due to noise).

TODO: docs

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *`DataGenerator`*) – Independent variable values of the dataset to evaluate (aka the "features").
- **ci** (*`float between 0 and 1`*) – Inner proportion of predictive distribution to use a the confidence interval. Default = 0.95
- **side** (*`str {'lower', 'upper', 'both'}`*) – Whether to get the one- or two-sided interval, and which side to get. If `'both'` (default), gets the upper and lower bounds of the central `ci` interval. If `'lower'`, gets the lower bound on the one-sided `ci` interval. If `'upper'`, gets the upper bound on the one-sided `ci` interval.
- **n** (*`int`*) – Number of samples from the aleatoric predictive distribution to take to compute the confidence intervals. Default = 1000
- **batch_size** (*`None or int`*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **lb** (*|ndarray|*) – Lower bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='upper'`.
- **ub** (*|ndarray|*) – Upper bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='lower'`.

**epistemic_interval** (*x*, *ci=0.95*, *side='both'*, *n=1000*, *batch_size=None*)

Compute confidence intervals on the model's estimate of the target given `x`, including only epistemic uncertainty (uncertainty due to uncertainty as to the model's parameter values).

TODO: docs

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **ci** (*float between 0 and 1*) – Inner proportion of predictive distribution to use a the confidence interval. Default = 0.95
- **side** (*str {'lower', 'upper', 'both'}*) – Whether to get the one- or two-sided interval, and which side to get. If `'both'` (default), gets the upper and lower bounds of the central `ci` interval. If `'lower'`, gets the lower bound on the one-sided `ci` interval. If `'upper'`, gets the upper bound on the one-sided `ci` interval.
- **n** (*int*) – Number of samples from the epistemic predictive distribution to take to compute the confidence intervals. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **lb** (*|ndarray|*) – Lower bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='upper'`.
- **ub** (*|ndarray|*) – Upper bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='lower'`.

**pred_dist_plot** (*x*, *n=10000*, *cols=1*, *individually=False*, *batch_size=None*, ***kwargs*)

Plot posterior predictive distribution from the model given `x`.

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model given `x`. Default = 10000
- **cols** (*int*) – Divide the subplots into a grid with this many columns (if `individually=True`.
- **individually** (*bool*) – If `True`, plot one subplot per datapoint in `x`, otherwise plot all the predictive distributions on the same plot.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_dist()*

### Example

TODO

**predictive_prc**(*x*, *y=None*, *n=1000*, *batch_size=None*)

Compute the percentile of each observation along the posterior predictive distribution.

TODO: Docs... Returns a percentile between 0 and 1

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model given x. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** prcs

**Return type** ndarray of float between 0 and 1

**pred_dist_covered**(*x*, *y=None*, *n: int = 1000*, *ci: float = 0.95*, *batch_size=None*)

Compute whether each observation was covered by a given confidence interval.

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model given x. Default = 1000
- **ci** (*float between 0 and 1*) – Confidence interval to use.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

**Return type** TODO

**pred_dist_coverage**(*x*, *y=None*, *n=1000*, *ci=0.95*, *batch_size=None*)

Compute what percent of samples are covered by a given confidence interval.

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").

- **n** (*int*) – Number of samples to draw from the model given x. Default = 1000
- **ci** (*float between 0 and 1*) – Confidence interval to use.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** **prc_covered** – Proportion of the samples which were covered by the predictive distribution's confidence interval.

**Return type** float between 0 and 1

**coverage_by**(*x_by*, *x*, *y=None*, *n: int = 1000*, *ci: float = 0.95*, *bins: int = 30*, *plot: bool = True*, *ideal_line_kwargs: dict = {}*, *batch_size=None*, ***kwargs*)

Compute and plot the coverage of a given confidence interval of the posterior predictive distribution as a function of specified independent variables.

TODO: Docs...

**Parameters**

- **x_by** (*int or str or list of int or list of str*) – Which independent variable(s) to plot the log probability as a function of. That is, which columns in x to plot by.
- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **ci** (*float between 0 and 1*) – Inner percentile to find the coverage of. For example, if ci=0.95, will compute the coverage of the inner 95% of the posterior predictive distribution.
- **bins** (*int*) – Number of bins to use for x_by
- **ideal_line_kwargs** (*dict*) – Dict of args to pass to matplotlib.pyplot.plot for ideal coverage line.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to plot_by

**Returns**

- **xo** (|*ndarray*|) – Values of x_by corresponding to bin centers.
- **co** (|*ndarray*|) – Coverage of the ci confidence interval of the predictive distribution in each bin.

**r_squared**(*x*, *y=None*, *n=1000*, *batch_size=None*)

Compute the Bayesian R-squared distribution (Gelman et al., 2018).

TODO: more info

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").

- **n** (*int*) – Number of posterior draws to use for computing the r-squared distribution. Default = *1000*.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the r-squared distribution. Size: `(num_samples,)`.

**Return type** `ndarray`

### Examples

TODO: Docs...

### References

- Andrew Gelman, Ben Goodrich, Jonah Gabry, & Aki Vehtari. R-squared for Bayesian regression models. *The American Statistician*, 2018.

**r_squared_plot**(*x*, *y=None*, *n=1000*, *style='hist'*, *batch_size=None*, ***kwargs*)

Plot the Bayesian R-squared distribution.

See *r_squared()* for more info on the Bayesian R-squared metric.

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series`) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of posterior draws to use for computing the r-squared distribution. Default = *1000*.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_dist()*

### Example

TODO

**residuals**(*x*, *y=None*, *batch_size=None*)

Compute the residuals of the model's predictions.

TODO: docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series`) – Dependent variable values of the dataset to evaluate (aka the "target").

- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The residuals.

**Return type** ndarray

### Example

TODO

**residuals_plot**(*x*, *y=None*, *batch_size=None*, ***kwargs*)

Plot the distribution of residuals of the model's predictions.

TODO: docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_dist()*

### Example

TODO

**calibration_curve**(*x*, *y*, *n=1000*, *resolution=100*, *batch_size=None*)

Compute the regression calibration curve (Kuleshov et al., 2018).

The regression calibration curve compares the empirical cumulative probability to the cumulative probability predicted by a regression model (Kuleshov et al., 2018). First, a vector $p$ of $m$ confidence levels are chosen, which correspond to the predicted cumulative probabilities:

$$0 \leq p_1 \leq p_2 \leq \ldots \leq p_m \leq 1$$

Then, a vector of empirical frequencies $\hat{p}$ at each of the predicted frequencies is computed by using validation data:

$$\hat{p}_j = \frac{1}{N} \sum_{i=1}^{N} [P_M(x_i \leq y_i) \leq p_j]$$

where $N$ is the number of validation datapoints, $P_M(x_i \leq y_i)$ is the model's predicted cumulative probability of datapoint $i$ (i.e., the percentile along the model's predicted probability distribution at which the true value of $y_i$ falls), and $\sum_i [a_i \leq b_i]$ is just the count of elements of $a$ which are less than corresponding elements in $b$.

The calibration curve then plots $p$ against $\hat{p}$.

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.

- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model for computing the predictive percentile. Default = 1000
- **resolution** (*int*) – Number of confidence levels to evaluate at. This corresponds to the $m$ parameter in section 3.5 of (Kuleshov et al., 2018).
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **p** (|*ndarray*|) – The predicted cumulative frequencies, $p$.
- **p_hat** (|*ndarray*|) – The empirical cumulative frequencies, $\hat{p}$.

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute the calibration curve with *calibration_curve()*:

```
p_pred, p_empirical = model.calibration_curve(x_val, y_val)
```

The returned values can be used directly or plotted against one another to get the calibration curve (as in Figure 3 in Kuleshov et al., 2018)

```
import matplotlib.pyplot as plt
plt.plot(p_pred, p_empirical)
```

Or, even more simply, just use *calibration_curve_plot()*.

**See also:**

- *calibration_curve_plot()*
- expected_calibration_error()

### References


- Volodymyr Kuleshov, Nathan Fenner, and Stefano Ermon. Accurate Uncertainties for Deep Learning Using Calibrated Regression, 2018.


**calibration_curve_plot** (*x*, *y*, *n=1000*, *resolution=100*, *batch_size=None*, ***kwargs*)

Plot the regression calibration curve.

See *calibration_curve()* for more info about the regression calibration curve.

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.

- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model for computing the predictive percentile. Default = 1000
- **resolution** (*int*) – Number of confidence levels to evaluate at. This corresponds to the $m$ parameter in section 3.5 of (Kuleshov et al., 2018).
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_dist()*

**See also:**

- *calibration_curve()*
- expected_calibration_error()

**calibration_metric**(*metric*, *x*, *y=None*, *n=1000*, *resolution=100*, *batch_size=None*)

Compute one or more of several calibration metrics

Regression calibration metrics measure the error between a model's regression calibration curve and the ideal calibration curve - i.e., what the curve would be if the model were perfectly calibrated (see Kuleshov et al., 2018 and Chung et al., 2020). First, a vector $p$ of $m$ confidence levels are chosen, which correspond to the predicted cumulative probabilities:

$$0 \leq p_1 \leq p_2 \leq \ldots \leq p_m \leq 1$$

Then, a vector of empirical frequencies $\hat{p}$ at each of the predicted frequencies is computed by using validation data:

$$\hat{p}_j = \frac{1}{N} \sum_{i=1}^{N} [P_M(x_i \leq y_i) \leq p_j]$$

where $N$ is the number of validation datapoints, $P_M(x_i \leq y_i)$ is the model's predicted cumulative probability of datapoint $i$ (i.e., the percentile along the model's predicted probability distribution at which the true value of $y_i$ falls), and $\sum_i [a_i \leq b_i]$ is just the count of elements of $a$ which are less than corresponding elements in $b$.

Various metrics can be computed from these curves to measure how accurately the regression model captures uncertainty:

The **mean squared calibration error (MSCE)** is the mean squared error between the empirical and predicted frequencies,

$$MSCE = \frac{1}{m} \sum_{j=1}^{m} (p_j - \hat{p}_j)^2$$

The **root mean squared calibration error (RMSCE)** is just the square root of the MSCE:

$$RMSCE = \sqrt{\frac{1}{m} \sum_{j=1}^{m} (p_j - \hat{p}_j)^2}$$

The **mean absolute calibration error (MACE)** is the mean of the absolute differences between the empirical and predicted frequencies:

$$MACE = \frac{1}{m} \sum_{j=1}^{m} |p_j - \hat{p}_j|$$

And the **miscalibration area (MA)** is the area between the calibration curve and the ideal calibration curve (the identity line from (0, 0) to (1, 1):

$$MA = \int_0^1 p_x - \hat{p}_x dx$$

Note that MA is equal to MACE as the number of bins (set by the `resolution` keyword argument) goes to infinity.

To choose which metric to compute, pass the name of the metric (`msce`, `rmsce`, `mace`, or `ma`) as the first argument to this function (or a list of them to compute multiple).

See Kuleshov et al., 2018, Chung et al., 2020 and the user guide page on *Evaluating Model Performance* for discussions of evaluating uncertainty estimates using calibration metrics, among other metrics. Note that calibration is generally less important than accuracy, but more important than other metrics like *`sharpness()`* and any *`dispersion_metric()`*.

**Parameters**

- **metric** (*`str {'msce', 'rmsce', 'mace', or 'ma'} or List[str]`*) – Which metric(s) to compute (see above for the definition of each metric). To compute multiple metrics, pass a list of the metric names you'd like to compute. Available metrics are:
  - `msce`: mean squared calibration error
  - `rmsce`: root mean squared calibration error
  - `mace`: mean absolute calibration error
  - `ma`: miscalibration area
- **x** (`ndarray` or `DataFrame` or `Series` or *`DataGenerator`*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *`DataGenerator`* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series`) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*`int`*) – Number of samples to draw from the model for computing the predictive percentile. Default = 1000
- **resolution** (*`int`*) – Number of confidence levels to evaluate at. This corresponds to the $m$ parameter in section 3.5 of (Kuleshov et al., 2018).
- **batch_size** (*`None or int`*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The requested calibration metric. If a list of metric names was passed, will return a dict whose keys are the metrics, and whose values are the corresponding metric values.

**Return type** float or Dict[str, float]

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute different calibration metrics using `expected_calibration_error()`. For example, to compute the mean squared calibration error (MSCE):

```
>>> model.calibration_metric("msce", x_val, y_val)
0.123
```

Or, to compute the mean absolute calibration error (MACE):

```
>>> model.calibration_metric("mace", x_val, y_val)
0.211
```

To compute multiple metrics at the same time, pass a list of metric names:

```
>>> model.calibration_metric(["msce", "mace"], x_val, y_val)
{"msce": 0.123, "mace": 0.211}
```

**See also:**

- *calibration_curve()*
- *calibration_curve_plot()*
- *sharpness()*
- *dispersion_metric()*

### References


- Volodymyr Kuleshov, Nathan Fenner, and Stefano Ermon. Accurate Uncertainties for Deep Learning Using Calibrated Regression, 2018.
- Youngseog Chung, Willie Neiswanger, Ian Char, Jeff Schneider. Beyond Pinball Loss: Quantile Methods for Calibrated Uncertainty Quantification, 2020.


**sharpness**(*x*, *n=1000*, *batch_size=None*)

Compute the sharpness of the model's uncertainty estimates

The "sharpness" of a model's uncertainty estimates is the root mean of the estimated variances:

$$SHA = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \mathrm{Var}(\hat{Y}_i)}$$

See Tran et al., 2020 and the user guide page on *Evaluating Model Performance* for discussions of evaluating uncertainty estimates using sharpness, among other metrics. Note that the sharpness should generally be one of the later things you consider - accuracy and calibration usually being more important.

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **n** (*int*) – Number of samples to draw from the model. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The sharpness of the model's uncertainty estimates

**Return type** float

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute the sharpness of our model's predictions with:

```
>>> model.sharpness(x_val)
0.173
```

**See also:**

- *calibration_metric()*
- *dispersion_metric()*

### References


- Kevin Tran, Willie Neiswanger, Junwoong Yoon, Qingyang Zhang, Eric Xing, Zachary W. Ulissi. Methods for comparing uncertainty quantifications for material property predictions, 2020.


**dispersion_metric**(*metric*, *x*, *n=1000*, *batch_size=None*)

Compute one or more of several calibration metrics

Dispersion metrics measure how much a model's uncertainty estimates vary. There are several different dispersion metrics:

The **coefficient of variation** ($C_v$) is the ratio of the standard deviation to the mean (of the model's uncertainty standard deviations):

$$C_v =$$

The **quartile coefficient of dispersion** ($QCD$) is less sensitive to outliers, as it simply measures the difference between the first and third quartile (of the model's uncertainty standard deviations) to their sum:

$$QCD = \frac{Q_3 - Q_1}{Q_3 + Q_1}$$

See Tran et al., 2020 and the user guide page on *Evaluating Model Performance* for discussions of evaluating uncertainty estimates using dispersion metrics, among other metrics. Note that dispersion metrics should generally be one of the last things you consider - accuracy, calibration, and sharpness usually being more important.

**Parameters**

- **metric** (*str {'cv' or 'qcd'} or List[str]*) – Dispersion metric to compute. Or,
- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **n** (*int*) – Number of samples to draw from the model. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The requested dispersion metric. If a list of metric names was passed, will return a dict whose keys are the metrics, and whose values are the corresponding metric values.

**Return type** float or Dict[str, float]

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute the coefficient of variation of our model's predictions with:

```
>>> model.dispersion_metric('cv', x_val)
0.732
```

Or the quartile coefficient of dispersion with:

```
>>> model.dispersion_metric('qcd', x_val)
0.625
```

**See also:**

- *calibration_metric()*
- *sharpness()*

### References


- Kevin Tran, Willie Neiswanger, Junwoong Yoon, Qingyang Zhang, Eric Xing, Zachary W. Ulissi. Methods for comparing uncertainty quantifications for material property predictions, 2020.


**add_kl_loss**(*loss*, *d2=None*)

Add additional loss due to KL divergences.

**aleatoric_sample**(*x=None*, *n=1000*, *batch_size=None*)

Draw samples of the model's estimate given x, including only aleatoric uncertainty (uncertainty due to noise)

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples,x.shape[0],...)

**Return type** ndarray

**bayesian_update**()
Perform a Bayesian update of all *Parameters* in this module. Sets the prior to the current variational posterior for all parameters.

**dumps**()
Serialize module object to bytes

**elbo_loss**(*x_data*, *y_data*, *n: int*, *n_mc: int*)
Compute the negative ELBO, scaled to a single sample.

**Parameters**

- **x_data** – The independent variable values (or None if this is a generative model)
- **y_data** – The dependent variable values
- **n** (*int*) – Total number of datapoints in the dataset
- **n_mc** (*int*) – Number of MC samples we're taking from the posteriors

**epistemic_sample**(*x=None*, *n=1000*, *batch_size=None*)
Draw samples of the model's estimate given x, including only epistemic uncertainty (uncertainty due to uncertainty as to the model's parameter values)

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples, x.shape[0], ...)

**Return type** ndarray

**fit**(*x*, *y=None*, *batch_size: int = 128*, *epochs: int = 200*, *shuffle: bool = False*, *optimizer=None*, *optimizer_kwargs: dict = {}*, *lr: Optional[float] = None*, *flipout: bool = True*, *num_workers: Optional[int] = None*, *callbacks: List[*probflow.utils.base.BaseCallback*] = []*, *eager: bool = False*, *n_mc: int = 1*)
Fit the model to data

TODO

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values (or, if fitting a generative model, the dependent variable values). Should be of shape (Nsamples,...)

- **y** (None or ndarray or DataFrame or Series) – Dependent variable values (or, if fitting a generative model, None). Should be of shape (Nsamples,...). Default = None
- **batch_size** (*int*) – Number of samples to use per minibatch. Default = 128
- **epochs** (*int*) – Number of epochs to train the model. Default = 200
- **shuffle** (*bool*) – Whether to shuffle the data each epoch. Note that this is ignored if x is a *DataGenerator* Default = True
- **optimizer** (None or a backend-specific optimizer) – What optimizer to use for optimizing the variational posterior distributions' variables. When the backend is TensorFlow the default is to use adam (tf.keras.optimizers.Adam). When the backend is PyTorch the default is to use TODO
- **optimizer_kwargs** (*dict*) – Keyword arguments to pass to the optimizer. Default is an empty dict.
- **lr** (*float*) – Learning rate for the optimizer. Note that the learning rate can be updated during training using the set_learning_rate method. Default is $\exp(-\log_{10}(N_p N_b))$, where $N_p$ is the number of parameters in the model, and $N_b$ is the number of samples per batch (batch_size).
- **flipout** (*bool*) – Whether to use flipout during training where possible Default = True
- **num_workers** (*None or int > 0*) – Number of parallel processes to run for loading the data. If None, will not use parallel processes. If an integer, will use a process pool with that many processes. Note that this parameter is ignored if a *DataGenerator* is passed as x. Default = None
- **callbacks** (*List[BaseCallback]*) – List of callbacks to run while training the model. Default is [], i.e. no callbacks.
- **eager** (*bool*) – Whether to use eager execution. If False, will use tf.function (for TensorFlow) or tracing (for PyTorch) to optimize the model fitting. Note that even if eager=True, you can still use eager execution when using the model after it is fit. Default = False
- **n_mc** (*int*) – Number of monte carlo samples to take from the variational posteriors per minibatch. The default is to just take one per batch. Using a smaller number of MC samples is faster, but using a greater number of MC samples will decrease the variance of the gradients, leading to more stable parameter optimization.

### Example

See the user guide section on *Fitting a Model*.

**get_elbo**()
Get the current ELBO on training data

**kl_loss**()
Compute the sum of the Kullback-Leibler divergences between priors and their variational posteriors for all *Parameters* in this *Module* and its sub-Modules.

**kl_loss_batch**()
Compute the sum of additional Kullback-Leibler divergences due to data in this batch

**log_likelihood**(*x_data*, *y_data*)
Compute the sum log likelihood of the model given a batch of data

**log_prob**(*x*, *y=None*, *individually=True*, *distribution=False*, *n=1000*, *batch_size=None*)
Compute the log probability of *y* given the model

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor) – Independent variable values of the dataset to evaluate (aka the "features").
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **individually** (*bool*) – If individually is True, returns log probability for each sample individually, so return shape is (x.shape[0], ?). If individually is False, returns sum of all log probabilities, so return shape is (1, ?).
- **distribution** (*bool*) – If distribution is True, returns log probability posterior distribution (n samples from the model), so return shape is (?, n). If distribution is False, returns log posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is (?, 1).
- **n** (*int*) – Number of samples to draw for each distribution if distribution=True.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns log_probs** – Log probabilities. Shape is determined by individually, distribution, and n kwargs.

**Return type** ndarray

**metric**(*metric*, *x*, *y=None*, *batch_size=None*)
Compute a metric of model performance

TODO: docs

TODO: note that this doesn't work w/ generative models

**Parameters**

- **metric** (*str or callable*) – Metric to evaluate. Available metrics:
  - 'lp': log likelihood sum
  - 'log_prob': log likelihood sum
  - 'accuracy': accuracy
  - 'acc': accuracy
  - 'mean_squared_error': mean squared error
  - 'mse': mean squared error
  - 'sum_squared_error': sum squared error
  - 'sse': sum squared error
  - 'mean_absolute_error': mean absolute error
  - 'mae': mean absolute error
  - 'r_squared': coefficient of determination
  - 'r2': coefficient of determination
  - 'recall': true positive rate

- 'sensitivity': true positive rate
- 'true_positive_rate': true positive rate
- 'tpr': true positive rate
- 'specificity': true negative rate
- 'selectivity': true negative rate
- 'true_negative_rate': true negative rate
- 'tnr': true negative rate
- 'precision': precision
- 'f1_score': F-measure
- 'f1': F-measure
- callable: a function which takes (y_true, y_pred)

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* to generate both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

**Return type** TODO

**property modules**
A list of sub-Modules in this *Module*, including itself.

**property n_parameters**
Get the number of independent parameters of this module

**property n_variables**
Get the number of underlying variables in this module

**property parameters**
A list of *Parameters* in this *Module* and its sub-Modules.

**posterior_ci**(*params=None*, *ci=0.95*, *n=10000*)
Posterior confidence intervals

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get the confidence intervals for all parameters in the model.
- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95
- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals Default = 10,000

**Returns** Confidence intervals of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain tuples. The first element of each

tuple is the lower bound, and the second element is the upper bound. Or just a single tuple if params was a str

**Return type** dict

**posterior_mean**(*params=None*)
Get the mean of the posterior distribution(s)

TODO: Docs... params is a list of strings of params to plot

**Parameters** **params** (*str or List[str] or None*) – Parameter name(s) for which to compute the means. Default is to get the mean for all parameters in the model.

**Returns** Means of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain ndarrays with the posterior means. The ndarrays are the same size as each parameter. Or just the ndarray if params was a str.

**Return type** dict

**posterior_plot**(*params=None*, *cols=1*, ***kwargs*)
Plot posterior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the posterior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.posterior_plot()*

**posterior_sample**(*params=None*, *n=10000*)
Draw samples from parameter posteriors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get a sample for all parameters in the model.
- **num_samples** (*int*) – Number of samples to take from each posterior distribution. Default = 1000

**Returns** Samples from the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain ndarrays with the posterior samples. The ndarrays are of size (num_samples, param.shape). Or just the ndarray if params was a str.

**Return type** dict

**predict**(*x=None*, *method='mean'*, *batch_size=None*)
Predict dependent variable using the model

TODO... using maximum a posteriori param estimates etc

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").

- **method** (*str*) – Method to use for prediction. If `'mean'`, uses the mean of the predicted target distribution as the prediction. If `'mode'`, uses the mode of the distribution.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Predicted y-value for each sample in `x`. Of size (x.shape[0], y.shape[0], ..., y.shape[-1])

**Return type** ndarray

### Examples

TODO: Docs...

**predictive_sample**(*x=None*, *n=1000*, *batch_size=None*)

Draw samples from the posterior predictive distribution given x

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predictive distribution. Size (num_samples, x.shape[0], ...)

**Return type** ndarray

**prior_plot**(*params=None*, *cols=1*, ***kwargs*)

Plot prior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the prior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.prior_plot()*

**prior_sample**(*params=None*, *n=10000*)

Draw samples from parameter priors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*list*) – List of parameter names to sample. Each element should be a str. Default is to sample priors of all parameters in the model.
- **n** (*int*) – Number of samples to take from each prior distribution. Default = 10000

**Returns** Samples from the parameter prior distributions. A dictionary where the keys contain the parameter names and the values contain ndarrays with the prior samples. The ndarrays are of size (n,param.shape).

**Return type** dict

**prob**(*x*, *y=None*, ***kwargs*)
Compute the probability of `y` given the model

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **individually** (*bool*) – If `individually` is True, returns probability for each sample individually, so return shape is `(x.shape[0], ?)`. If `individually` is False, returns product of all probabilities, so return shape is `(1, ?)`.
- **distribution** (*bool*) – If `distribution` is True, returns posterior probability distribution (`n` samples from the model), so return shape is `(?, n)`. If `distribution` is False, returns posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is `(?, 1)`.
- **n** (*int*) – Number of samples to draw for each distribution if `distribution=True`.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** **probs** – Probabilities. Shape is determined by `individually`, `distribution`, and `n` kwargs.

**Return type** `ndarray`

**reset_kl_loss**()
Reset additional loss due to KL divergences

**save**(*filename: str*)
Save module object to file

**Parameters** **filename** (*str*) – Filename for file to which to save this object

**set_kl_weight**(*w*)
Set the weight of the KL term's contribution to the ELBO loss

**set_learning_rate**(*lr*)
Set the learning rate used by this model's optimizer

**stop_training**()
Stop the training of the model

**summary**()
Show a summary of the model and its parameters.

TODO

TODO: though maybe this should be a method of module... model would have to add to it the observation dist

**train_step**(*x_data*, *y_data*)
Perform one training step

**property trainable_variables**
A list of trainable backend variables within this *Module*

**class** probflow.models.**DiscreteModel**(**args*)

Bases: probflow.models.continuous_model.ContinuousModel

Abstract base class for probflow models where the dependent variable (the target) is discrete (e.g. drawn from a Poisson distribution).

TODO : why use this over just Model

This class inherits several methods from *Module*:

- *parameters*
- *modules*
- *trainable_variables*
- *kl_loss()*
- *kl_loss_batch()*
- *reset_kl_loss()*
- *add_kl_loss()*

as well as several methods from *Model*:

- *log_likelihood()*
- *train_step()*
- *fit()*
- *stop_training()*
- *set_learning_rate()*
- *predictive_sample()*
- *aleatoric_sample()*
- *epistemic_sample()*
- *predict()*
- *metric()*
- *posterior_mean()*
- *posterior_sample()*
- *posterior_ci()*
- *prior_sample()*
- *posterior_plot()*
- *prior_plot()*
- *log_prob()*
- *prob()*
- *save()*
- *summary()*

as well as several methods from *ContinuousModel*:

- *predictive_interval()*
- *predictive_prc()*

- *pred_dist_covered()*
- *pred_dist_coverage()*
- *coverage_by()*
- *residuals()*

but overrides the following discrete-model-specific methods:

- *pred_dist_plot()*
- *residuals_plot()*

Note that *DiscreteModel* does *not* implement *r_squared()* or *r_squared_plot()*.

### Example

TODO

**pred_dist_plot**(*x*, *n=10000*, *cols=1*, ***kwargs*)
Plot posterior predictive distribution from the model given `x`.

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model given `x`. Default = 10000
- **cols** (*int*) – Divide the subplots into a grid with this many columns (if `individually=True`.
- ****kwargs** – Additional keyword arguments are passed to *plot_discrete_dist()*

**r_squared**(**args*, ***kwargs*)
Cannot compute R squared for a discrete model

**r_squared_plot**(**args*, ***kwargs*)
Cannot compute R squared for a discrete model

**add_kl_loss**(*loss*, *d2=None*)
Add additional loss due to KL divergences.

**aleatoric_interval**(*x*, *ci=0.95*, *side='both'*, *n=1000*, *batch_size=None*)
Compute confidence intervals on the model's estimate of the target given `x`, including only aleatoric uncertainty (uncertainty due to noise).

TODO: docs

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **ci** (*float between 0 and 1*) – Inner proportion of predictive distribution to use a the confidence interval. Default = 0.95
- **side** (*str {'lower', 'upper', 'both'}*) – Whether to get the one- or two-sided interval, and which side to get. If `'both'` (default), gets the upper and lower bounds of the central `ci` interval. If `'lower'`, gets the lower bound on the one-sided `ci` interval. If `'upper'`, gets the upper bound on the one-sided `ci` interval.

- **n** (*int*) – Number of samples from the aleatoric predictive distribution to take to compute the confidence intervals. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **lb** (|*ndarray*|) – Lower bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='upper'`.
- **ub** (|*ndarray*|) – Upper bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='lower'`.

**aleatoric_sample**(*x=None*, *n=1000*, *batch_size=None*)

Draw samples of the model's estimate given x, including only aleatoric uncertainty (uncertainty due to noise)

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples,x.shape[0],...)

**Return type** `ndarray`

**bayesian_update**()

Perform a Bayesian update of all *Parameters* in this module. Sets the prior to the current variational posterior for all parameters.

**calibration_curve**(*x*, *y*, *n=1000*, *resolution=100*, *batch_size=None*)

Compute the regression calibration curve (Kuleshov et al., 2018).

The regression calibration curve compares the empirical cumulative probability to the cumulative probability predicted by a regression model (Kuleshov et al., 2018). First, a vector $p$ of $m$ confidence levels are chosen, which correspond to the predicted cumulative probabilities:

$$0 \leq p_1 \leq p_2 \leq \ldots \leq p_m \leq 1$$

Then, a vector of empirical frequencies $\hat{p}$ at each of the predicted frequencies is computed by using validation data:

$$\hat{p}_j = \frac{1}{N} \sum_{i=1}^{N} [P_M(x_i \leq y_i) \leq p_j]$$

where $N$ is the number of validation datapoints, $P_M(x_i \leq y_i)$ is the model's predicted cumulative probability of datapoint $i$ (i.e., the percentile along the model's predicted probability distribution at which the true value of $y_i$ falls), and $\sum_i [a_i \leq b_i]$ is just the count of elements of $a$ which are less than corresponding elements in $b$.

The calibration curve then plots $p$ against $\hat{p}$.

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model for computing the predictive percentile. Default = 1000
- **resolution** (*int*) – Number of confidence levels to evaluate at. This corresponds to the $m$ parameter in section 3.5 of (Kuleshov et al., 2018).
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **p** (|*ndarray*|) – The predicted cumulative frequencies, $p$.
- **p_hat** (|*ndarray*|) – The empirical cumulative frequencies, $\hat{p}$.

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute the calibration curve with *calibration_curve()*:

```
p_pred, p_empirical = model.calibration_curve(x_val, y_val)
```

The returned values can be used directly or plotted against one another to get the calibration curve (as in Figure 3 in Kuleshov et al., 2018)

```
import matplotlib.pyplot as plt
plt.plot(p_pred, p_empirical)
```

Or, even more simply, just use *calibration_curve_plot()*.

**See also:**

- *calibration_curve_plot()*
- expected_calibration_error()

### References


- Volodymyr Kuleshov, Nathan Fenner, and Stefano Ermon. Accurate Uncertainties for Deep Learning Using Calibrated Regression, 2018.


**calibration_curve_plot**(*x*, *y*, *n=1000*, *resolution=100*, *batch_size=None*, ***kwargs*)

Plot the regression calibration curve.

See *calibration_curve()* for more info about the regression calibration curve.

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model for computing the predictive percentile. Default = 1000
- **resolution** (*int*) – Number of confidence levels to evaluate at. This corresponds to the $m$ parameter in section 3.5 of (Kuleshov et al., 2018).
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_dist()*

**See also:**

- *calibration_curve()*
- expected_calibration_error()

**calibration_metric**(*metric*, *x*, *y=None*, *n=1000*, *resolution=100*, *batch_size=None*)

Compute one or more of several calibration metrics

Regression calibration metrics measure the error between a model's regression calibration curve and the ideal calibration curve - i.e., what the curve would be if the model were perfectly calibrated (see Kuleshov et al., 2018 and Chung et al., 2020). First, a vector $p$ of $m$ confidence levels are chosen, which correspond to the predicted cumulative probabilities:

$$0 \leq p_1 \leq p_2 \leq \ldots \leq p_m \leq 1$$

Then, a vector of empirical frequencies $\hat{p}$ at each of the predicted frequencies is computed by using validation data:

$$\hat{p}_j = \frac{1}{N} \sum_{i=1}^{N} [P_M(x_i \leq y_i) \leq p_j]$$

where $N$ is the number of validation datapoints, $P_M(x_i \leq y_i)$ is the model's predicted cumulative probability of datapoint $i$ (i.e., the percentile along the model's predicted probability distribution at which the true value of $y_i$ falls), and $\sum_i [a_i \leq b_i]$ is just the count of elements of $a$ which are less than corresponding elements in $b$.

Various metrics can be computed from these curves to measure how accurately the regression model captures uncertainty:

The **mean squared calibration error (MSCE)** is the mean squared error between the empirical and predicted frequencies,

$$MSCE = \frac{1}{m} \sum_{j=1}^{m} (p_j - \hat{p}_j)^2$$

The **root mean squared calibration error (RMSCE)** is just the square root of the MSCE:

$$RMSCE = \sqrt{\frac{1}{m} \sum_{j=1}^{m} (p_j - \hat{p}_j)^2}$$

The **mean absolute calibration error (MACE)** is the mean of the absolute differences between the empirical and predicted frequencies:

$$MACE = \frac{1}{m}\sum_{j=1}^{m} |p_j - \hat{p}_j|$$

And the **miscalibration area (MA)** is the area between the calibration curve and the ideal calibration curve (the identity line from (0, 0) to (1, 1):

$$MA = \int_0^1 p_x - \hat{p}_x dx$$

Note that MA is equal to MACE as the number of bins (set by the `resolution` keyword argument) goes to infinity.

To choose which metric to compute, pass the name of the metric (`msce`, `rmsce`, `mace`, or `ma`) as the first argument to this function (or a list of them to compute multiple).

See Kuleshov et al., 2018, Chung et al., 2020 and the user guide page on *Evaluating Model Performance* for discussions of evaluating uncertainty estimates using calibration metrics, among other metrics. Note that calibration is generally less important than accuracy, but more important than other metrics like *`sharpness()`* and any *`dispersion_metric()`*.

**Parameters**

- **metric** (*`str {'msce', 'rmsce', 'mace', or 'ma'} or List[str]`*) – Which metric(s) to compute (see above for the definition of each metric). To compute multiple metrics, pass a list of the metric names you'd like to compute. Available metrics are:
  - `msce`: mean squared calibration error
  - `rmsce`: root mean squared calibration error
  - `mace`: mean absolute calibration error
  - `ma`: miscalibration area
- **x** (`ndarray` or `DataFrame` or `Series` or *`DataGenerator`*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *`DataGenerator`* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series`) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*`int`*) – Number of samples to draw from the model for computing the predictive percentile. Default = 1000
- **resolution** (*`int`*) – Number of confidence levels to evaluate at. This corresponds to the $m$ parameter in section 3.5 of (Kuleshov et al., 2018).
- **batch_size** (*`None or int`*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The requested calibration metric. If a list of metric names was passed, will return a dict whose keys are the metrics, and whose values are the corresponding metric values.

**Return type** float or Dict[str, float]

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute different calibration metrics using `expected_calibration_error()`. For example, to compute the mean squared calibration error (MSCE):

```
>>> model.calibration_metric("msce", x_val, y_val)
0.123
```

Or, to compute the mean absolute calibration error (MACE):

```
>>> model.calibration_metric("mace", x_val, y_val)
0.211
```

To compute multiple metrics at the same time, pass a list of metric names:

```
>>> model.calibration_metric(["msce", "mace"], x_val, y_val)
{"msce": 0.123, "mace": 0.211}
```

**See also:**

- *calibration_curve()*
- *calibration_curve_plot()*
- *sharpness()*
- *dispersion_metric()*

### References


- Volodymyr Kuleshov, Nathan Fenner, and Stefano Ermon. Accurate Uncertainties for Deep Learning Using Calibrated Regression, 2018.
- Youngseog Chung, Willie Neiswanger, Ian Char, Jeff Schneider. Beyond Pinball Loss: Quantile Methods for Calibrated Uncertainty Quantification, 2020.


**coverage_by**(*x_by*, *x*, *y=None*, *n: int = 1000*, *ci: float = 0.95*, *bins: int = 30*, *plot: bool = True*, *ideal_line_kwargs: dict = {}*, *batch_size=None*, ***kwargs*)

Compute and plot the coverage of a given confidence interval of the posterior predictive distribution as a function of specified independent variables.

TODO: Docs...

**Parameters**

- **x_by** (*int or str or list of int or list of str*) – Which independent variable(s) to plot the log probability as a function of. That is, which columns in `x` to plot by.
- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.

- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **ci** (*float between 0 and 1*) – Inner percentile to find the coverage of. For example, if `ci=0.95`, will compute the coverage of the inner 95% of the posterior predictive distribution.
- **bins** (*int*) – Number of bins to use for x_by
- **ideal_line_kwargs** (*dict*) – Dict of args to pass to matplotlib.pyplot.plot for ideal coverage line.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to plot_by

**Returns**

- **xo** (|*ndarray*|) – Values of x_by corresponding to bin centers.
- **co** (|*ndarray*|) – Coverage of the `ci` confidence interval of the predictive distribution in each bin.

**dispersion_metric**(*metric*, *x*, *n=1000*, *batch_size=None*)

Compute one or more of several calibration metrics

Dispersion metrics measure how much a model's uncertainty estimates vary. There are several different dispersion metrics:

The **coefficient of variation** ($C_v$) is the ratio of the standard deviation to the mean (of the model's uncertainty standard deviations):

$$C_v =$$

The **quartile coefficient of dispersion** ($QCD$) is less sensitive to outliers, as it simply measures the difference between the first and third quartile (of the model's uncertainty standard deviations) to their sum:

$$QCD = \frac{Q_3 - Q_1}{Q_3 + Q_1}$$

See Tran et al., 2020 and the user guide page on *Evaluating Model Performance* for discussions of evaluating uncertainty estimates using dispersion metrics, among other metrics. Note that dispersion metrics should generally be one of the last things you consider - accuracy, calibration, and sharpness usually being more important.

**Parameters**

- **metric** (*str {'cv' or 'qcd'} or List[str]*) – Dispersion metric to compute. Or,
- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **n** (*int*) – Number of samples to draw from the model. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The requested dispersion metric. If a list of metric names was passed, will return a dict whose keys are the metrics, and whose values are the corresponding metric values.

**Return type** float or Dict[str, float]

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute the coefficient of variation of our model's predictions with:

```
>>> model.dispersion_metric('cv', x_val)
0.732
```

Or the quartile coefficient of dispersion with:

```
>>> model.dispersion_metric('qcd', x_val)
0.625
```

**See also:**

- *calibration_metric()*
- *sharpness()*

### References


- Kevin Tran, Willie Neiswanger, Junwoong Yoon, Qingyang Zhang, Eric Xing, Zachary W. Ulissi. Methods for comparing uncertainty quantifications for material property predictions, 2020.


**dumps**()
Serialize module object to bytes

**elbo_loss**(*x_data*, *y_data*, *n: int*, *n_mc: int*)
Compute the negative ELBO, scaled to a single sample.

**Parameters**

- **x_data** – The independent variable values (or None if this is a generative model)
- **y_data** – The dependent variable values
- **n** (*int*) – Total number of datapoints in the dataset
- **n_mc** (*int*) – Number of MC samples we're taking from the posteriors

**epistemic_interval**(*x*, *ci=0.95*, *side='both'*, *n=1000*, *batch_size=None*)
Compute confidence intervals on the model's estimate of the target given `x`, including only epistemic uncertainty (uncertainty due to uncertainty as to the model's parameter values).

TODO: docs

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **ci** (*float between 0 and 1*) – Inner proportion of predictive distribution to use a the confidence interval. Default = 0.95

- **side** (*str {'lower', 'upper', 'both'}*) – Whether to get the one- or two-sided interval, and which side to get. If `'both'` (default), gets the upper and lower bounds of the central `ci` interval. If `'lower'`, gets the lower bound on the one-sided `ci` interval. If `'upper'`, gets the upper bound on the one-sided `ci` interval.
- **n** (*int*) – Number of samples from the epistemic predictive distribution to take to compute the confidence intervals. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **lb** (*|ndarray|*) – Lower bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='upper'`.
- **ub** (*|ndarray|*) – Upper bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='lower'`.

**epistemic_sample**(*x=None*, *n=1000*, *batch_size=None*)

Draw samples of the model's estimate given x, including only epistemic uncertainty (uncertainty due to uncertainty as to the model's parameter values)

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples, x.shape[0], ...)

**Return type** `ndarray`

**fit**(*x*, *y=None*, *batch_size: int = 128*, *epochs: int = 200*, *shuffle: bool = False*, *optimizer=None*, *optimizer_kwargs: dict = {}*, *lr: Optional[float] = None*, *flipout: bool = True*, *num_workers: Optional[int] = None*, *callbacks: List[probflow.utils.base.BaseCallback] = []*, *eager: bool = False*, *n_mc: int = 1*)

Fit the model to data

TODO

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values (or, if fitting a generative model, the dependent variable values). Should be of shape (Nsamples,...)
- **y** (`None` or `ndarray` or `DataFrame` or `Series`) – Dependent variable values (or, if fitting a generative model, `None`). Should be of shape (Nsamples,...). Default = `None`
- **batch_size** (*int*) – Number of samples to use per minibatch. Default = `128`
- **epochs** (*int*) – Number of epochs to train the model. Default = `200`
- **shuffle** (*bool*) – Whether to shuffle the data each epoch. Note that this is ignored if `x` is a *DataGenerator* Default = `True`
- **optimizer** (`None` or a backend-specific optimizer) – What optimizer to use for optimizing the variational posterior distributions' variables. When the backend is TensorFlow

the default is to use adam (`tf.keras.optimizers.Adam`). When the backend is PyTorch the default is to use TODO

- **optimizer_kwargs** (*dict*) – Keyword arguments to pass to the optimizer. Default is an empty dict.
- **lr** (*float*) – Learning rate for the optimizer. Note that the learning rate can be updated during training using the set_learning_rate method. Default is $\exp(-\log_{10}(N_p N_b))$, where $N_p$ is the number of parameters in the model, and $N_b$ is the number of samples per batch (`batch_size`).
- **flipout** (*bool*) – Whether to use flipout during training where possible Default = True
- **num_workers** (*None or int > 0*) – Number of parallel processes to run for loading the data. If `None`, will not use parallel processes. If an integer, will use a process pool with that many processes. Note that this parameter is ignored if a *DataGenerator* is passed as `x`. Default = None
- **callbacks** (*List[BaseCallback]*) – List of callbacks to run while training the model. Default is `[]`, i.e. no callbacks.
- **eager** (*bool*) – Whether to use eager execution. If False, will use `tf.function` (for TensorFlow) or tracing (for PyTorch) to optimize the model fitting. Note that even if eager=True, you can still use eager execution when using the model after it is fit. Default = False
- **n_mc** (*int*) – Number of monte carlo samples to take from the variational posteriors per minibatch. The default is to just take one per batch. Using a smaller number of MC samples is faster, but using a greater number of MC samples will decrease the variance of the gradients, leading to more stable parameter optimization.

### Example

See the user guide section on *Fitting a Model*.

**get_elbo**()
Get the current ELBO on training data

**kl_loss**()
Compute the sum of the Kullback-Leibler divergences between priors and their variational posteriors for all *Parameters* in this *Module* and its sub-Modules.

**kl_loss_batch**()
Compute the sum of additional Kullback-Leibler divergences due to data in this batch

**log_likelihood**(*x_data*, *y_data*)
Compute the sum log likelihood of the model given a batch of data

**log_prob**(*x*, *y=None*, *individually=True*, *distribution=False*, *n=1000*, *batch_size=None*)
Compute the log probability of *y* given the model

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor) – Independent variable values of the dataset to evaluate (aka the "features").
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").

- **individually** (*bool*) – If `individually` is True, returns log probability for each sample individually, so return shape is `(x.shape[0], ?)`. If `individually` is False, returns sum of all log probabilities, so return shape is `(1, ?)`.
- **distribution** (*bool*) – If `distribution` is True, returns log probability posterior distribution (`n` samples from the model), so return shape is `(?, n)`. If `distribution` is False, returns log posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is `(?, 1)`.
- **n** (*int*) – Number of samples to draw for each distribution if `distribution=True`.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns log_probs** – Log probabilities. Shape is determined by `individually`, `distribution`, and `n` kwargs.

**Return type** `ndarray`

**metric** (*metric*, *x*, *y=None*, *batch_size=None*)

Compute a metric of model performance

TODO: docs

TODO: note that this doesn't work w/ generative models

**Parameters**

- **metric** (*str or callable*) – Metric to evaluate. Available metrics:
  - 'lp': log likelihood sum
  - 'log_prob': log likelihood sum
  - 'accuracy': accuracy
  - 'acc': accuracy
  - 'mean_squared_error': mean squared error
  - 'mse': mean squared error
  - 'sum_squared_error': sum squared error
  - 'sse': sum squared error
  - 'mean_absolute_error': mean absolute error
  - 'mae': mean absolute error
  - 'r_squared': coefficient of determination
  - 'r2': coefficient of determination
  - 'recall': true positive rate
  - 'sensitivity': true positive rate
  - 'true_positive_rate': true positive rate
  - 'tpr': true positive rate
  - 'specificity': true negative rate
  - 'selectivity': true negative rate
  - 'true_negative_rate': true negative rate
  - 'tnr': true negative rate

- 'precision': precision
- 'f1_score': F-measure
- 'f1': F-measure
- callable: a function which takes (y_true, y_pred)

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* to generate both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

**Return type** TODO

**property modules**
A list of sub-Modules in this *Module*, including itself.

**property n_parameters**
Get the number of independent parameters of this module

**property n_variables**
Get the number of underlying variables in this module

**property parameters**
A list of *Parameters* in this *Module* and its sub-Modules.

**posterior_ci**(*params=None*, *ci=0.95*, *n=10000*)
Posterior confidence intervals

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get the confidence intervals for all parameters in the model.
- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95
- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals Default = 10,000

**Returns** Confidence intervals of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain tuples. The first element of each tuple is the lower bound, and the second element is the upper bound. Or just a single tuple if params was a str

**Return type** dict

**posterior_mean**(*params=None*)
Get the mean of the posterior distribution(s)

TODO: Docs... params is a list of strings of params to plot

**Parameters** **params** (*str or List[str] or None*) – Parameter name(s) for which to compute the means. Default is to get the mean for all parameters in the model.

**Returns** Means of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the posterior means. The `ndarrays` are the same size as each parameter. Or just the `ndarray` if `params` was a str.

**Return type** dict

**posterior_plot** (*params=None*, *cols=1*, ***kwargs*)
Plot posterior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the posterior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.posterior_plot()*

**posterior_sample** (*params=None*, *n=10000*)
Draw samples from parameter posteriors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get a sample for all parameters in the model.
- **num_samples** (*int*) – Number of samples to take from each posterior distribution. Default = 1000

**Returns** Samples from the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the posterior samples. The `ndarrays` are of size (`num_samples`, param.shape). Or just the `ndarray` if `params` was a str.

**Return type** dict

**pred_dist_coverage** (*x*, *y=None*, *n=1000*, *ci=0.95*, *batch_size=None*)
Compute what percent of samples are covered by a given confidence interval.

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model given `x`. Default = 1000
- **ci** (*float between 0 and 1*) – Confidence interval to use.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** **prc_covered** – Proportion of the samples which were covered by the predictive distribution's confidence interval.

**Return type** float between 0 and 1

**pred_dist_covered**(*x*, *y=None*, *n: int = 1000*, *ci: float = 0.95*, *batch_size=None*)

Compute whether each observation was covered by a given confidence interval.

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model given x. Default = 1000
- **ci** (*float between 0 and 1*) – Confidence interval to use.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

**Return type** TODO

**predict**(*x=None*, *method='mean'*, *batch_size=None*)

Predict dependent variable using the model

TODO... using maximum a posteriori param estimates etc

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **method** (*str*) – Method to use for prediction. If 'mean', uses the mean of the predicted target distribution as the prediction. If 'mode', uses the mode of the distribution.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Predicted y-value for each sample in x. Of size (x.shape[0], y.shape[0], ..., y.shape[-1])

**Return type** ndarray

### Examples

TODO: Docs...

**predictive_interval**(*x*, *ci=0.95*, *side='both'*, *n=1000*, *batch_size=None*)

Compute confidence intervals on the model's estimate of the target given x, including all sources of uncertainty.

TODO: docs

TODO: using side= both, upper, vs lower

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").

- **ci** (*float between 0 and 1*) – Inner proportion of predictive distribution to use a the confidence interval. Default = 0.95
- **side** (*str {'lower', 'upper', 'both'}*) – Whether to get the one- or two-sided interval, and which side to get. If `'both'` (default), gets the upper and lower bounds of the central `ci` interval. If `'lower'`, gets the lower bound on the one-sided `ci` interval. If `'upper'`, gets the upper bound on the one-sided `ci` interval.
- **n** (*int*) – Number of samples from the posterior predictive distribution to take to compute the confidence intervals. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **lb** (|*ndarray*|) – Lower bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='upper'`.
- **ub** (|*ndarray*|) – Upper bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='lower'`.

**predictive_prc**(*x*, *y=None*, *n=1000*, *batch_size=None*)

Compute the percentile of each observation along the posterior predictive distribution.

TODO: Docs... Returns a percentile between 0 and 1

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model given `x`. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** prcs

**Return type** `ndarray` of float between 0 and 1

**predictive_sample**(*x=None*, *n=1000*, *batch_size=None*)

Draw samples from the posterior predictive distribution given x

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predictive distribution. Size (num_samples, x.shape[0], ...)

**Return type** `ndarray`

**prior_plot** (*params=None*, *cols=1*, ***kwargs*)
Plot prior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the prior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.prior_plot()*

**prior_sample** (*params=None*, *n=10000*)
Draw samples from parameter priors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*list*) – List of parameter names to sample. Each element should be a str. Default is to sample priors of all parameters in the model.
- **n** (*int*) – Number of samples to take from each prior distribution. Default = 10000

**Returns** Samples from the parameter prior distributions. A dictionary where the keys contain the parameter names and the values contain ndarrays with the prior samples. The ndarrays are of size (n,param.shape).

**Return type** dict

**prob** (*x*, *y=None*, ***kwargs*)
Compute the probability of y given the model

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **individually** (*bool*) – If individually is True, returns probability for each sample individually, so return shape is (x.shape[0], ?). If individually is False, returns product of all probabilities, so return shape is (1, ?).
- **distribution** (*bool*) – If distribution is True, returns posterior probability distribution (n samples from the model), so return shape is (?, n). If distribution is False, returns posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is (?, 1).
- **n** (*int*) – Number of samples to draw for each distribution if distribution=True.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** **probs** – Probabilities. Shape is determined by individually, distribution, and n kwargs.

**Return type** ndarray

**reset_kl_loss**()
Reset additional loss due to KL divergences

**residuals**(*x*, *y=None*, *batch_size=None*)
Compute the residuals of the model's predictions.

TODO: docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The residuals.

**Return type** ndarray

### Example

TODO

**residuals_plot**(*x*, *y=None*, *batch_size=None*, ***kwargs*)
Plot the distribution of residuals of the model's predictions.

TODO: docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_dist()*

### Example

TODO

**save**(*filename: str*)
Save module object to file

**Parameters** **filename** (*str*) – Filename for file to which to save this object

**set_kl_weight**(*w*)
Set the weight of the KL term's contribution to the ELBO loss

**set_learning_rate**(*lr*)
Set the learning rate used by this model's optimizer

**sharpness**(*x*, *n=1000*, *batch_size=None*)

Compute the sharpness of the model's uncertainty estimates

The "sharpness" of a model's uncertainty estimates is the root mean of the estimated variances:

$$SHA = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \mathrm{Var}(\hat{Y}_i)}$$

See Tran et al., 2020 and the user guide page on *Evaluating Model Performance* for discussions of evaluating uncertainty estimates using sharpness, among other metrics. Note that the sharpness should generally be one of the later things you consider - accuracy and calibration usually being more important.

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **n** (*int*) – Number of samples to draw from the model. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The sharpness of the model's uncertainty estimates

**Return type** float

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute the sharpness of our model's predictions with:

```
>>> model.sharpness(x_val)
0.173
```

**See also:**

- *calibration_metric()*
- *dispersion_metric()*

### References


- Kevin Tran, Willie Neiswanger, Junwoong Yoon, Qingyang Zhang, Eric Xing, Zachary W. Ulissi. Methods for comparing uncertainty quantifications for material property predictions, 2020.


**stop_training**()

Stop the training of the model

**summary**()

Show a summary of the model and its parameters.

TODO

TODO: though maybe this should be a method of module... model would have to add to it the observation dist

**train_step**(*x_data*, *y_data*)
Perform one training step

**property trainable_variables**
A list of trainable backend variables within this *Module*

**class** probflow.models.**CategoricalModel**(**args*)
Bases: probflow.models.model.Model

Abstract base class for probflow models where the dependent variable (the target) is categorical (e.g. drawn from a Bernoulli distribution).

TODO : why use this over just Model

This class inherits several methods from *Module*:

- *parameters*
- *modules*
- *trainable_variables*
- *kl_loss()*
- *kl_loss_batch()*
- *reset_kl_loss()*
- *add_kl_loss()*

as well as several methods from *Model*:

- *log_likelihood()*
- *train_step()*
- *fit()*
- *stop_training()*
- *set_learning_rate()*
- *predictive_sample()*
- *aleatoric_sample()*
- *epistemic_sample()*
- *predict()*
- *metric()*
- *posterior_mean()*
- *posterior_sample()*
- *posterior_ci()*
- *prior_sample()*
- *posterior_plot()*
- *prior_plot()*
- *log_prob()*
- *prob()*

- *save()*
- *summary()*

and adds the following categorical-model-specific methods:

- *pred_dist_plot()*
- *calibration_curve()*

### Example

TODO

**pred_dist_plot**(*x*, *n=10000*, *cols=1*, *batch_size=None*, ***kwargs*)
Plot posterior predictive distribution from the model given `x`.

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model given `x`. Default = 10000
- **cols** (*int*) – Divide the subplots into a grid with this many columns (if `individually=True`.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_categorical_dist()*

**calibration_curve**(*x*, *y=None*, *split_by=None*, *bins=10*, *plot=True*, *batch_size=None*)
Plot and return the categorical calibration curve.

Plots and returns the calibration curve (estimated probability of outcome vs the true probability of that outcome).

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **split_by** (*int*) – Draw the calibration curve independently for datapoints with each unique value in *x[:,split_by]* (a categorical column).
- **bins** (int, list of float, or ndarray) – Bins used to compute the curve. If an integer, will use *bins* evenly-spaced bins from 0 to 1. If a vector, *bins* is the vector of bin edges.
- **plot** (*bool*) – Whether to plot the curve
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- **#TODO** (*split by continuous cols as well? Then will need to define bins or edges too*) –
- **TODO** (*Docs...*) –

**add_kl_loss**(*loss*, *d2=None*)
Add additional loss due to KL divergences.

**aleatoric_sample**(*x=None*, *n=1000*, *batch_size=None*)
Draw samples of the model's estimate given x, including only aleatoric uncertainty (uncertainty due to noise)

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples,x.shape[0],...)

**Return type** ndarray

**bayesian_update**()
Perform a Bayesian update of all *Parameters* in this module. Sets the prior to the current variational posterior for all parameters.

**dumps**()
Serialize module object to bytes

**elbo_loss**(*x_data*, *y_data*, *n: int*, *n_mc: int*)
Compute the negative ELBO, scaled to a single sample.

**Parameters**

- **x_data** – The independent variable values (or None if this is a generative model)
- **y_data** – The dependent variable values
- **n** (*int*) – Total number of datapoints in the dataset
- **n_mc** (*int*) – Number of MC samples we're taking from the posteriors

**epistemic_sample**(*x=None*, *n=1000*, *batch_size=None*)
Draw samples of the model's estimate given x, including only epistemic uncertainty (uncertainty due to uncertainty as to the model's parameter values)

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples, x.shape[0], ...)

**Return type** ndarray

**fit** (*x*, *y=None*, *batch_size: int = 128*, *epochs: int = 200*, *shuffle: bool = False*, *optimizer=None*, *optimizer_kwargs: dict = {}*, *lr: Optional[float] = None*, *flipout: bool = True*, *num_workers: Optional[int] = None*, *callbacks: List[probflow.utils.base.BaseCallback] = []*, *eager: bool = False*, *n_mc: int = 1*)

Fit the model to data

TODO

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *`DataGenerator`*) – Independent variable values (or, if fitting a generative model, the dependent variable values). Should be of shape (Nsamples,...)
- **y** (`None` or `ndarray` or `DataFrame` or `Series`) – Dependent variable values (or, if fitting a generative model, `None`). Should be of shape (Nsamples,...). Default = `None`
- **batch_size** (*`int`*) – Number of samples to use per minibatch. Default = `128`
- **epochs** (*`int`*) – Number of epochs to train the model. Default = `200`
- **shuffle** (*`bool`*) – Whether to shuffle the data each epoch. Note that this is ignored if `x` is a *`DataGenerator`* Default = `True`
- **optimizer** (`None` or a backend-specific optimizer) – What optimizer to use for optimizing the variational posterior distributions' variables. When the backend is TensorFlow the default is to use adam (`tf.keras.optimizers.Adam`). When the backend is PyTorch the default is to use TODO
- **optimizer_kwargs** (*`dict`*) – Keyword arguments to pass to the optimizer. Default is an empty dict.
- **lr** (*`float`*) – Learning rate for the optimizer. Note that the learning rate can be updated during training using the set_learning_rate method. Default is $\exp(-\log_{10}(N_p N_b))$, where $N_p$ is the number of parameters in the model, and $N_b$ is the number of samples per batch (`batch_size`).
- **flipout** (*`bool`*) – Whether to use flipout during training where possible Default = True
- **num_workers** (*`None or int > 0`*) – Number of parallel processes to run for loading the data. If `None`, will not use parallel processes. If an integer, will use a process pool with that many processes. Note that this parameter is ignored if a *`DataGenerator`* is passed as `x`. Default = None
- **callbacks** (*`List[BaseCallback]`*) – List of callbacks to run while training the model. Default is `[]`, i.e. no callbacks.
- **eager** (*`bool`*) – Whether to use eager execution. If False, will use `tf.function` (for TensorFlow) or tracing (for PyTorch) to optimize the model fitting. Note that even if eager=True, you can still use eager execution when using the model after it is fit. Default = False
- **n_mc** (*`int`*) – Number of monte carlo samples to take from the variational posteriors per minibatch. The default is to just take one per batch. Using a smaller number of MC samples is faster, but using a greater number of MC samples will decrease the variance of the gradients, leading to more stable parameter optimization.

### Example

See the user guide section on *Fitting a Model*.

**get_elbo**()
Get the current ELBO on training data

**kl_loss**()
Compute the sum of the Kullback-Leibler divergences between priors and their variational posteriors for all *Parameters* in this *Module* and its sub-Modules.

**kl_loss_batch**()
Compute the sum of additional Kullback-Leibler divergences due to data in this batch

**log_likelihood**(*x_data*, *y_data*)
Compute the sum log likelihood of the model given a batch of data

**log_prob**(*x*, *y=None*, *individually=True*, *distribution=False*, *n=1000*, *batch_size=None*)
Compute the log probability of *y* given the model

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor) – Independent variable values of the dataset to evaluate (aka the "features").
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **individually** (*bool*) – If individually is True, returns log probability for each sample individually, so return shape is (x.shape[0], ?). If individually is False, returns sum of all log probabilities, so return shape is (1, ?).
- **distribution** (*bool*) – If distribution is True, returns log probability posterior distribution (n samples from the model), so return shape is (?, n). If distribution is False, returns log posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is (?, 1).
- **n** (*int*) – Number of samples to draw for each distribution if distribution=True.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns log_probs** – Log probabilities. Shape is determined by individually, distribution, and n kwargs.

**Return type** ndarray

**metric**(*metric*, *x*, *y=None*, *batch_size=None*)
Compute a metric of model performance

TODO: docs

TODO: note that this doesn't work w/ generative models

**Parameters**

- **metric** (*str or callable*) – Metric to evaluate. Available metrics:
  - 'lp': log likelihood sum
  - 'log_prob': log likelihood sum
  - 'accuracy': accuracy

- 'acc': accuracy
- 'mean_squared_error': mean squared error
- 'mse': mean squared error
- 'sum_squared_error': sum squared error
- 'sse': sum squared error
- 'mean_absolute_error': mean absolute error
- 'mae': mean absolute error
- 'r_squared': coefficient of determination
- 'r2': coefficient of determination
- 'recall': true positive rate
- 'sensitivity': true positive rate
- 'true_positive_rate': true positive rate
- 'tpr': true positive rate
- 'specificity': true negative rate
- 'selectivity': true negative rate
- 'true_negative_rate': true negative rate
- 'tnr': true negative rate
- 'precision': precision
- 'f1_score': F-measure
- 'f1': F-measure
- callable: a function which takes (y_true, y_pred)

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* to generate both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

**Return type** TODO

**property modules**
A list of sub-Modules in this *Module*, including itself.

**property n_parameters**
Get the number of independent parameters of this module

**property n_variables**
Get the number of underlying variables in this module

**property parameters**
A list of *Parameters* in this *Module* and its sub-Modules.

**posterior_ci** (*params=None*, *ci=0.95*, *n=10000*)
Posterior confidence intervals

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get the confidence intervals for all parameters in the model.
- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95
- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals Default = 10,000

**Returns** Confidence intervals of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain tuples. The first element of each tuple is the lower bound, and the second element is the upper bound. Or just a single tuple if params was a str

**Return type** dict

**posterior_mean** (*params=None*)
Get the mean of the posterior distribution(s)

TODO: Docs... params is a list of strings of params to plot

**Parameters** **params** (*str or List[str] or None*) – Parameter name(s) for which to compute the means. Default is to get the mean for all parameters in the model.

**Returns** Means of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the posterior means. The `ndarrays` are the same size as each parameter. Or just the `ndarray` if `params` was a str.

**Return type** dict

**posterior_plot** (*params=None*, *cols=1*, ***kwargs*)
Plot posterior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the posterior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.posterior_plot()*

**posterior_sample** (*params=None*, *n=10000*)
Draw samples from parameter posteriors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get a sample for all parameters in the model.
- **num_samples** (*int*) – Number of samples to take from each posterior distribution. Default = 1000

**Returns** Samples from the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the posterior samples. The `ndarrays` are of size (`num_samples`, param.shape). Or just the `ndarray` if `params` was a str.

**Return type** dict

**predict** (*x=None*, *method='mean'*, *batch_size=None*)

Predict dependent variable using the model

TODO... using maximum a posteriori param estimates etc

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **method** (*str*) – Method to use for prediction. If `'mean'`, uses the mean of the predicted target distribution as the prediction. If `'mode'`, uses the mode of the distribution.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Predicted y-value for each sample in `x`. Of size (x.shape[0], y.shape[0], ..., y.shape[-1])

**Return type** `ndarray`

### Examples

TODO: Docs...

**predictive_sample** (*x=None*, *n=1000*, *batch_size=None*)

Draw samples from the posterior predictive distribution given x

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predictive distribution. Size (num_samples, x.shape[0], ...)

**Return type** `ndarray`

**prior_plot** (*params=None*, *cols=1*, ***kwargs*)

Plot prior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the prior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.prior_plot()*

**prior_sample**(*params=None*, *n=10000*)
Draw samples from parameter priors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*list*) – List of parameter names to sample. Each element should be a str. Default is to sample priors of all parameters in the model.
- **n** (*int*) – Number of samples to take from each prior distribution. Default = 10000

**Returns** Samples from the parameter prior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the prior samples. The `ndarrays` are of size (`n`,param.shape).

**Return type** dict

**prob**(*x*, *y=None*, ***kwargs*)
Compute the probability of `y` given the model

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **individually** (*bool*) – If `individually` is True, returns probability for each sample individually, so return shape is `(x.shape[0], ?)`. If `individually` is False, returns product of all probabilities, so return shape is `(1, ?)`.
- **distribution** (*bool*) – If `distribution` is True, returns posterior probability distribution (`n` samples from the model), so return shape is `(?, n)`. If `distribution` is False, returns posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is `(?, 1)`.
- **n** (*int*) – Number of samples to draw for each distribution if `distribution=True`.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** **probs** – Probabilities. Shape is determined by `individually`, `distribution`, and `n` kwargs.

**Return type** `ndarray`

**reset_kl_loss**()
Reset additional loss due to KL divergences

**save**(*filename: str*)
Save module object to file

**Parameters** **filename** (*str*) – Filename for file to which to save this object

**set_kl_weight**(*w*)
Set the weight of the KL term's contribution to the ELBO loss

**set_learning_rate**(*lr*)
Set the learning rate used by this model's optimizer

**stop_training**()
Stop the training of the model

**summary**()
Show a summary of the model and its parameters.

TODO

TODO: though maybe this should be a method of module... model would have to add to it the observation dist

**train_step**(*x_data*, *y_data*)
Perform one training step

**property trainable_variables**
A list of trainable backend variables within this *Module*

## 3.5 Callbacks

The callbacks module contains classes for monitoring and adjusting the training process.

- *Callback* - abstract base class for all callbacks
- *EarlyStopping* - stop training if some metric stops improving
- *KLWeightScheduler* - set the KL weight by epoch
- *LearningRateScheduler* - set the learning rate by epoch
- *MonitorELBO* - record the ELBO loss over the course of training
- *MonitorMetric* - record a metric over the course of training
- *MonitorParameter* - record a parameter over the course of training
- *TimeOut* - stop training after a certain amount of time

---

**class** probflow.callbacks.**Callback**(**args*)
Bases: *probflow.utils.base.BaseCallback*

Base class for all callbacks.

See the user guide section on *Callbacks*.

**on_train_start**()
Will be called at the start of training. By default does nothing.

**on_epoch_start**()
Will be called at the start of each training epoch. By default does nothing.

**on_epoch_end**()
Will be called at the end of each training epoch. By default does nothing.

**on_train_end**()
Will be called at the end of training. By default does nothing.

**class** probflow.callbacks.**EarlyStopping**(*metric_fn*, *patience=0*, *verbose=True*, *name='EarlyStopping'*)
Bases: probflow.callbacks.callback.Callback

Stop training early when some metric stops decreasing

**Parameters**

- **metric_fn** (*callable, MonitorMetric, or MonitorELBO*) – Any arbitrary function, or a *MonitorMetric* or *MonitorELBO* callback. Training will be stopped when the value returned by that function stops decreasing (or, if `metric_fn` was a *MonitorMetric* or *MonitorELBO*, training is stopped when the metric being monitored or the ELBO stops decreasing.
- **patience** (*int*) – Number of epochs to allow training to continue even if metric is not decreasing. Default is 0.
- **restore_best_weights** (*bool*) – Whether or not to restore the weights from the best epoch after training is stopped. Default = False.
- **verbose** (*bool*) – Whether to print when training was stopped. Default = False
- **name** (*str*) – Name for this callback

### Example

See the user guide section on *Ending training when a metric stops improving*.

**on_epoch_end**()
Stop training if there was no improvement since the last epoch.

**on_epoch_start**()
Will be called at the start of each training epoch. By default does nothing.

**on_train_end**()
Will be called at the end of training. By default does nothing.

**on_train_start**()
Will be called at the start of training. By default does nothing.

**class** probflow.callbacks.**KLWeightScheduler**(*fn*, *verbose=False*)
Bases: probflow.callbacks.callback.Callback

Set the weight of the KL term's contribution to the ELBO loss each epoch

**Parameters**

- **fn** (*callable*) – Function which takes the current epoch as an argument and returns a kl weight, a float between 0 and 1
- **verbose** (*bool*) – Whether to print the KL weight each epoch (if True) or not (if False). Default = False

### Examples

See the user guide section on *Changing the KL weight over training*.

**on_epoch_start**()
Set the KL weight at the start of each epoch.

**plot**(***kwargs*)
Plot the KL weight as a function of epoch

**Parameters** ****kwargs** – Additional keyword arguments are passed to plt.plot

**on_epoch_end**()
Will be called at the end of each training epoch. By default does nothing.

**on_train_end**()
Will be called at the end of training. By default does nothing.

**on_train_start**()
Will be called at the start of training. By default does nothing.

**class** probflow.callbacks.**LearningRateScheduler**(*fn*, *verbose: bool = False*)
Bases: probflow.callbacks.callback.Callback

Set the learning rate as a function of the current epoch

**Parameters**

- **fn** (*callable*) – Function which takes the current epoch as an argument and returns a learning rate.
- **verbose** (*bool*) – Whether to print the learning rate each epoch (if True) or not (if False). Default = False

### Examples

See the user guide section on *Changing the learning rate over training*. training`.

**on_epoch_start**()
Set the learning rate at the start of each epoch.

**plot**(***kwargs*)
Plot the learning rate as a function of epoch

**Parameters** ****kwargs** – Additional keyword arguments are passed to matplotlib.pyplot.plot

**on_epoch_end**()
Will be called at the end of each training epoch. By default does nothing.

**on_train_end**()
Will be called at the end of training. By default does nothing.

**on_train_start**()
Will be called at the start of training. By default does nothing.

**class** probflow.callbacks.**MonitorELBO**(*verbose=False*)
Bases: probflow.callbacks.callback.Callback

Monitor the ELBO on the training data

**Parameters** **verbose** (*bool*) – Whether to print the average ELBO at the end of every training epoch (if True) or not (if False). Default = False

### Example

See the user guide section on *Monitoring the loss*.

**on_epoch_start**()
Record start time at the beginning of the first epoch

**on_epoch_end**()
Store the ELBO at the end of each epoch.

**plot**(*x='epoch'*, ***kwargs*)
Plot the ELBO as a function of epoch

**Parameters**

- **x** (*str {'epoch' or 'time'}*) – Whether to plot the metric as a function of epoch or wall time Default is to plot by epoch.
- ****kwargs** – Additional keyword arguments are passed to plt.plot

**on_train_end**()
Will be called at the end of training. By default does nothing.

**on_train_start**()
Will be called at the start of training. By default does nothing.

**class** probflow.callbacks.**MonitorMetric**(*metric*, *x*, *y=None*, *verbose=False*)
Bases: probflow.callbacks.callback.Callback

Monitor some metric on validation data

**Parameters**

- **metric** (*str*) – Name of the metric to evaluate. See *Model.metric()* for a list of available metrics.
- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the validation dataset to evaluate (aka the "features"). Or a *DataGenerator* to generate both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the validation dataset to evaluate (aka the "target").
- **verbose** (*bool*) – Whether to print the average ELBO at the end of every training epoch (if True) or not (if False). Default = False

### Example

See the user guide section on *Monitoring a metric*.

**on_epoch_start**()
Record start time at the beginning of the first epoch

**on_epoch_end**()
Compute the metric on validation data at the end of each epoch.

**on_train_end**()
Will be called at the end of training. By default does nothing.

**on_train_start**()
Will be called at the start of training. By default does nothing.

**plot**(*x='epoch'*, ***kwargs*)
Plot the metric being monitored as a function of epoch

**Parameters**

- **x** (*str {'epoch' or 'time'}*) – Whether to plot the metric as a function of epoch or wall time. Default is to plot by epoch.
- ****kwargs** – Additional keyword arguments are passed to plt.plot

**class** probflow.callbacks.**MonitorParameter**(*params*)
Bases: probflow.callbacks.callback.Callback

Monitor the mean value of Parameter(s) over the course of training

**Parameters** **params** (*str or List[str] or None*) – Name(s) of the parameters to monitor.

### Examples

See the user guide section on *Monitoring the value of parameter(s)*.

**on_epoch_end**()
Store mean values of Parameter(s) at the end of each epoch.

**plot**(*param=None*, ***kwargs*)
Plot the parameter value(s) as a function of epoch

**Parameters** **param** (*None or str*) – Parameter to plot. If None, assumes we've only been monitoring one parameter and plots that. If a str, plots the parameter with that name (assuming we've been monitoring it).

**on_epoch_start**()
Will be called at the start of each training epoch. By default does nothing.

**on_train_end**()
Will be called at the end of training. By default does nothing.

**on_train_start**()
Will be called at the start of training. By default does nothing.

**class** probflow.callbacks.**TimeOut**(*time_limit*, *verbose=True*)
Bases: probflow.callbacks.callback.Callback

Stop training after a certain amount of time

**Parameters**

- **time_limit** (*float or int*) – Number of seconds after which to stop training
- **verbose** (*bool*) – Whether to print that we stopped training early (if True) or not (if False). Default = False

### Example

Stop training after five hours:

```
time_out = pf.callbacks.TimeOut(5*60*60)
model.fit(x, y, callbacks=[time_out])
```

**on_epoch_start**()
Record start time at the beginning of the first epoch

**on_epoch_end**()
Stop training if time limit has been passed

**on_train_end**()
Will be called at the end of training. By default does nothing.

**on_train_start**()
Will be called at the start of training. By default does nothing.

## 3.6 Data

TODO: Data utilities, more info...

- *DataGenerator* - base class for data generators w/ multiprocessing
- *ArrayDataGenerator* - data generator for array-structured data

---

**class** probflow.data.**DataGenerator**(*num_workers=None*)

Bases: *probflow.utils.base.BaseDataGenerator*

Abstract base class for a data generator, which uses multiprocessing to load the data in parallel.

TODO

User needs to implement:

- `__init__()`
- *n_samples()*
- *batch_size()*
- *get_batch()*

And can optionally implement:

- *on_epoch_start()*
- *on_epoch_end()*

**abstract get_batch**(*index*)
Generate one batch of data

**abstract property batch_size**
Number of samples to generate each minibatch

**abstract property n_samples**
Number of samples in the dataset

**on_epoch_end**()
Will be called at the end of each training epoch

**on_epoch_start**()
Will be called at the start of each training epoch

**class** probflow.data.**ArrayDataGenerator**(*x=None*, *y=None*, *batch_size=None*, *shuffle=False*, *test=False*, *num_workers=None*)

Bases: probflow.data.data_generator.DataGenerator

Generate array-structured data to feed through a model.

TODO

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values (or, if fitting a generative model, the dependent variable values). Should be of shape (Nsamples,...)
- **y** (None or ndarray or DataFrame or Series) – Dependent variable values (or, if fitting a generative model, None). Should be of shape (Nsamples,...). Default = None
- **batch_size** (*int*) – Number of samples to use per minibatch. Use None to use a single batch for all the data. Default = None

- **shuffle** (*bool*) – Whether to shuffle the data each epoch. Default = `False`
- **testing** (*bool*) – Whether to treat data as testing data (allow no dependent variable). Default = `False`

**on_epoch_start**()
Will be called at the start of each training epoch

**property n_samples**
Number of samples in the dataset

**property batch_size**
Number of samples to generate each minibatch

**get_batch**(*index*)
Generate one batch of data

**on_epoch_end**()
Shuffle data each epoch

probflow.data.**make_generator**(*x=None*, *y=None*, *batch_size=None*, *shuffle=False*, *test=False*, *num_workers=None*)
Make input a DataGenerator if not already

## 3.7 Applications

The applications module contains pre-built *Models*

- *LinearRegression* - a linear regression model
- *LogisticRegression* - a Bi- or Multinomial logistic regression model
- *PoissonRegression* - a Poisson regression model
- *DenseRegression* - a multi-layer dense neural net regression model
- *DenseClassifier* - a multi-layer dense neural net classifier model

---

**class** probflow.applications.**LinearRegression**(*d: int*, *d_o: int = 1*, *heteroscedastic: bool = False*)
Bases: `probflow.models.continuous_model.ContinuousModel`

A multiple linear regression

TODO: explain, math, diagram, examples, etc

**Parameters**

- **d** (*int*) – Dimensionality of the independent variable (number of features)
- **d_o** (*int*) – Dimensionality of the dependent variable (number of target dimensions)
- **heteroscedastic** (*bool*) – Whether to model a change in noise as a function of **x** (if `heteroscedastic=True`), or not (if `heteroscedastic=False`, the default).

**weights**
Regression weights

**Type** *Parameter*

**bias**
Regression intercept

**Type** *Parameter*

**std**
Standard deviation of the Normal observation distribution

**Type** *ScaleParameter*

**add_kl_loss**(*loss*, *d2=None*)
Add additional loss due to KL divergences.

**aleatoric_interval**(*x*, *ci=0.95*, *side='both'*, *n=1000*, *batch_size=None*)
Compute confidence intervals on the model's estimate of the target given `x`, including only aleatoric uncertainty (uncertainty due to noise).

TODO: docs

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **ci** (*float between 0 and 1*) – Inner proportion of predictive distribution to use a the confidence interval. Default = 0.95
- **side** (*str {'lower', 'upper', 'both'}*) – Whether to get the one- or two-sided interval, and which side to get. If `'both'` (default), gets the upper and lower bounds of the central `ci` interval. If `'lower'`, gets the lower bound on the one-sided `ci` interval. If `'upper'`, gets the upper bound on the one-sided `ci` interval.
- **n** (*int*) – Number of samples from the aleatoric predictive distribution to take to compute the confidence intervals. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **lb** (|*ndarray*|) – Lower bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='upper'`.
- **ub** (|*ndarray*|) – Upper bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='lower'`.

**aleatoric_sample**(*x=None*, *n=1000*, *batch_size=None*)
Draw samples of the model's estimate given x, including only aleatoric uncertainty (uncertainty due to noise)

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples,x.shape[0],...)

**Return type** `ndarray`

**bayesian_update**()
Perform a Bayesian update of all *Parameters* in this module. Sets the prior to the current variational posterior for all parameters.

**calibration_curve**(*x*, *y*, *n=1000*, *resolution=100*, *batch_size=None*)
Compute the regression calibration curve (Kuleshov et al., 2018).

The regression calibration curve compares the empirical cumulative probability to the cumulative probability predicted by a regression model (Kuleshov et al., 2018). First, a vector $p$ of $m$ confidence levels are chosen, which correspond to the predicted cumulative probabilities:

$$0 \leq p_1 \leq p_2 \leq \ldots \leq p_m \leq 1$$

Then, a vector of empirical frequencies $\hat{p}$ at each of the predicted frequencies is computed by using validation data:

$$\hat{p}_j = \frac{1}{N} \sum_{i=1}^{N} [P_M(x_i \leq y_i) \leq p_j]$$

where $N$ is the number of validation datapoints, $P_M(x_i \leq y_i)$ is the model's predicted cumulative probability of datapoint $i$ (i.e., the percentile along the model's predicted probability distribution at which the true value of $y_i$ falls), and $\sum_i [a_i \leq b_i]$ is just the count of elements of $a$ which are less than corresponding elements in $b$.

The calibration curve then plots $p$ against $\hat{p}$.

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model for computing the predictive percentile. Default = 1000
- **resolution** (*int*) – Number of confidence levels to evaluate at. This corresponds to the $m$ parameter in section 3.5 of (Kuleshov et al., 2018).
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **p** (|*ndarray*|) – The predicted cumulative frequencies, $p$.
- **p_hat** (|*ndarray*|) – The empirical cumulative frequencies, $\hat{p}$.

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute the calibration curve with *calibration_curve()*:

```
p_pred, p_empirical = model.calibration_curve(x_val, y_val)
```

The returned values can be used directly or plotted against one another to get the calibration curve (as in Figure 3 in Kuleshov et al., 2018)

```
import matplotlib.pyplot as plt
plt.plot(p_pred, p_empirical)
```

Or, even more simply, just use *calibration_curve_plot()*.

**See also:**

- *calibration_curve_plot()*
- expected_calibration_error()

### References


- Volodymyr Kuleshov, Nathan Fenner, and Stefano Ermon. Accurate Uncertainties for Deep Learning Using Calibrated Regression, 2018.


**calibration_curve_plot** (*x*, *y*, *n=1000*, *resolution=100*, *batch_size=None*, ***kwargs*)
Plot the regression calibration curve.

See *calibration_curve()* for more info about the regression calibration curve.

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model for computing the predictive percentile. Default = 1000
- **resolution** (*int*) – Number of confidence levels to evaluate at. This corresponds to the $m$ parameter in section 3.5 of (Kuleshov et al., 2018).
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_dist()*

**See also:**

- *calibration_curve()*
- expected_calibration_error()

**calibration_metric** (*metric*, *x*, *y=None*, *n=1000*, *resolution=100*, *batch_size=None*)
Compute one or more of several calibration metrics

Regression calibration metrics measure the error between a model's regression calibration curve and the ideal calibration curve - i.e., what the curve would be if the model were perfectly calibrated (see Kuleshov

et al., 2018 and Chung et al., 2020). First, a vector $p$ of $m$ confidence levels are chosen, which correspond to the predicted cumulative probabilities:

$$0 \leq p_1 \leq p_2 \leq \ldots \leq p_m \leq 1$$

Then, a vector of empirical frequencies $\hat{p}$ at each of the predicted frequencies is computed by using validation data:

$$\hat{p}_j = \frac{1}{N} \sum_{i=1}^{N} [P_M(x_i \leq y_i) \leq p_j]$$

where $N$ is the number of validation datapoints, $P_M(x_i \leq y_i)$ is the model's predicted cumulative probability of datapoint $i$ (i.e., the percentile along the model's predicted probability distribution at which the true value of $y_i$ falls), and $\sum_i [a_i \leq b_i]$ is just the count of elements of $a$ which are less than corresponding elements in $b$.

Various metrics can be computed from these curves to measure how accurately the regression model captures uncertainty:

The **mean squared calibration error (MSCE)** is the mean squared error between the empirical and predicted frequencies,

$$MSCE = \frac{1}{m} \sum_{j=1}^{m} (p_j - \hat{p}_j)^2$$

The **root mean squared calibration error (RMSCE)** is just the square root of the MSCE:

$$RMSCE = \sqrt{\frac{1}{m} \sum_{j=1}^{m} (p_j - \hat{p}_j)^2}$$

The **mean absolute calibration error (MACE)** is the mean of the absolute differences between the empirical and predicted frequencies:

$$MACE = \frac{1}{m} \sum_{j=1}^{m} |p_j - \hat{p}_j|$$

And the **miscalibration area (MA)** is the area between the calibration curve and the ideal calibration curve (the identity line from (0, 0) to (1, 1):

$$MA = \int_0^1 p_x - \hat{p}_x dx$$

Note that MA is equal to MACE as the number of bins (set by the `resolution` keyword argument) goes to infinity.

To choose which metric to compute, pass the name of the metric (`msce`, `rmsce`, `mace`, or `ma`) as the first argument to this function (or a list of them to compute multiple).

See Kuleshov et al., 2018, Chung et al., 2020 and the user guide page on *Evaluating Model Performance* for discussions of evaluating uncertainty estimates using calibration metrics, among other metrics. Note that calibration is generally less important than accuracy, but more important than other metrics like *`sharpness()`* and any *`dispersion_metric()`*.

**Parameters**

- **`metric`** (*`str {'msce', 'rmsce', 'mace', or 'ma'} or List[str]`*) – Which metric(s) to compute (see above for the definition of each metric). To compute multiple metrics, pass a list of the metric names you'd like to compute. Available metrics are:

  - `msce`: mean squared calibration error
  - `rmsce`: root mean squared calibration error
  - `mace`: mean absolute calibration error
  - `ma`: miscalibration area
- **x** (`ndarray` or `DataFrame` or `Series` or *`DataGenerator`*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *`DataGenerator`* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series`) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*`int`*) – Number of samples to draw from the model for computing the predictive percentile. Default = 1000
- **resolution** (*`int`*) – Number of confidence levels to evaluate at. This corresponds to the $m$ parameter in section 3.5 of (Kuleshov et al., 2018).
- **batch_size** (*`None or int`*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The requested calibration metric. If a list of metric names was passed, will return a dict whose keys are the metrics, and whose values are the corresponding metric values.

**Return type** float or Dict[str, float]

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute different calibration metrics using `expected_calibration_error()`. For example, to compute the mean squared calibration error (MSCE):

```
>>> model.calibration_metric("msce", x_val, y_val)
0.123
```

Or, to compute the mean absolute calibration error (MACE):

```
>>> model.calibration_metric("mace", x_val, y_val)
0.211
```

To compute multiple metrics at the same time, pass a list of metric names:

```
>>> model.calibration_metric(["msce", "mace"], x_val, y_val)
{"msce": 0.123, "mace": 0.211}
```

**See also:**

- *`calibration_curve()`*
- *`calibration_curve_plot()`*
- *`sharpness()`*

- *dispersion_metric()*

### References


- Volodymyr Kuleshov, Nathan Fenner, and Stefano Ermon. Accurate Uncertainties for Deep Learning Using Calibrated Regression, 2018.
- Youngseog Chung, Willie Neiswanger, Ian Char, Jeff Schneider. Beyond Pinball Loss: Quantile Methods for Calibrated Uncertainty Quantification, 2020.


**coverage_by**(*x_by*, *x*, *y=None*, *n: int = 1000*, *ci: float = 0.95*, *bins: int = 30*, *plot: bool = True*, *ideal_line_kwargs: dict = {}*, *batch_size=None*, ***kwargs*)

Compute and plot the coverage of a given confidence interval of the posterior predictive distribution as a function of specified independent variables.

TODO: Docs...

**Parameters**

- **x_by** (*int or str or list of int or list of str*) – Which independent variable(s) to plot the log probability as a function of. That is, which columns in `x` to plot by.
- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **ci** (*float between 0 and 1*) – Inner percentile to find the coverage of. For example, if `ci=0.95`, will compute the coverage of the inner 95% of the posterior predictive distribution.
- **bins** (*int*) – Number of bins to use for x_by
- **ideal_line_kwargs** (*dict*) – Dict of args to pass to matplotlib.pyplot.plot for ideal coverage line.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to plot_by

**Returns**

- **xo** (|*ndarray*|) – Values of x_by corresponding to bin centers.
- **co** (|*ndarray*|) – Coverage of the `ci` confidence interval of the predictive distribution in each bin.

**dispersion_metric**(*metric*, *x*, *n=1000*, *batch_size=None*)

Compute one or more of several calibration metrics

Dispersion metrics measure how much a model's uncertainty estimates vary. There are several different dispersion metrics:

The **coefficient of variation** ($C_v$) is the ratio of the standard deviation to the mean (of the model's uncertainty standard deviations):

$$C_v =$$

The **quartile coefficient of dispersion** ($QCD$) is less sensitive to outliers, as it simply measures the difference between the first and third quartile (of the model's uncertainty standard deviations) to their sum:

$$QCD = \frac{Q_3 - Q_1}{Q_3 + Q_1}$$

See Tran et al., 2020 and the user guide page on *Evaluating Model Performance* for discussions of evaluating uncertainty estimates using dispersion metrics, among other metrics. Note that dispersion metrics should generally be one of the last things you consider - accuracy, calibration, and sharpness usually being more important.

**Parameters**

- **metric** (*str {'cv' or 'qcd'} or List[str]*) – Dispersion metric to compute. Or,
- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **n** (*int*) – Number of samples to draw from the model. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The requested dispersion metric. If a list of metric names was passed, will return a dict whose keys are the metrics, and whose values are the corresponding metric values.

**Return type** float or Dict[str, float]

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute the coefficient of variation of our model's predictions with:

```
>>> model.dispersion_metric('cv', x_val)
0.732
```

Or the quartile coefficient of dispersion with:

```
>>> model.dispersion_metric('qcd', x_val)
0.625
```

**See also:**

- *calibration_metric()*
- *sharpness()*

### References


- Kevin Tran, Willie Neiswanger, Junwoong Yoon, Qingyang Zhang, Eric Xing, Zachary W. Ulissi. Methods for comparing uncertainty quantifications for material property predictions, 2020.


**dumps**()
Serialize module object to bytes

**elbo_loss**(*x_data*, *y_data*, *n: int*, *n_mc: int*)
Compute the negative ELBO, scaled to a single sample.

**Parameters**

- **x_data** – The independent variable values (or None if this is a generative model)
- **y_data** – The dependent variable values
- **n** (*int*) – Total number of datapoints in the dataset
- **n_mc** (*int*) – Number of MC samples we're taking from the posteriors

**epistemic_interval**(*x*, *ci=0.95*, *side='both'*, *n=1000*, *batch_size=None*)
Compute confidence intervals on the model's estimate of the target given `x`, including only epistemic uncertainty (uncertainty due to uncertainty as to the model's parameter values).

TODO: docs

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **ci** (*float between 0 and 1*) – Inner proportion of predictive distribution to use a the confidence interval. Default = 0.95
- **side** (*str {'lower', 'upper', 'both'}*) – Whether to get the one- or two-sided interval, and which side to get. If `'both'` (default), gets the upper and lower bounds of the central `ci` interval. If `'lower'`, gets the lower bound on the one-sided `ci` interval. If `'upper'`, gets the upper bound on the one-sided `ci` interval.
- **n** (*int*) – Number of samples from the epistemic predictive distribution to take to compute the confidence intervals. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **lb** (|*ndarray*|) – Lower bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='upper'`.
- **ub** (|*ndarray*|) – Upper bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='lower'`.

**epistemic_sample**(*x=None*, *n=1000*, *batch_size=None*)
Draw samples of the model's estimate given x, including only epistemic uncertainty (uncertainty due to uncertainty as to the model's parameter values)

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").

- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples, x.shape[0], ...)

**Return type** ndarray

**fit** (*x*, *y=None*, *batch_size: int = 128*, *epochs: int = 200*, *shuffle: bool = False*, *optimizer=None*, *optimizer_kwargs: dict = {}*, *lr: Optional[float] = None*, *flipout: bool = True*, *num_workers: Optional[int] = None*, *callbacks: List[*probflow.utils.base.BaseCallback*] = []*, *eager: bool = False*, *n_mc: int = 1*)
Fit the model to data

TODO

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values (or, if fitting a generative model, the dependent variable values). Should be of shape (Nsamples,...)
- **y** (None or ndarray or DataFrame or Series) – Dependent variable values (or, if fitting a generative model, None). Should be of shape (Nsamples,...). Default = None
- **batch_size** (*int*) – Number of samples to use per minibatch. Default = 128
- **epochs** (*int*) – Number of epochs to train the model. Default = 200
- **shuffle** (*bool*) – Whether to shuffle the data each epoch. Note that this is ignored if x is a *DataGenerator* Default = True
- **optimizer** (None or a backend-specific optimizer) – What optimizer to use for optimizing the variational posterior distributions' variables. When the backend is TensorFlow the default is to use adam (tf.keras.optimizers.Adam). When the backend is PyTorch the default is to use TODO
- **optimizer_kwargs** (*dict*) – Keyword arguments to pass to the optimizer. Default is an empty dict.
- **lr** (*float*) – Learning rate for the optimizer. Note that the learning rate can be updated during training using the set_learning_rate method. Default is $\exp(-\log_{10}(N_p N_b))$, where $N_p$ is the number of parameters in the model, and $N_b$ is the number of samples per batch (batch_size).
- **flipout** (*bool*) – Whether to use flipout during training where possible Default = True
- **num_workers** (*None or int > 0*) – Number of parallel processes to run for loading the data. If None, will not use parallel processes. If an integer, will use a process pool with that many processes. Note that this parameter is ignored if a *DataGenerator* is passed as x. Default = None
- **callbacks** (*List[*BaseCallback*]*) – List of callbacks to run while training the model. Default is [], i.e. no callbacks.
- **eager** (*bool*) – Whether to use eager execution. If False, will use tf.function (for TensorFlow) or tracing (for PyTorch) to optimize the model fitting. Note that even if eager=True, you can still use eager execution when using the model after it is fit. Default = False
- **n_mc** (*int*) – Number of monte carlo samples to take from the variational posteriors per minibatch. The default is to just take one per batch. Using a smaller number of MC

samples is faster, but using a greater number of MC samples will decrease the variance of the gradients, leading to more stable parameter optimization.

### Example

See the user guide section on *Fitting a Model*.

**get_elbo**()
Get the current ELBO on training data

**kl_loss**()
Compute the sum of the Kullback-Leibler divergences between priors and their variational posteriors for all *Parameters* in this *Module* and its sub-Modules.

**kl_loss_batch**()
Compute the sum of additional Kullback-Leibler divergences due to data in this batch

**log_likelihood**(*x_data*, *y_data*)
Compute the sum log likelihood of the model given a batch of data

**log_prob**(*x*, *y=None*, *individually=True*, *distribution=False*, *n=1000*, *batch_size=None*)
Compute the log probability of *y* given the model

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor) – Independent variable values of the dataset to evaluate (aka the "features").
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **individually** (*bool*) – If `individually` is True, returns log probability for each sample individually, so return shape is `(x.shape[0], ?)`. If `individually` is False, returns sum of all log probabilities, so return shape is `(1, ?)`.
- **distribution** (*bool*) – If `distribution` is True, returns log probability posterior distribution (`n` samples from the model), so return shape is `(?, n)`. If `distribution` is False, returns log posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is `(?, 1)`.
- **n** (*int*) – Number of samples to draw for each distribution if `distribution=True`.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns log_probs** – Log probabilities. Shape is determined by `individually`, `distribution`, and `n` kwargs.

**Return type** `ndarray`

**metric**(*metric*, *x*, *y=None*, *batch_size=None*)
Compute a metric of model performance

TODO: docs

TODO: note that this doesn't work w/ generative models

**Parameters**

- **metric** (*str or callable*) – Metric to evaluate. Available metrics:
  - 'lp': log likelihood sum

- 'log_prob': log likelihood sum
- 'accuracy': accuracy
- 'acc': accuracy
- 'mean_squared_error': mean squared error
- 'mse': mean squared error
- 'sum_squared_error': sum squared error
- 'sse': sum squared error
- 'mean_absolute_error': mean absolute error
- 'mae': mean absolute error
- 'r_squared': coefficient of determination
- 'r2': coefficient of determination
- 'recall': true positive rate
- 'sensitivity': true positive rate
- 'true_positive_rate': true positive rate
- 'tpr': true positive rate
- 'specificity': true negative rate
- 'selectivity': true negative rate
- 'true_negative_rate': true negative rate
- 'tnr': true negative rate
- 'precision': precision
- 'f1_score': F-measure
- 'f1': F-measure
- callable: a function which takes (y_true, y_pred)

• **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* to generate both x and y.

• **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").

• **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

**Return type** TODO

**property modules**
A list of sub-Modules in this *Module*, including itself.

**property n_parameters**
Get the number of independent parameters of this module

**property n_variables**
Get the number of underlying variables in this module

**property parameters**
A list of *Parameters* in this *Module* and its sub-Modules.

**posterior_ci** (*params=None*, *ci=0.95*, *n=10000*)
Posterior confidence intervals

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get the confidence intervals for all parameters in the model.
- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95
- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals Default = 10,000

**Returns** Confidence intervals of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain tuples. The first element of each tuple is the lower bound, and the second element is the upper bound. Or just a single tuple if params was a str

**Return type** dict

**posterior_mean** (*params=None*)
Get the mean of the posterior distribution(s)

TODO: Docs... params is a list of strings of params to plot

**Parameters** **params** (*str or List[str] or None*) – Parameter name(s) for which to compute the means. Default is to get the mean for all parameters in the model.

**Returns** Means of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the posterior means. The `ndarrays` are the same size as each parameter. Or just the `ndarray` if `params` was a str.

**Return type** dict

**posterior_plot** (*params=None*, *cols=1*, ***kwargs*)
Plot posterior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the posterior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.posterior_plot()*

**posterior_sample** (*params=None*, *n=10000*)
Draw samples from parameter posteriors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get a sample for all parameters in the model.

- **num_samples** (*int*) – Number of samples to take from each posterior distribution. Default = 1000

**Returns** Samples from the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the posterior samples. The `ndarrays` are of size (`num_samples`, param.shape). Or just the `ndarray` if `params` was a str.

**Return type** dict

**pred_dist_coverage**(*x*, *y=None*, *n=1000*, *ci=0.95*, *batch_size=None*)

Compute what percent of samples are covered by a given confidence interval.

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model given `x`. Default = 1000
- **ci** (*float between 0 and 1*) – Confidence interval to use.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** **prc_covered** – Proportion of the samples which were covered by the predictive distribution's confidence interval.

**Return type** float between 0 and 1

**pred_dist_covered**(*x*, *y=None*, *n: int = 1000*, *ci: float = 0.95*, *batch_size=None*)

Compute whether each observation was covered by a given confidence interval.

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model given `x`. Default = 1000
- **ci** (*float between 0 and 1*) – Confidence interval to use.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

**Return type** TODO

**pred_dist_plot**(*x*, *n=10000*, *cols=1*, *individually=False*, *batch_size=None*, ***kwargs*)

Plot posterior predictive distribution from the model given `x`.

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model given x. Default = 10000
- **cols** (*int*) – Divide the subplots into a grid with this many columns (if individually=True.
- **individually** (*bool*) – If True, plot one subplot per datapoint in x, otherwise plot all the predictive distributions on the same plot.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_dist()*

### Example

TODO

**predict**(*x=None*, *method='mean'*, *batch_size=None*)

Predict dependent variable using the model

TODO... using maximum a posteriori param estimates etc

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **method** (*str*) – Method to use for prediction. If 'mean', uses the mean of the predicted target distribution as the prediction. If 'mode', uses the mode of the distribution.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Predicted y-value for each sample in x. Of size (x.shape[0], y.shape[0], ..., y.shape[-1])

**Return type** ndarray

### Examples

TODO: Docs...

**predictive_interval**(*x*, *ci=0.95*, *side='both'*, *n=1000*, *batch_size=None*)

Compute confidence intervals on the model's estimate of the target given x, including all sources of uncertainty.

TODO: docs

TODO: using side= both, upper, vs lower

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **ci** (*float between 0 and 1*) – Inner proportion of predictive distribution to use a the confidence interval. Default = 0.95

- **side** (*str {'lower', 'upper', 'both'}*) – Whether to get the one- or two-sided interval, and which side to get. If `'both'` (default), gets the upper and lower bounds of the central `ci` interval. If `'lower'`, gets the lower bound on the one-sided `ci` interval. If `'upper'`, gets the upper bound on the one-sided `ci` interval.
- **n** (*int*) – Number of samples from the posterior predictive distribution to take to compute the confidence intervals. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **lb** (|*ndarray*|) – Lower bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='upper'`.
- **ub** (|*ndarray*|) – Upper bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='lower'`.

**predictive_prc**(*x*, *y=None*, *n=1000*, *batch_size=None*)

Compute the percentile of each observation along the posterior predictive distribution.

TODO: Docs... Returns a percentile between 0 and 1

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model given `x`. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** **prcs**

**Return type** `ndarray` of float between 0 and 1

**predictive_sample**(*x=None*, *n=1000*, *batch_size=None*)

Draw samples from the posterior predictive distribution given x

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predictive distribution. Size (num_samples, x.shape[0], ...)

**Return type** `ndarray`

**prior_plot**(*params=None*, *cols=1*, ***kwargs*)

Plot prior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the prior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.prior_plot()*

**prior_sample**(*params=None*, *n=10000*)

Draw samples from parameter priors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*list*) – List of parameter names to sample. Each element should be a str. Default is to sample priors of all parameters in the model.
- **n** (*int*) – Number of samples to take from each prior distribution. Default = 10000

**Returns** Samples from the parameter prior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the prior samples. The `ndarrays` are of size (`n`,param.shape).

**Return type** dict

**prob**(*x*, *y=None*, ***kwargs*)

Compute the probability of `y` given the model

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **individually** (*bool*) – If `individually` is True, returns probability for each sample individually, so return shape is `(x.shape[0], ?)`. If `individually` is False, returns product of all probabilities, so return shape is `(1, ?)`.
- **distribution** (*bool*) – If `distribution` is True, returns posterior probability distribution (`n` samples from the model), so return shape is `(?, n)`. If `distribution` is False, returns posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is `(?, 1)`.
- **n** (*int*) – Number of samples to draw for each distribution if `distribution=True`.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** **probs** – Probabilities. Shape is determined by `individually`, `distribution`, and `n` kwargs.

**Return type** `ndarray`

**r_squared**(*x*, *y=None*, *n=1000*, *batch_size=None*)

Compute the Bayesian R-squared distribution (Gelman et al., 2018).

TODO: more info

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of posterior draws to use for computing the r-squared distribution. Default = *1000*.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the r-squared distribution. Size: `(num_samples,)`.

**Return type** ndarray

### Examples

TODO: Docs...

### References


- Andrew Gelman, Ben Goodrich, Jonah Gabry, & Aki Vehtari. R-squared for Bayesian regression models. *The American Statistician*, 2018.


**r_squared_plot**(*x*, *y=None*, *n=1000*, *style='hist'*, *batch_size=None*, ***kwargs*)

Plot the Bayesian R-squared distribution.

See *r_squared()* for more info on the Bayesian R-squared metric.

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of posterior draws to use for computing the r-squared distribution. Default = *1000*.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_dist()*

### Example

TODO

**reset_kl_loss**()
Reset additional loss due to KL divergences

**residuals**(*x*, *y=None*, *batch_size=None*)
Compute the residuals of the model's predictions.

TODO: docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The residuals.

**Return type** ndarray

### Example

TODO

**residuals_plot**(*x*, *y=None*, *batch_size=None*, ***kwargs*)
Plot the distribution of residuals of the model's predictions.

TODO: docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_dist()*

### Example

TODO

**save** (*filename: str*)
Save module object to file

**Parameters** **filename** (*str*) – Filename for file to which to save this object

**set_kl_weight** (*w*)
Set the weight of the KL term's contribution to the ELBO loss

**set_learning_rate** (*lr*)
Set the learning rate used by this model's optimizer

**sharpness** (*x*, *n=1000*, *batch_size=None*)
Compute the sharpness of the model's uncertainty estimates

The "sharpness" of a model's uncertainty estimates is the root mean of the estimated variances:

$$SHA = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \text{Var}(\hat{Y}_i)}$$

See Tran et al., 2020 and the user guide page on *Evaluating Model Performance* for discussions of evaluating uncertainty estimates using sharpness, among other metrics. Note that the sharpness should generally be one of the later things you consider - accuracy and calibration usually being more important.

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **n** (*int*) – Number of samples to draw from the model. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The sharpness of the model's uncertainty estimates

**Return type** float

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute the sharpness of our model's predictions with:

```
>>> model.sharpness(x_val)
0.173
```

**See also:**

- *calibration_metric()*
- *dispersion_metric()*

### References


- Kevin Tran, Willie Neiswanger, Junwoong Yoon, Qingyang Zhang, Eric Xing, Zachary W. Ulissi. Methods for comparing uncertainty quantifications for material property predictions, 2020.


**stop_training**()
Stop the training of the model

**summary**()
Show a summary of the model and its parameters.

TODO

TODO: though maybe this should be a method of module... model would have to add to it the observation dist

**train_step**(*x_data*, *y_data*)
Perform one training step

**property trainable_variables**
A list of trainable backend variables within this *Module*

**class** probflow.applications.**LogisticRegression**(*d: int*, *k: int = 2*)
Bases: probflow.models.categorical_model.CategoricalModel

A logistic regression

TODO: explain, math, diagram, examples, etc

TODO: set k>2 for a Multinomial logistic regression

**Parameters**

- **d** (*int*) – Dimensionality of the independent variable (number of features)
- **k** (*int*) – Number of classes of the dependent variable

**weights**
Regression weights

**Type** *Parameter*

**bias**
Regression intercept

**Type** *Parameter*

**add_kl_loss**(*loss*, *d2=None*)
Add additional loss due to KL divergences.

**aleatoric_sample**(*x=None*, *n=1000*, *batch_size=None*)
Draw samples of the model's estimate given x, including only aleatoric uncertainty (uncertainty due to noise)

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples,x.shape[0],...)

**Return type** ndarray

**bayesian_update**()
Perform a Bayesian update of all *Parameters* in this module. Sets the prior to the current variational posterior for all parameters.

**calibration_curve**(*x*, *y=None*, *split_by=None*, *bins=10*, *plot=True*, *batch_size=None*)
Plot and return the categorical calibration curve.

Plots and returns the calibration curve (estimated probability of outcome vs the true probability of that outcome).

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **split_by** (*int*) – Draw the calibration curve independently for datapoints with each unique value in *x[:,split_by]* (a categorical column).
- **bins** (int, list of float, or ndarray) – Bins used to compute the curve. If an integer, will use *bins* evenly-spaced bins from 0 to 1. If a vector, *bins* is the vector of bin edges.
- **plot** (*bool*) – Whether to plot the curve
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- **#TODO** (*split by continuous cols as well? Then will need to define bins or edges too*) –
- **TODO** (*Docs...*) –

**dumps**()
Serialize module object to bytes

**elbo_loss**(*x_data*, *y_data*, *n: int*, *n_mc: int*)
Compute the negative ELBO, scaled to a single sample.

**Parameters**

- **x_data** – The independent variable values (or None if this is a generative model)
- **y_data** – The dependent variable values
- **n** (*int*) – Total number of datapoints in the dataset
- **n_mc** (*int*) – Number of MC samples we're taking from the posteriors

**epistemic_sample**(*x=None*, *n=1000*, *batch_size=None*)
Draw samples of the model's estimate given x, including only epistemic uncertainty (uncertainty due to uncertainty as to the model's parameter values)

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").

- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples, x.shape[0], ...)

**Return type** ndarray

**fit** (*x*, *y=None*, *batch_size: int = 128*, *epochs: int = 200*, *shuffle: bool = False*, *optimizer=None*, *optimizer_kwargs: dict = {}*, *lr: Optional[float] = None*, *flipout: bool = True*, *num_workers: Optional[int] = None*, *callbacks: List[*probflow.utils.base.BaseCallback*] = []*, *eager: bool = False*, *n_mc: int = 1*)

Fit the model to data

TODO

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values (or, if fitting a generative model, the dependent variable values). Should be of shape (Nsamples,...)
- **y** (None or ndarray or DataFrame or Series) – Dependent variable values (or, if fitting a generative model, None). Should be of shape (Nsamples,...). Default = None
- **batch_size** (*int*) – Number of samples to use per minibatch. Default = 128
- **epochs** (*int*) – Number of epochs to train the model. Default = 200
- **shuffle** (*bool*) – Whether to shuffle the data each epoch. Note that this is ignored if x is a *DataGenerator* Default = True
- **optimizer** (None or a backend-specific optimizer) – What optimizer to use for optimizing the variational posterior distributions' variables. When the backend is TensorFlow the default is to use adam (tf.keras.optimizers.Adam). When the backend is PyTorch the default is to use TODO
- **optimizer_kwargs** (*dict*) – Keyword arguments to pass to the optimizer. Default is an empty dict.
- **lr** (*float*) – Learning rate for the optimizer. Note that the learning rate can be updated during training using the set_learning_rate method. Default is $\exp(-\log_{10}(N_p N_b))$, where $N_p$ is the number of parameters in the model, and $N_b$ is the number of samples per batch (batch_size).
- **flipout** (*bool*) – Whether to use flipout during training where possible Default = True
- **num_workers** (*None or int > 0*) – Number of parallel processes to run for loading the data. If None, will not use parallel processes. If an integer, will use a process pool with that many processes. Note that this parameter is ignored if a *DataGenerator* is passed as x. Default = None
- **callbacks** (*List[*BaseCallback*]*) – List of callbacks to run while training the model. Default is [], i.e. no callbacks.
- **eager** (*bool*) – Whether to use eager execution. If False, will use tf.function (for TensorFlow) or tracing (for PyTorch) to optimize the model fitting. Note that even if eager=True, you can still use eager execution when using the model after it is fit. Default = False
- **n_mc** (*int*) – Number of monte carlo samples to take from the variational posteriors per minibatch. The default is to just take one per batch. Using a smaller number of MC

samples is faster, but using a greater number of MC samples will decrease the variance of the gradients, leading to more stable parameter optimization.

### Example

See the user guide section on *Fitting a Model*.

**get_elbo**()
Get the current ELBO on training data

**kl_loss**()
Compute the sum of the Kullback-Leibler divergences between priors and their variational posteriors for all *Parameters* in this *Module* and its sub-Modules.

**kl_loss_batch**()
Compute the sum of additional Kullback-Leibler divergences due to data in this batch

**log_likelihood**(*x_data*, *y_data*)
Compute the sum log likelihood of the model given a batch of data

**log_prob**(*x*, *y=None*, *individually=True*, *distribution=False*, *n=1000*, *batch_size=None*)
Compute the log probability of *y* given the model

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor) – Independent variable values of the dataset to evaluate (aka the "features").
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **individually** (*bool*) – If `individually` is True, returns log probability for each sample individually, so return shape is `(x.shape[0], ?)`. If `individually` is False, returns sum of all log probabilities, so return shape is `(1, ?)`.
- **distribution** (*bool*) – If `distribution` is True, returns log probability posterior distribution (`n` samples from the model), so return shape is `(?, n)`. If `distribution` is False, returns log posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is `(?, 1)`.
- **n** (*int*) – Number of samples to draw for each distribution if `distribution=True`.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns log_probs** – Log probabilities. Shape is determined by `individually`, `distribution`, and `n` kwargs.

**Return type** `ndarray`

**metric**(*metric*, *x*, *y=None*, *batch_size=None*)
Compute a metric of model performance

TODO: docs

TODO: note that this doesn't work w/ generative models

**Parameters**

- **metric** (*str or callable*) – Metric to evaluate. Available metrics:
  - 'lp': log likelihood sum

- 'log_prob': log likelihood sum
- 'accuracy': accuracy
- 'acc': accuracy
- 'mean_squared_error': mean squared error
- 'mse': mean squared error
- 'sum_squared_error': sum squared error
- 'sse': sum squared error
- 'mean_absolute_error': mean absolute error
- 'mae': mean absolute error
- 'r_squared': coefficient of determination
- 'r2': coefficient of determination
- 'recall': true positive rate
- 'sensitivity': true positive rate
- 'true_positive_rate': true positive rate
- 'tpr': true positive rate
- 'specificity': true negative rate
- 'selectivity': true negative rate
- 'true_negative_rate': true negative rate
- 'tnr': true negative rate
- 'precision': precision
- 'f1_score': F-measure
- 'f1': F-measure
- callable: a function which takes (y_true, y_pred)

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* to generate both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

**Return type** TODO

**property modules**
A list of sub-Modules in this *Module*, including itself.

**property n_parameters**
Get the number of independent parameters of this module

**property n_variables**
Get the number of underlying variables in this module

**property parameters**
A list of *Parameters* in this *Module* and its sub-Modules.

**posterior_ci**(*params=None*, *ci=0.95*, *n=10000*)
Posterior confidence intervals

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get the confidence intervals for all parameters in the model.
- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95
- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals Default = 10,000

**Returns** Confidence intervals of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain tuples. The first element of each tuple is the lower bound, and the second element is the upper bound. Or just a single tuple if params was a str

**Return type** dict

**posterior_mean**(*params=None*)
Get the mean of the posterior distribution(s)

TODO: Docs... params is a list of strings of params to plot

**Parameters** **params** (*str or List[str] or None*) – Parameter name(s) for which to compute the means. Default is to get the mean for all parameters in the model.

**Returns** Means of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the posterior means. The `ndarrays` are the same size as each parameter. Or just the `ndarray` if `params` was a str.

**Return type** dict

**posterior_plot**(*params=None*, *cols=1*, ***kwargs*)
Plot posterior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the posterior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.posterior_plot()*

**posterior_sample**(*params=None*, *n=10000*)
Draw samples from parameter posteriors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get a sample for all parameters in the model.

- **num_samples** (*int*) – Number of samples to take from each posterior distribution. Default = 1000

**Returns** Samples from the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain ndarrays with the posterior samples. The ndarrays are of size (num_samples, param.shape). Or just the ndarray if params was a str.

**Return type** dict

**pred_dist_plot**(*x*, *n=10000*, *cols=1*, *batch_size=None*, ***kwargs*)

Plot posterior predictive distribution from the model given x.

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model given x. Default = 10000
- **cols** (*int*) – Divide the subplots into a grid with this many columns (if individually=True.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_categorical_dist()*

**predict**(*x=None*, *method='mean'*, *batch_size=None*)

Predict dependent variable using the model

TODO... using maximum a posteriori param estimates etc

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **method** (*str*) – Method to use for prediction. If 'mean', uses the mean of the predicted target distribution as the prediction. If 'mode', uses the mode of the distribution.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Predicted y-value for each sample in x. Of size (x.shape[0], y.shape[0], ..., y.shape[-1])

**Return type** ndarray

### Examples

TODO: Docs...

**predictive_sample**(*x=None*, *n=1000*, *batch_size=None*)

Draw samples from the posterior predictive distribution given x

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predictive distribution. Size (num_samples, x.shape[0], ...)

**Return type** ndarray

**prior_plot** (*params=None*, *cols=1*, ***kwargs*)

Plot prior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the prior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.prior_plot()*

**prior_sample** (*params=None*, *n=10000*)

Draw samples from parameter priors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*list*) – List of parameter names to sample. Each element should be a str. Default is to sample priors of all parameters in the model.
- **n** (*int*) – Number of samples to take from each prior distribution. Default = 10000

**Returns** Samples from the parameter prior distributions. A dictionary where the keys contain the parameter names and the values contain ndarrays with the prior samples. The ndarrays are of size (n,param.shape).

**Return type** dict

**prob** (*x*, *y=None*, ***kwargs*)

Compute the probability of y given the model

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **individually** (*bool*) – If individually is True, returns probability for each sample individually, so return shape is (x.shape[0], ?). If individually is False, returns product of all probabilities, so return shape is (1, ?).
- **distribution** (*bool*) – If distribution is True, returns posterior probability distribution (n samples from the model), so return shape is (?, n). If distribution is False, returns posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is (?, 1).

- **n** (*int*) – Number of samples to draw for each distribution if `distribution=True`.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** **probs** – Probabilities. Shape is determined by `individually`, `distribution`, and `n` kwargs.

**Return type** `ndarray`

**reset_kl_loss**()
Reset additional loss due to KL divergences

**save**(*filename: str*)
Save module object to file

**Parameters** **filename** (*str*) – Filename for file to which to save this object

**set_kl_weight**(*w*)
Set the weight of the KL term's contribution to the ELBO loss

**set_learning_rate**(*lr*)
Set the learning rate used by this model's optimizer

**stop_training**()
Stop the training of the model

**summary**()
Show a summary of the model and its parameters.

TODO

TODO: though maybe this should be a method of module... model would have to add to it the observation dist

**train_step**(*x_data*, *y_data*)
Perform one training step

**property trainable_variables**
A list of trainable backend variables within this *Module*

**class** probflow.applications.**PoissonRegression**(*d: int*)
Bases: `probflow.models.discrete_model.DiscreteModel`

A Poisson regression (a type of generalized linear model)

TODO: explain, math, diagram, examples, etc

**Parameters** **d** (*int*) – Dimensionality of the independent variable (number of features)

**weights**
Regression weights

**Type** *Parameter*

**bias**
Regression intercept

**Type** *Parameter*

**add_kl_loss**(*loss*, *d2=None*)
Add additional loss due to KL divergences.

**aleatoric_interval**(*x*, *ci=0.95*, *side='both'*, *n=1000*, *batch_size=None*)
Compute confidence intervals on the model's estimate of the target given `x`, including only aleatoric uncertainty (uncertainty due to noise).

TODO: docs

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **ci** (*float between 0 and 1*) – Inner proportion of predictive distribution to use a the confidence interval. Default = 0.95
- **side** (*str {'lower', 'upper', 'both'}*) – Whether to get the one- or two-sided interval, and which side to get. If 'both' (default), gets the upper and lower bounds of the central ci interval. If 'lower', gets the lower bound on the one-sided ci interval. If 'upper', gets the upper bound on the one-sided ci interval.
- **n** (*int*) – Number of samples from the aleatoric predictive distribution to take to compute the confidence intervals. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **lb** (|*ndarray*|) – Lower bounds of the ci confidence intervals on the predictions for samples in x. Doesn't return this if side='upper'.
- **ub** (|*ndarray*|) – Upper bounds of the ci confidence intervals on the predictions for samples in x. Doesn't return this if side='lower'.

**aleatoric_sample**(*x=None*, *n=1000*, *batch_size=None*)

Draw samples of the model's estimate given x, including only aleatoric uncertainty (uncertainty due to noise)

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples,x.shape[0],...)

**Return type** ndarray

**bayesian_update**()

Perform a Bayesian update of all *Parameters* in this module. Sets the prior to the current variational posterior for all parameters.

**calibration_curve**(*x*, *y*, *n=1000*, *resolution=100*, *batch_size=None*)

Compute the regression calibration curve (Kuleshov et al., 2018).

The regression calibration curve compares the empirical cumulative probability to the cumulative probability predicted by a regression model (Kuleshov et al., 2018). First, a vector $p$ of $m$ confidence levels are chosen, which correspond to the predicted cumulative probabilities:

$$0 \leq p_1 \leq p_2 \leq \ldots \leq p_m \leq 1$$

Then, a vector of empirical frequencies $\hat{p}$ at each of the predicted frequencies is computed by using validation data:

$$\hat{p}_j = \frac{1}{N}\sum_{i=1}^{N}[P_M(x_i \leq y_i) \leq p_j]$$

where $N$ is the number of validation datapoints, $P_M(x_i \leq y_i)$ is the model's predicted cumulative probability of datapoint $i$ (i.e., the percentile along the model's predicted probability distribution at which the true value of $y_i$ falls), and $\sum_i[a_i \leq b_i]$ is just the count of elements of $a$ which are less than corresponding elements in $b$.

The calibration curve then plots $p$ against $\hat{p}$.

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model for computing the predictive percentile. Default = 1000
- **resolution** (*int*) – Number of confidence levels to evaluate at. This corresponds to the $m$ parameter in section 3.5 of (Kuleshov et al., 2018).
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **p** (*|ndarray|*) – The predicted cumulative frequencies, $p$.
- **p_hat** (*|ndarray|*) – The empirical cumulative frequencies, $\hat{p}$.

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute the calibration curve with *calibration_curve()*:

```
p_pred, p_empirical = model.calibration_curve(x_val, y_val)
```

The returned values can be used directly or plotted against one another to get the calibration curve (as in Figure 3 in Kuleshov et al., 2018)

```
import matplotlib.pyplot as plt
plt.plot(p_pred, p_empirical)
```

Or, even more simply, just use *calibration_curve_plot()*.

**See also:**

- *calibration_curve_plot()*

- `expected_calibration_error()`

### References


- Volodymyr Kuleshov, Nathan Fenner, and Stefano Ermon. Accurate Uncertainties for Deep Learning Using Calibrated Regression, 2018.


**`calibration_curve_plot`** (*x*, *y*, *n=1000*, *resolution=100*, *batch_size=None*, ***kwargs*)

Plot the regression calibration curve.

See *`calibration_curve()`* for more info about the regression calibration curve.

**Parameters**

- **`x`** (`ndarray` or `DataFrame` or `Series` or *`DataGenerator`*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *`DataGenerator`* for both x and y.
- **`y`** (`ndarray` or `DataFrame` or `Series`) – Dependent variable values of the dataset to evaluate (aka the "target").
- **`n`** (*`int`*) – Number of samples to draw from the model for computing the predictive percentile. Default = 1000
- **`resolution`** (*`int`*) – Number of confidence levels to evaluate at. This corresponds to the $m$ parameter in section 3.5 of (Kuleshov et al., 2018).
- **`batch_size`** (*`None or int`*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- **`**kwargs`** – Additional keyword arguments are passed to *`plot_dist()`*

**See also:**

- *`calibration_curve()`*
- `expected_calibration_error()`

**`calibration_metric`** (*metric*, *x*, *y=None*, *n=1000*, *resolution=100*, *batch_size=None*)

Compute one or more of several calibration metrics

Regression calibration metrics measure the error between a model's regression calibration curve and the ideal calibration curve - i.e., what the curve would be if the model were perfectly calibrated (see Kuleshov et al., 2018 and Chung et al., 2020). First, a vector $p$ of $m$ confidence levels are chosen, which correspond to the predicted cumulative probabilities:

$$0 \leq p_1 \leq p_2 \leq \ldots \leq p_m \leq 1$$

Then, a vector of empirical frequencies $\hat{p}$ at each of the predicted frequencies is computed by using validation data:

$$\hat{p}_j = \frac{1}{N} \sum_{i=1}^{N} [P_M(x_i \leq y_i) \leq p_j]$$

where $N$ is the number of validation datapoints, $P_M(x_i \leq y_i)$ is the model's predicted cumulative probability of datapoint $i$ (i.e., the percentile along the model's predicted probability distribution at which the true value of $y_i$ falls), and $\sum_i [a_i \leq b_i]$ is just the count of elements of $a$ which are less than corresponding elements in $b$.

Various metrics can be computed from these curves to measure how accurately the regression model captures uncertainty:

The **mean squared calibration error (MSCE)** is the mean squared error between the empirical and predicted frequencies,

$$MSCE = \frac{1}{m}\sum_{j=1}^{m}(p_j - \hat{p}_j)^2$$

The **root mean squared calibration error (RMSCE)** is just the square root of the MSCE:

$$RMSCE = \sqrt{\frac{1}{m}\sum_{j=1}^{m}(p_j - \hat{p}_j)^2}$$

The **mean absolute calibration error (MACE)** is the mean of the absolute differences between the empirical and predicted frequencies:

$$MACE = \frac{1}{m}\sum_{j=1}^{m}|p_j - \hat{p}_j|$$

And the **miscalibration area (MA)** is the area between the calibration curve and the ideal calibration curve (the identity line from (0, 0) to (1, 1):

$$MA = \int_0^1 p_x - \hat{p}_x dx$$

Note that MA is equal to MACE as the number of bins (set by the `resolution` keyword argument) goes to infinity.

To choose which metric to compute, pass the name of the metric (`msce`, `rmsce`, `mace`, or `ma`) as the first argument to this function (or a list of them to compute multiple).

See Kuleshov et al., 2018, Chung et al., 2020 and the user guide page on *Evaluating Model Performance* for discussions of evaluating uncertainty estimates using calibration metrics, among other metrics. Note that calibration is generally less important than accuracy, but more important than other metrics like *`sharpness()`* and any *`dispersion_metric()`*.

**Parameters**

- **metric** (*`str {'msce', 'rmsce', 'mace', or 'ma'} or List[str]`*) – Which metric(s) to compute (see above for the definition of each metric). To compute multiple metrics, pass a list of the metric names you'd like to compute. Available metrics are:
  - `msce`: mean squared calibration error
  - `rmsce`: root mean squared calibration error
  - `mace`: mean absolute calibration error
  - `ma`: miscalibration area
- **x** (`ndarray` or `DataFrame` or `Series` or *`DataGenerator`*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *`DataGenerator`* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series`) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*`int`*) – Number of samples to draw from the model for computing the predictive percentile. Default = 1000

- **resolution** (*int*) – Number of confidence levels to evaluate at. This corresponds to the $m$ parameter in section 3.5 of (Kuleshov et al., 2018).
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The requested calibration metric. If a list of metric names was passed, will return a dict whose keys are the metrics, and whose values are the corresponding metric values.

**Return type** float or Dict[str, float]

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute different calibration metrics using `expected_calibration_error()`. For example, to compute the mean squared calibration error (MSCE):

```
>>> model.calibration_metric("msce", x_val, y_val)
0.123
```

Or, to compute the mean absolute calibration error (MACE):

```
>>> model.calibration_metric("mace", x_val, y_val)
0.211
```

To compute multiple metrics at the same time, pass a list of metric names:

```
>>> model.calibration_metric(["msce", "mace"], x_val, y_val)
{"msce": 0.123, "mace": 0.211}
```

**See also:**

- *calibration_curve()*
- *calibration_curve_plot()*
- *sharpness()*
- *dispersion_metric()*

### References


- Volodymyr Kuleshov, Nathan Fenner, and Stefano Ermon. Accurate Uncertainties for Deep Learning Using Calibrated Regression, 2018.
- Youngseog Chung, Willie Neiswanger, Ian Char, Jeff Schneider. Beyond Pinball Loss: Quantile Methods for Calibrated Uncertainty Quantification, 2020.


**coverage_by**(*x_by*, *x*, *y=None*, *n: int = 1000*, *ci: float = 0.95*, *bins: int = 30*, *plot: bool = True*, *ideal_line_kwargs: dict = {}*, *batch_size=None*, ***kwargs*)

Compute and plot the coverage of a given confidence interval of the posterior predictive distribution as a function of specified independent variables.

TODO: Docs...

**Parameters**

- **x_by** (*int or str or list of int or list of str*) – Which independent variable(s) to plot the log probability as a function of. That is, which columns in `x` to plot by.
- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **ci** (*float between 0 and 1*) – Inner percentile to find the coverage of. For example, if `ci=0.95`, will compute the coverage of the inner 95% of the posterior predictive distribution.
- **bins** (*int*) – Number of bins to use for x_by
- **ideal_line_kwargs** (*dict*) – Dict of args to pass to matplotlib.pyplot.plot for ideal coverage line.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to plot_by

**Returns**

- **xo** (|*ndarray*|) – Values of x_by corresponding to bin centers.
- **co** (|*ndarray*|) – Coverage of the `ci` confidence interval of the predictive distribution in each bin.

**dispersion_metric**(*metric*, *x*, *n=1000*, *batch_size=None*)

Compute one or more of several calibration metrics

Dispersion metrics measure how much a model's uncertainty estimates vary. There are several different dispersion metrics:

The **coefficient of variation** ($C_v$) is the ratio of the standard deviation to the mean (of the model's uncertainty standard deviations):

$$C_v =$$

The **quartile coefficient of dispersion** ($QCD$) is less sensitive to outliers, as it simply measures the difference between the first and third quartile (of the model's uncertainty standard deviations) to their sum:

$$QCD = \frac{Q_3 - Q_1}{Q_3 + Q_1}$$

See Tran et al., 2020 and the user guide page on *Evaluating Model Performance* for discussions of evaluating uncertainty estimates using dispersion metrics, among other metrics. Note that dispersion metrics should generally be one of the last things you consider - accuracy, calibration, and sharpness usually being more important.

**Parameters**

- **metric** (*str {'cv' or 'qcd'} or List[str]*) – Dispersion metric to compute. Or,

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **n** (*int*) – Number of samples to draw from the model. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The requested dispersion metric. If a list of metric names was passed, will return a dict whose keys are the metrics, and whose values are the corresponding metric values.

**Return type** float or Dict[str, float]

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute the coefficient of variation of our model's predictions with:

```
>>> model.dispersion_metric('cv', x_val)
0.732
```

Or the quartile coefficient of dispersion with:

```
>>> model.dispersion_metric('qcd', x_val)
0.625
```

**See also:**

- *calibration_metric()*
- *sharpness()*

### References


- Kevin Tran, Willie Neiswanger, Junwoong Yoon, Qingyang Zhang, Eric Xing, Zachary W. Ulissi. Methods for comparing uncertainty quantifications for material property predictions, 2020.


**dumps**()
Serialize module object to bytes

**elbo_loss**(*x_data*, *y_data*, *n: int*, *n_mc: int*)
Compute the negative ELBO, scaled to a single sample.

**Parameters**

- **x_data** – The independent variable values (or None if this is a generative model)
- **y_data** – The dependent variable values
- **n** (*int*) – Total number of datapoints in the dataset
- **n_mc** (*int*) – Number of MC samples we're taking from the posteriors

**epistemic_interval**(*x*, *ci=0.95*, *side='both'*, *n=1000*, *batch_size=None*)

Compute confidence intervals on the model's estimate of the target given `x`, including only epistemic uncertainty (uncertainty due to uncertainty as to the model's parameter values).

TODO: docs

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *`DataGenerator`*) – Independent variable values of the dataset to evaluate (aka the "features").
- **ci** (*`float between 0 and 1`*) – Inner proportion of predictive distribution to use a the confidence interval. Default = 0.95
- **side** (*`str {'lower', 'upper', 'both'}`*) – Whether to get the one- or two-sided interval, and which side to get. If `'both'` (default), gets the upper and lower bounds of the central `ci` interval. If `'lower'`, gets the lower bound on the one-sided `ci` interval. If `'upper'`, gets the upper bound on the one-sided `ci` interval.
- **n** (*`int`*) – Number of samples from the epistemic predictive distribution to take to compute the confidence intervals. Default = 1000
- **batch_size** (*`None or int`*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **lb** (|*ndarray*|) – Lower bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='upper'`.
- **ub** (|*ndarray*|) – Upper bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='lower'`.

**epistemic_sample**(*x=None*, *n=1000*, *batch_size=None*)

Draw samples of the model's estimate given x, including only epistemic uncertainty (uncertainty due to uncertainty as to the model's parameter values)

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *`DataGenerator`*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*`int`*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*`None or int`*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples, x.shape[0], ...)

**Return type** `ndarray`

**fit**(*x*, *y=None*, *batch_size: int = 128*, *epochs: int = 200*, *shuffle: bool = False*, *optimizer=None*, *optimizer_kwargs: dict = {}*, *lr: Optional[float] = None*, *flipout: bool = True*, *num_workers: Optional[int] = None*, *callbacks: List[*probflow.utils.base.BaseCallback*] = []*, *eager: bool = False*, *n_mc: int = 1*)

Fit the model to data

TODO

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values (or, if fitting a generative model, the dependent variable values). Should be of shape (Nsamples,...)
- **y** (None or ndarray or DataFrame or Series) – Dependent variable values (or, if fitting a generative model, None). Should be of shape (Nsamples,...). Default = None
- **batch_size** (*int*) – Number of samples to use per minibatch. Default = 128
- **epochs** (*int*) – Number of epochs to train the model. Default = 200
- **shuffle** (*bool*) – Whether to shuffle the data each epoch. Note that this is ignored if x is a *DataGenerator* Default = True
- **optimizer** (None or a backend-specific optimizer) – What optimizer to use for optimizing the variational posterior distributions' variables. When the backend is TensorFlow the default is to use adam (tf.keras.optimizers.Adam). When the backend is PyTorch the default is to use TODO
- **optimizer_kwargs** (*dict*) – Keyword arguments to pass to the optimizer. Default is an empty dict.
- **lr** (*float*) – Learning rate for the optimizer. Note that the learning rate can be updated during training using the set_learning_rate method. Default is $\exp(-\log_{10}(N_p N_b))$, where $N_p$ is the number of parameters in the model, and $N_b$ is the number of samples per batch (batch_size).
- **flipout** (*bool*) – Whether to use flipout during training where possible Default = True
- **num_workers** (*None or int > 0*) – Number of parallel processes to run for loading the data. If None, will not use parallel processes. If an integer, will use a process pool with that many processes. Note that this parameter is ignored if a *DataGenerator* is passed as x. Default = None
- **callbacks** (*List[BaseCallback]*) – List of callbacks to run while training the model. Default is [], i.e. no callbacks.
- **eager** (*bool*) – Whether to use eager execution. If False, will use tf.function (for TensorFlow) or tracing (for PyTorch) to optimize the model fitting. Note that even if eager=True, you can still use eager execution when using the model after it is fit. Default = False
- **n_mc** (*int*) – Number of monte carlo samples to take from the variational posteriors per minibatch. The default is to just take one per batch. Using a smaller number of MC samples is faster, but using a greater number of MC samples will decrease the variance of the gradients, leading to more stable parameter optimization.

#### Example

See the user guide section on *Fitting a Model*.

**get_elbo**()
Get the current ELBO on training data

**kl_loss**()
Compute the sum of the Kullback-Leibler divergences between priors and their variational posteriors for all *Parameters* in this *Module* and its sub-Modules.

**kl_loss_batch**()
Compute the sum of additional Kullback-Leibler divergences due to data in this batch

**log_likelihood**(*x_data*, *y_data*)
Compute the sum log likelihood of the model given a batch of data

**log_prob**(*x*, *y=None*, *individually=True*, *distribution=False*, *n=1000*, *batch_size=None*)
Compute the log probability of *y* given the model

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor) – Independent variable values of the dataset to evaluate (aka the "features").
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **individually** (*bool*) – If `individually` is True, returns log probability for each sample individually, so return shape is `(x.shape[0], ?)`. If `individually` is False, returns sum of all log probabilities, so return shape is `(1, ?)`.
- **distribution** (*bool*) – If `distribution` is True, returns log probability posterior distribution (`n` samples from the model), so return shape is `(?, n)`. If `distribution` is False, returns log posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is `(?, 1)`.
- **n** (*int*) – Number of samples to draw for each distribution if `distribution=True`.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns log_probs** – Log probabilities. Shape is determined by `individually`, `distribution`, and `n` kwargs.

**Return type** ndarray

**metric**(*metric*, *x*, *y=None*, *batch_size=None*)
Compute a metric of model performance

TODO: docs

TODO: note that this doesn't work w/ generative models

**Parameters**

- **metric** (*str or callable*) – Metric to evaluate. Available metrics:
  - 'lp': log likelihood sum
  - 'log_prob': log likelihood sum
  - 'accuracy': accuracy
  - 'acc': accuracy
  - 'mean_squared_error': mean squared error
  - 'mse': mean squared error
  - 'sum_squared_error': sum squared error
  - 'sse': sum squared error
  - 'mean_absolute_error': mean absolute error
  - 'mae': mean absolute error
  - 'r_squared': coefficient of determination

- 'r2': coefficient of determination
- 'recall': true positive rate
- 'sensitivity': true positive rate
- 'true_positive_rate': true positive rate
- 'tpr': true positive rate
- 'specificity': true negative rate
- 'selectivity': true negative rate
- 'true_negative_rate': true negative rate
- 'tnr': true negative rate
- 'precision': precision
- 'f1_score': F-measure
- 'f1': F-measure
- callable: a function which takes (y_true, y_pred)

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* to generate both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

**Return type** TODO

**property modules**
A list of sub-Modules in this *Module*, including itself.

**property n_parameters**
Get the number of independent parameters of this module

**property n_variables**
Get the number of underlying variables in this module

**property parameters**
A list of *Parameters* in this *Module* and its sub-Modules.

**posterior_ci**(*params=None*, *ci=0.95*, *n=10000*)
Posterior confidence intervals

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get the confidence intervals for all parameters in the model.
- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95
- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals Default = 10,000

**Returns** Confidence intervals of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain tuples. The first element of each tuple is the lower bound, and the second element is the upper bound. Or just a single tuple if params was a str

**Return type** dict

**posterior_mean**(*params=None*)
Get the mean of the posterior distribution(s)

TODO: Docs... params is a list of strings of params to plot

**Parameters** **params** (*str or List[str] or None*) – Parameter name(s) for which to compute the means. Default is to get the mean for all parameters in the model.

**Returns** Means of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the posterior means. The `ndarrays` are the same size as each parameter. Or just the `ndarray` if `params` was a str.

**Return type** dict

**posterior_plot**(*params=None*, *cols=1*, ***kwargs*)
Plot posterior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the posterior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.posterior_plot()*

**posterior_sample**(*params=None*, *n=10000*)
Draw samples from parameter posteriors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get a sample for all parameters in the model.
- **num_samples** (*int*) – Number of samples to take from each posterior distribution. Default = 1000

**Returns** Samples from the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the posterior samples. The `ndarrays` are of size (`num_samples`, param.shape). Or just the `ndarray` if `params` was a str.

**Return type** dict

**pred_dist_coverage**(*x*, *y=None*, *n=1000*, *ci=0.95*, *batch_size=None*)
Compute what percent of samples are covered by a given confidence interval.

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model given x. Default = 1000
- **ci** (*float between 0 and 1*) – Confidence interval to use.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** **prc_covered** – Proportion of the samples which were covered by the predictive distribution's confidence interval.

**Return type** float between 0 and 1

**pred_dist_covered**(*x*, *y=None*, *n: int = 1000*, *ci: float = 0.95*, *batch_size=None*)

Compute whether each observation was covered by a given confidence interval.

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model given x. Default = 1000
- **ci** (*float between 0 and 1*) – Confidence interval to use.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

**Return type** TODO

**pred_dist_plot**(*x*, *n=10000*, *cols=1*, ***kwargs*)

Plot posterior predictive distribution from the model given x.

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model given x. Default = 10000
- **cols** (*int*) – Divide the subplots into a grid with this many columns (if individually=True.
- ****kwargs** – Additional keyword arguments are passed to *plot_discrete_dist()*

**predict**(*x=None*, *method='mean'*, *batch_size=None*)

Predict dependent variable using the model

TODO... using maximum a posteriori param estimates etc

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **method** (*str*) – Method to use for prediction. If `'mean'`, uses the mean of the predicted target distribution as the prediction. If `'mode'`, uses the mode of the distribution.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Predicted y-value for each sample in `x`. Of size (x.shape[0], y.shape[0], ..., y.shape[-1])

**Return type** ndarray

### Examples

TODO: Docs...

**predictive_interval**(*x*, *ci=0.95*, *side='both'*, *n=1000*, *batch_size=None*)

Compute confidence intervals on the model's estimate of the target given `x`, including all sources of uncertainty.

TODO: docs

TODO: using side= both, upper, vs lower

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **ci** (*float between 0 and 1*) – Inner proportion of predictive distribution to use a the confidence interval. Default = 0.95
- **side** (*str {'lower', 'upper', 'both'}*) – Whether to get the one- or two-sided interval, and which side to get. If `'both'` (default), gets the upper and lower bounds of the central `ci` interval. If `'lower'`, gets the lower bound on the one-sided `ci` interval. If `'upper'`, gets the upper bound on the one-sided `ci` interval.
- **n** (*int*) – Number of samples from the posterior predictive distribution to take to compute the confidence intervals. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **lb** (|*ndarray*|) – Lower bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='upper'`.
- **ub** (|*ndarray*|) – Upper bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='lower'`.

**predictive_prc**(*x*, *y=None*, *n=1000*, *batch_size=None*)

Compute the percentile of each observation along the posterior predictive distribution.

TODO: Docs... Returns a percentile between 0 and 1

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.

- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model given x. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** prcs

**Return type** ndarray of float between 0 and 1

**predictive_sample**(*x=None*, *n=1000*, *batch_size=None*)

Draw samples from the posterior predictive distribution given x

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predictive distribution. Size (num_samples, x.shape[0], ...)

**Return type** ndarray

**prior_plot**(*params=None*, *cols=1*, ***kwargs*)

Plot prior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the prior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.prior_plot()*

**prior_sample**(*params=None*, *n=10000*)

Draw samples from parameter priors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*list*) – List of parameter names to sample. Each element should be a str. Default is to sample priors of all parameters in the model.
- **n** (*int*) – Number of samples to take from each prior distribution. Default = 10000

**Returns** Samples from the parameter prior distributions. A dictionary where the keys contain the parameter names and the values contain ndarrays with the prior samples. The ndarrays are of size (n,param.shape).

**Return type** dict

**prob**(*x*, *y=None*, ***kwargs*)

Compute the probability of y given the model

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **individually** (*bool*) – If `individually` is True, returns probability for each sample individually, so return shape is `(x.shape[0], ?)`. If `individually` is False, returns product of all probabilities, so return shape is `(1, ?)`.
- **distribution** (*bool*) – If `distribution` is True, returns posterior probability distribution (`n` samples from the model), so return shape is `(?, n)`. If `distribution` is False, returns posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is `(?, 1)`.
- **n** (*int*) – Number of samples to draw for each distribution if `distribution=True`.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** **probs** – Probabilities. Shape is determined by `individually`, `distribution`, and `n` kwargs.

**Return type** ndarray

**r_squared**(**args*, ***kwargs*)
Cannot compute R squared for a discrete model

**r_squared_plot**(**args*, ***kwargs*)
Cannot compute R squared for a discrete model

**reset_kl_loss**()
Reset additional loss due to KL divergences

**residuals**(*x*, *y=None*, *batch_size=None*)
Compute the residuals of the model's predictions.

TODO: docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The residuals.

**Return type** ndarray

### Example

TODO

**residuals_plot** (*x*, *y=None*, *batch_size=None*, ***kwargs*)
Plot the distribution of residuals of the model's predictions.

TODO: docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_dist()*

### Example

TODO

**save** (*filename: str*)
Save module object to file

**Parameters** **filename** (*str*) – Filename for file to which to save this object

**set_kl_weight** (*w*)
Set the weight of the KL term's contribution to the ELBO loss

**set_learning_rate** (*lr*)
Set the learning rate used by this model's optimizer

**sharpness** (*x*, *n=1000*, *batch_size=None*)
Compute the sharpness of the model's uncertainty estimates

The "sharpness" of a model's uncertainty estimates is the root mean of the estimated variances:

$$SHA = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \mathrm{Var}(\hat{Y}_i)}$$

See Tran et al., 2020 and the user guide page on *Evaluating Model Performance* for discussions of evaluating uncertainty estimates using sharpness, among other metrics. Note that the sharpness should generally be one of the later things you consider - accuracy and calibration usually being more important.

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **n** (*int*) – Number of samples to draw from the model. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The sharpness of the model's uncertainty estimates

**Return type** float

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute the sharpness of our model's predictions with:

```
>>> model.sharpness(x_val)
0.173
```

**See also:**

- *calibration_metric()*
- *dispersion_metric()*

### References


- Kevin Tran, Willie Neiswanger, Junwoong Yoon, Qingyang Zhang, Eric Xing, Zachary W. Ulissi. Methods for comparing uncertainty quantifications for material property predictions, 2020.


**stop_training**()
Stop the training of the model

**summary**()
Show a summary of the model and its parameters.

TODO

TODO: though maybe this should be a method of module... model would have to add to it the observation dist

**train_step**(*x_data*, *y_data*)
Perform one training step

**property trainable_variables**
A list of trainable backend variables within this *Module*

**class** probflow.applications.**DenseRegression**(*d: List[int]*, *heteroscedastic: bool = False*, ***kwargs*)

Bases: probflow.models.continuous_model.ContinuousModel

A regression using a multilayer dense neural network

TODO: explain, math, diagram, examples, etc

**Parameters**

- **d** (*List[int]*) – Dimensionality (number of units) for each layer. The first element should be the dimensionality of the independent variable (number of features), and the last element should be the dimensionality of the dependent variable (number of dimensions of the target).
- **heteroscedastic** (*bool*) – Whether to model a change in noise as a function of x (if `heteroscedastic=True`), or not (if `heteroscedastic=False`, the default).

- **activation** (*callable*) – Activation function to apply to the outputs of each layer. Note that the activation function will not be applied to the outputs of the final layer. Default = $\max(0, x)$
- **kwargs** – Additional keyword arguments are passed to *DenseNetwork*

**network**
The multilayer dense neural network which generates predictions of the mean

**Type** *DenseNetwork*

**std**
Standard deviation of the Normal observation distribution

**Type** *ScaleParameter*

**add_kl_loss**(*loss*, *d2=None*)
Add additional loss due to KL divergences.

**aleatoric_interval**(*x*, *ci=0.95*, *side='both'*, *n=1000*, *batch_size=None*)
Compute confidence intervals on the model's estimate of the target given `x`, including only aleatoric uncertainty (uncertainty due to noise).

TODO: docs

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **ci** (*float between 0 and 1*) – Inner proportion of predictive distribution to use a the confidence interval. Default = 0.95
- **side** (*str {'lower', 'upper', 'both'}*) – Whether to get the one- or two-sided interval, and which side to get. If `'both'` (default), gets the upper and lower bounds of the central `ci` interval. If `'lower'`, gets the lower bound on the one-sided `ci` interval. If `'upper'`, gets the upper bound on the one-sided `ci` interval.
- **n** (*int*) – Number of samples from the aleatoric predictive distribution to take to compute the confidence intervals. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **lb** (|*ndarray*|) – Lower bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='upper'`.
- **ub** (|*ndarray*|) – Upper bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='lower'`.

**aleatoric_sample**(*x=None*, *n=1000*, *batch_size=None*)
Draw samples of the model's estimate given x, including only aleatoric uncertainty (uncertainty due to noise)

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.

- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples,x.shape[0],...)

**Return type** ndarray

**bayesian_update**()
Perform a Bayesian update of all *Parameters* in this module. Sets the prior to the current variational posterior for all parameters.

**calibration_curve**(*x*, *y*, *n=1000*, *resolution=100*, *batch_size=None*)
Compute the regression calibration curve (Kuleshov et al., 2018).

The regression calibration curve compares the empirical cumulative probability to the cumulative probability predicted by a regression model (Kuleshov et al., 2018). First, a vector $p$ of $m$ confidence levels are chosen, which correspond to the predicted cumulative probabilities:

$$0 \leq p_1 \leq p_2 \leq \ldots \leq p_m \leq 1$$

Then, a vector of empirical frequencies $\hat{p}$ at each of the predicted frequencies is computed by using validation data:

$$\hat{p}_j = \frac{1}{N} \sum_{i=1}^{N} [P_M(x_i \leq y_i) \leq p_j]$$

where $N$ is the number of validation datapoints, $P_M(x_i \leq y_i)$ is the model's predicted cumulative probability of datapoint $i$ (i.e., the percentile along the model's predicted probability distribution at which the true value of $y_i$ falls), and $\sum_i [a_i \leq b_i]$ is just the count of elements of $a$ which are less than corresponding elements in $b$.

The calibration curve then plots $p$ against $\hat{p}$.

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model for computing the predictive percentile. Default = 1000
- **resolution** (*int*) – Number of confidence levels to evaluate at. This corresponds to the $m$ parameter in section 3.5 of (Kuleshov et al., 2018).
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **p** (*|ndarray|*) – The predicted cumulative frequencies, $p$.
- **p_hat** (*|ndarray|*) – The empirical cumulative frequencies, $\hat{p}$.

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute the calibration curve with *calibration_curve()*:

```
p_pred, p_empirical = model.calibration_curve(x_val, y_val)
```

The returned values can be used directly or plotted against one another to get the calibration curve (as in Figure 3 in Kuleshov et al., 2018)

```
import matplotlib.pyplot as plt
plt.plot(p_pred, p_empirical)
```

Or, even more simply, just use *calibration_curve_plot()*.

**See also:**

- *calibration_curve_plot()*
- `expected_calibration_error()`

### References


- Volodymyr Kuleshov, Nathan Fenner, and Stefano Ermon. Accurate Uncertainties for Deep Learning Using Calibrated Regression, 2018.


**calibration_curve_plot** (*x*, *y*, *n=1000*, *resolution=100*, *batch_size=None*, ***kwargs*)

Plot the regression calibration curve.

See *calibration_curve()* for more info about the regression calibration curve.

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series`) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model for computing the predictive percentile. Default = 1000
- **resolution** (*int*) – Number of confidence levels to evaluate at. This corresponds to the $m$ parameter in section 3.5 of (Kuleshov et al., 2018).
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_dist()*

**See also:**

- *calibration_curve()*

- `expected_calibration_error()`

**`calibration_metric`**(*metric*, *x*, *y=None*, *n=1000*, *resolution=100*, *batch_size=None*)

Compute one or more of several calibration metrics

Regression calibration metrics measure the error between a model's regression calibration curve and the ideal calibration curve - i.e., what the curve would be if the model were perfectly calibrated (see Kuleshov et al., 2018 and Chung et al., 2020). First, a vector $p$ of $m$ confidence levels are chosen, which correspond to the predicted cumulative probabilities:

$$0 \leq p_1 \leq p_2 \leq \ldots \leq p_m \leq 1$$

Then, a vector of empirical frequencies $\hat{p}$ at each of the predicted frequencies is computed by using validation data:

$$\hat{p}_j = \frac{1}{N} \sum_{i=1}^{N} [P_M(x_i \leq y_i) \leq p_j]$$

where $N$ is the number of validation datapoints, $P_M(x_i \leq y_i)$ is the model's predicted cumulative probability of datapoint $i$ (i.e., the percentile along the model's predicted probability distribution at which the true value of $y_i$ falls), and $\sum_i [a_i \leq b_i]$ is just the count of elements of $a$ which are less than corresponding elements in $b$.

Various metrics can be computed from these curves to measure how accurately the regression model captures uncertainty:

The **mean squared calibration error (MSCE)** is the mean squared error between the empirical and predicted frequencies,

$$MSCE = \frac{1}{m} \sum_{j=1}^{m} (p_j - \hat{p}_j)^2$$

The **root mean squared calibration error (RMSCE)** is just the square root of the MSCE:

$$RMSCE = \sqrt{\frac{1}{m} \sum_{j=1}^{m} (p_j - \hat{p}_j)^2}$$

The **mean absolute calibration error (MACE)** is the mean of the absolute differences between the empirical and predicted frequencies:

$$MACE = \frac{1}{m} \sum_{j=1}^{m} |p_j - \hat{p}_j|$$

And the **miscalibration area (MA)** is the area between the calibration curve and the ideal calibration curve (the identity line from (0, 0) to (1, 1):

$$MA = \int_0^1 p_x - \hat{p}_x dx$$

Note that MA is equal to MACE as the number of bins (set by the `resolution` keyword argument) goes to infinity.

To choose which metric to compute, pass the name of the metric (`msce`, `rmsce`, `mace`, or `ma`) as the first argument to this function (or a list of them to compute multiple).

See Kuleshov et al., 2018, Chung et al., 2020 and the user guide page on *Evaluating Model Performance* for discussions of evaluating uncertainty estimates using calibration metrics, among other metrics. Note that calibration is generally less important than accuracy, but more important than other metrics like *`sharpness()`* and any *`dispersion_metric()`*.

**Parameters**

- **metric** (*str {'msce', 'rmsce', 'mace', or 'ma'} or List[str]*) – Which metric(s) to compute (see above for the definition of each metric). To compute multiple metrics, pass a list of the metric names you'd like to compute. Available metrics are:
  - `msce`: mean squared calibration error
  - `rmsce`: root mean squared calibration error
  - `mace`: mean absolute calibration error
  - `ma`: miscalibration area
- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series`) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model for computing the predictive percentile. Default = 1000
- **resolution** (*int*) – Number of confidence levels to evaluate at. This corresponds to the $m$ parameter in section 3.5 of (Kuleshov et al., 2018).
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The requested calibration metric. If a list of metric names was passed, will return a dict whose keys are the metrics, and whose values are the corresponding metric values.

**Return type** float or Dict[str, float]

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute different calibration metrics using `expected_calibration_error()`. For example, to compute the mean squared calibration error (MSCE):

```
>>> model.calibration_metric("msce", x_val, y_val)
0.123
```

Or, to compute the mean absolute calibration error (MACE):

```
>>> model.calibration_metric("mace", x_val, y_val)
0.211
```

To compute multiple metrics at the same time, pass a list of metric names:

```
>>> model.calibration_metric(["msce", "mace"], x_val, y_val)
{"msce": 0.123, "mace": 0.211}
```

**See also:**

- *calibration_curve()*
- *calibration_curve_plot()*
- *sharpness()*
- *dispersion_metric()*

### References


- Volodymyr Kuleshov, Nathan Fenner, and Stefano Ermon. Accurate Uncertainties for Deep Learning Using Calibrated Regression, 2018.
- Youngseog Chung, Willie Neiswanger, Ian Char, Jeff Schneider. Beyond Pinball Loss: Quantile Methods for Calibrated Uncertainty Quantification, 2020.


**coverage_by**(*x_by*, *x*, *y=None*, *n: int = 1000*, *ci: float = 0.95*, *bins: int = 30*, *plot: bool = True*, *ideal_line_kwargs: dict = {}*, *batch_size=None*, ***kwargs*)

Compute and plot the coverage of a given confidence interval of the posterior predictive distribution as a function of specified independent variables.

TODO: Docs...

**Parameters**

- **x_by** (*int or str or list of int or list of str*) – Which independent variable(s) to plot the log probability as a function of. That is, which columns in `x` to plot by.
- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **ci** (*float between 0 and 1*) – Inner percentile to find the coverage of. For example, if `ci=0.95`, will compute the coverage of the inner 95% of the posterior predictive distribution.
- **bins** (*int*) – Number of bins to use for x_by
- **ideal_line_kwargs** (*dict*) – Dict of args to pass to matplotlib.pyplot.plot for ideal coverage line.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to plot_by

**Returns**

- **xo** (|*ndarray*|) – Values of x_by corresponding to bin centers.
- **co** (|*ndarray*|) – Coverage of the `ci` confidence interval of the predictive distribution in each bin.

**dispersion_metric**(*metric*, *x*, *n=1000*, *batch_size=None*)

Compute one or more of several calibration metrics

Dispersion metrics measure how much a model's uncertainty estimates vary. There are several different dispersion metrics:

The **coefficient of variation** ($C_v$) is the ratio of the standard deviation to the mean (of the model's uncertainty standard deviations):

$$C_v =$$

The **quartile coefficient of dispersion** ($QCD$) is less sensitive to outliers, as it simply measures the difference between the first and third quartile (of the model's uncertainty standard deviations) to their sum:

$$QCD = \frac{Q_3 - Q_1}{Q_3 + Q_1}$$

See Tran et al., 2020 and the user guide page on *Evaluating Model Performance* for discussions of evaluating uncertainty estimates using dispersion metrics, among other metrics. Note that dispersion metrics should generally be one of the last things you consider - accuracy, calibration, and sharpness usually being more important.

**Parameters**

- **metric** (*str {'cv' or 'qcd'} or List[str]*) – Dispersion metric to compute. Or,
- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **n** (*int*) – Number of samples to draw from the model. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The requested dispersion metric. If a list of metric names was passed, will return a dict whose keys are the metrics, and whose values are the corresponding metric values.

**Return type** float or Dict[str, float]

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute the coefficient of variation of our model's predictions with:

```
>>> model.dispersion_metric('cv', x_val)
0.732
```

Or the quartile coefficient of dispersion with:

```
>>> model.dispersion_metric('qcd', x_val)
0.625
```

**See also:**

- *calibration_metric()*
- *sharpness()*

### References


- Kevin Tran, Willie Neiswanger, Junwoong Yoon, Qingyang Zhang, Eric Xing, Zachary W. Ulissi. Methods for comparing uncertainty quantifications for material property predictions, 2020.


**dumps**()
Serialize module object to bytes

**elbo_loss**(*x_data*, *y_data*, *n: int*, *n_mc: int*)
Compute the negative ELBO, scaled to a single sample.

**Parameters**

- **x_data** – The independent variable values (or None if this is a generative model)
- **y_data** – The dependent variable values
- **n** (*int*) – Total number of datapoints in the dataset
- **n_mc** (*int*) – Number of MC samples we're taking from the posteriors

**epistemic_interval**(*x*, *ci=0.95*, *side='both'*, *n=1000*, *batch_size=None*)
Compute confidence intervals on the model's estimate of the target given `x`, including only epistemic uncertainty (uncertainty due to uncertainty as to the model's parameter values).

TODO: docs

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **ci** (*float between 0 and 1*) – Inner proportion of predictive distribution to use a the confidence interval. Default = 0.95
- **side** (*str {'lower', 'upper', 'both'}*) – Whether to get the one- or two-sided interval, and which side to get. If `'both'` (default), gets the upper and lower bounds of the central `ci` interval. If `'lower'`, gets the lower bound on the one-sided `ci` interval. If `'upper'`, gets the upper bound on the one-sided `ci` interval.
- **n** (*int*) – Number of samples from the epistemic predictive distribution to take to compute the confidence intervals. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **lb** (|*ndarray*|) – Lower bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='upper'`.
- **ub** (|*ndarray*|) – Upper bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='lower'`.

**epistemic_sample**(*x=None*, *n=1000*, *batch_size=None*)
Draw samples of the model's estimate given x, including only epistemic uncertainty (uncertainty due to uncertainty as to the model's parameter values)

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").

- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples, x.shape[0], ...)

**Return type** ndarray

**fit** (*x*, *y=None*, *batch_size: int = 128*, *epochs: int = 200*, *shuffle: bool = False*, *optimizer=None*, *optimizer_kwargs: dict = {}*, *lr: Optional[float] = None*, *flipout: bool = True*, *num_workers: Optional[int] = None*, *callbacks: List[*probflow.utils.base.BaseCallback*] = []*, *eager: bool = False*, *n_mc: int = 1*)
Fit the model to data

TODO

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values (or, if fitting a generative model, the dependent variable values). Should be of shape (Nsamples,...)
- **y** (None or ndarray or DataFrame or Series) – Dependent variable values (or, if fitting a generative model, None). Should be of shape (Nsamples,...). Default = None
- **batch_size** (*int*) – Number of samples to use per minibatch. Default = 128
- **epochs** (*int*) – Number of epochs to train the model. Default = 200
- **shuffle** (*bool*) – Whether to shuffle the data each epoch. Note that this is ignored if x is a *DataGenerator* Default = True
- **optimizer** (None or a backend-specific optimizer) – What optimizer to use for optimizing the variational posterior distributions' variables. When the backend is TensorFlow the default is to use adam (tf.keras.optimizers.Adam). When the backend is PyTorch the default is to use TODO
- **optimizer_kwargs** (*dict*) – Keyword arguments to pass to the optimizer. Default is an empty dict.
- **lr** (*float*) – Learning rate for the optimizer. Note that the learning rate can be updated during training using the set_learning_rate method. Default is $\exp(-\log_{10}(N_p N_b))$, where $N_p$ is the number of parameters in the model, and $N_b$ is the number of samples per batch (batch_size).
- **flipout** (*bool*) – Whether to use flipout during training where possible Default = True
- **num_workers** (*None or int > 0*) – Number of parallel processes to run for loading the data. If None, will not use parallel processes. If an integer, will use a process pool with that many processes. Note that this parameter is ignored if a *DataGenerator* is passed as x. Default = None
- **callbacks** (*List[*BaseCallback*]*) – List of callbacks to run while training the model. Default is [], i.e. no callbacks.
- **eager** (*bool*) – Whether to use eager execution. If False, will use tf.function (for TensorFlow) or tracing (for PyTorch) to optimize the model fitting. Note that even if eager=True, you can still use eager execution when using the model after it is fit. Default = False
- **n_mc** (*int*) – Number of monte carlo samples to take from the variational posteriors per minibatch. The default is to just take one per batch. Using a smaller number of MC

samples is faster, but using a greater number of MC samples will decrease the variance of the gradients, leading to more stable parameter optimization.

### Example

See the user guide section on *Fitting a Model*.

**get_elbo**()
Get the current ELBO on training data

**kl_loss**()
Compute the sum of the Kullback-Leibler divergences between priors and their variational posteriors for all *Parameters* in this *Module* and its sub-Modules.

**kl_loss_batch**()
Compute the sum of additional Kullback-Leibler divergences due to data in this batch

**log_likelihood**(*x_data*, *y_data*)
Compute the sum log likelihood of the model given a batch of data

**log_prob**(*x*, *y=None*, *individually=True*, *distribution=False*, *n=1000*, *batch_size=None*)
Compute the log probability of *y* given the model

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor) – Independent variable values of the dataset to evaluate (aka the "features").
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **individually** (*bool*) – If `individually` is True, returns log probability for each sample individually, so return shape is `(x.shape[0], ?)`. If `individually` is False, returns sum of all log probabilities, so return shape is `(1, ?)`.
- **distribution** (*bool*) – If `distribution` is True, returns log probability posterior distribution (`n` samples from the model), so return shape is `(?, n)`. If `distribution` is False, returns log posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is `(?, 1)`.
- **n** (*int*) – Number of samples to draw for each distribution if `distribution=True`.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns log_probs** – Log probabilities. Shape is determined by `individually`, `distribution`, and `n` kwargs.

**Return type** `ndarray`

**metric**(*metric*, *x*, *y=None*, *batch_size=None*)
Compute a metric of model performance

TODO: docs

TODO: note that this doesn't work w/ generative models

**Parameters**

- **metric** (*str or callable*) – Metric to evaluate. Available metrics:
  - 'lp': log likelihood sum

- 'log_prob': log likelihood sum
- 'accuracy': accuracy
- 'acc': accuracy
- 'mean_squared_error': mean squared error
- 'mse': mean squared error
- 'sum_squared_error': sum squared error
- 'sse': sum squared error
- 'mean_absolute_error': mean absolute error
- 'mae': mean absolute error
- 'r_squared': coefficient of determination
- 'r2': coefficient of determination
- 'recall': true positive rate
- 'sensitivity': true positive rate
- 'true_positive_rate': true positive rate
- 'tpr': true positive rate
- 'specificity': true negative rate
- 'selectivity': true negative rate
- 'true_negative_rate': true negative rate
- 'tnr': true negative rate
- 'precision': precision
- 'f1_score': F-measure
- 'f1': F-measure
- callable: a function which takes (y_true, y_pred)

• **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* to generate both x and y.

• **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").

• **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

**Return type** TODO

**property modules**
A list of sub-Modules in this *Module*, including itself.

**property n_parameters**
Get the number of independent parameters of this module

**property n_variables**
Get the number of underlying variables in this module

**property parameters**
A list of *Parameters* in this *Module* and its sub-Modules.

**posterior_ci** (*params=None*, *ci=0.95*, *n=10000*)
Posterior confidence intervals

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get the confidence intervals for all parameters in the model.
- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95
- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals Default = 10,000

**Returns** Confidence intervals of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain tuples. The first element of each tuple is the lower bound, and the second element is the upper bound. Or just a single tuple if params was a str

**Return type** dict

**posterior_mean** (*params=None*)
Get the mean of the posterior distribution(s)

TODO: Docs... params is a list of strings of params to plot

**Parameters** **params** (*str or List[str] or None*) – Parameter name(s) for which to compute the means. Default is to get the mean for all parameters in the model.

**Returns** Means of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the posterior means. The `ndarrays` are the same size as each parameter. Or just the `ndarray` if `params` was a str.

**Return type** dict

**posterior_plot** (*params=None*, *cols=1*, ***kwargs*)
Plot posterior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the posterior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.posterior_plot()*

**posterior_sample** (*params=None*, *n=10000*)
Draw samples from parameter posteriors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get a sample for all parameters in the model.

- **num_samples** (*int*) – Number of samples to take from each posterior distribution. Default = 1000

**Returns** Samples from the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the posterior samples. The `ndarrays` are of size (`num_samples`, param.shape). Or just the `ndarray` if `params` was a str.

**Return type** dict

**pred_dist_coverage**(*x*, *y=None*, *n=1000*, *ci=0.95*, *batch_size=None*)

Compute what percent of samples are covered by a given confidence interval.

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model given `x`. Default = 1000
- **ci** (*float between 0 and 1*) – Confidence interval to use.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** **prc_covered** – Proportion of the samples which were covered by the predictive distribution's confidence interval.

**Return type** float between 0 and 1

**pred_dist_covered**(*x*, *y=None*, *n: int = 1000*, *ci: float = 0.95*, *batch_size=None*)

Compute whether each observation was covered by a given confidence interval.

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model given `x`. Default = 1000
- **ci** (*float between 0 and 1*) – Confidence interval to use.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

**Return type** TODO

**pred_dist_plot**(*x*, *n=10000*, *cols=1*, *individually=False*, *batch_size=None*, ***kwargs*)

Plot posterior predictive distribution from the model given `x`.

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model given x. Default = 10000
- **cols** (*int*) – Divide the subplots into a grid with this many columns (if individually=True.
- **individually** (*bool*) – If True, plot one subplot per datapoint in x, otherwise plot all the predictive distributions on the same plot.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_dist()*

### Example

TODO

**predict**(*x=None*, *method='mean'*, *batch_size=None*)

Predict dependent variable using the model

TODO... using maximum a posteriori param estimates etc

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **method** (*str*) – Method to use for prediction. If 'mean', uses the mean of the predicted target distribution as the prediction. If 'mode', uses the mode of the distribution.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Predicted y-value for each sample in x. Of size (x.shape[0], y.shape[0], ..., y.shape[-1])

**Return type** ndarray

### Examples

TODO: Docs...

**predictive_interval**(*x*, *ci=0.95*, *side='both'*, *n=1000*, *batch_size=None*)

Compute confidence intervals on the model's estimate of the target given x, including all sources of uncertainty.

TODO: docs

TODO: using side= both, upper, vs lower

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **ci** (*float between 0 and 1*) – Inner proportion of predictive distribution to use a the confidence interval. Default = 0.95

- **side** (*str {'lower', 'upper', 'both'}*) – Whether to get the one- or two-sided interval, and which side to get. If `'both'` (default), gets the upper and lower bounds of the central `ci` interval. If `'lower'`, gets the lower bound on the one-sided `ci` interval. If `'upper'`, gets the upper bound on the one-sided `ci` interval.
- **n** (*int*) – Number of samples from the posterior predictive distribution to take to compute the confidence intervals. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

- **lb** (|*ndarray*|) – Lower bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='upper'`.
- **ub** (|*ndarray*|) – Upper bounds of the `ci` confidence intervals on the predictions for samples in `x`. Doesn't return this if `side='lower'`.

**predictive_prc** (*x*, *y=None*, *n=1000*, *batch_size=None*)

Compute the percentile of each observation along the posterior predictive distribution.

TODO: Docs... Returns a percentile between 0 and 1

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of samples to draw from the model given `x`. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** **prcs**

**Return type** `ndarray` of float between 0 and 1

**predictive_sample** (*x=None*, *n=1000*, *batch_size=None*)

Draw samples from the posterior predictive distribution given x

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predictive distribution. Size (num_samples, x.shape[0], ...)

**Return type** `ndarray`

**prior_plot** (*params=None*, *cols=1*, ***kwargs*)

Plot prior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the prior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.prior_plot()*

**prior_sample**(*params=None*, *n=10000*)

Draw samples from parameter priors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*list*) – List of parameter names to sample. Each element should be a str. Default is to sample priors of all parameters in the model.
- **n** (*int*) – Number of samples to take from each prior distribution. Default = 10000

**Returns** Samples from the parameter prior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the prior samples. The `ndarrays` are of size (`n`,param.shape).

**Return type** dict

**prob**(*x*, *y=None*, ***kwargs*)

Compute the probability of `y` given the model

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **individually** (*bool*) – If `individually` is True, returns probability for each sample individually, so return shape is `(x.shape[0], ?)`. If `individually` is False, returns product of all probabilities, so return shape is `(1, ?)`.
- **distribution** (*bool*) – If `distribution` is True, returns posterior probability distribution (`n` samples from the model), so return shape is `(?, n)`. If `distribution` is False, returns posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is `(?, 1)`.
- **n** (*int*) – Number of samples to draw for each distribution if `distribution=True`.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** **probs** – Probabilities. Shape is determined by `individually`, `distribution`, and `n` kwargs.

**Return type** `ndarray`

**r_squared**(*x*, *y=None*, *n=1000*, *batch_size=None*)

Compute the Bayesian R-squared distribution (Gelman et al., 2018).

TODO: more info

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of posterior draws to use for computing the r-squared distribution. Default = *1000*.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the r-squared distribution. Size: `(num_samples,)`.

**Return type** ndarray

### Examples

TODO: Docs...

### References


- Andrew Gelman, Ben Goodrich, Jonah Gabry, & Aki Vehtari. R-squared for Bayesian regression models. *The American Statistician*, 2018.


**r_squared_plot**(*x*, *y=None*, *n=1000*, *style='hist'*, *batch_size=None*, ***kwargs*)

Plot the Bayesian R-squared distribution.

See *r_squared()* for more info on the Bayesian R-squared metric.

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **n** (*int*) – Number of posterior draws to use for computing the r-squared distribution. Default = *1000*.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_dist()*

### Example

TODO

**reset_kl_loss**()
Reset additional loss due to KL divergences

**residuals**(*x*, *y=None*, *batch_size=None*)
Compute the residuals of the model's predictions.

TODO: docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The residuals.

**Return type** ndarray

### Example

TODO

**residuals_plot**(*x*, *y=None*, *batch_size=None*, ***kwargs*)
Plot the distribution of residuals of the model's predictions.

TODO: docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series) – Dependent variable values of the dataset to evaluate (aka the "target").
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_dist()*

### Example

TODO

**save** (*filename: str*)
Save module object to file

**Parameters** **filename** (*str*) – Filename for file to which to save this object

**set_kl_weight** (*w*)
Set the weight of the KL term's contribution to the ELBO loss

**set_learning_rate** (*lr*)
Set the learning rate used by this model's optimizer

**sharpness** (*x*, *n=1000*, *batch_size=None*)
Compute the sharpness of the model's uncertainty estimates

The "sharpness" of a model's uncertainty estimates is the root mean of the estimated variances:

$$SHA = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \text{Var}(\hat{Y}_i)}$$

See Tran et al., 2020 and the user guide page on *Evaluating Model Performance* for discussions of evaluating uncertainty estimates using sharpness, among other metrics. Note that the sharpness should generally be one of the later things you consider - accuracy and calibration usually being more important.

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **n** (*int*) – Number of samples to draw from the model. Default = 1000
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** The sharpness of the model's uncertainty estimates

**Return type** float

### Example

Supposing we have some training data (`x_train` and `y_train`) and validation data (`x_val` and `y_val`), and have already fit a model to the training data,

```
model = # some ProbFlow model...
model.fit(x_train, y_train)
```

Then we can compute the sharpness of our model's predictions with:

```
>>> model.sharpness(x_val)
0.173
```

**See also:**

- *calibration_metric()*
- *dispersion_metric()*

### References


- Kevin Tran, Willie Neiswanger, Junwoong Yoon, Qingyang Zhang, Eric Xing, Zachary W. Ulissi. Methods for comparing uncertainty quantifications for material property predictions, 2020.


**stop_training**()
Stop the training of the model

**summary**()
Show a summary of the model and its parameters.

TODO

TODO: though maybe this should be a method of module... model would have to add to it the observation dist

**train_step**(*x_data*, *y_data*)
Perform one training step

**property trainable_variables**
A list of trainable backend variables within this *Module*

**class** probflow.applications.**DenseClassifier**(*d: List[int]*, ***kwargs*)
Bases: probflow.models.categorical_model.CategoricalModel

A classifier which uses a multilayer dense neural network

TODO: explain, math, diagram, examples, etc

**Parameters**

- **d** (*List[int]*) – Dimensionality (number of units) for each layer. The first element should be the dimensionality of the independent variable (number of features), and the last element should be the number of classes of the target.
- **activation** (*callable*) – Activation function to apply to the outputs of each layer. Note that the activation function will not be applied to the outputs of the final layer. Default = $\max(0, x)$
- **kwargs** – Additional keyword arguments are passed to *DenseNetwork*

**network**
The multilayer dense neural network which generates predictions of the class probabilities

**Type** *DenseNetwork*

**add_kl_loss**(*loss*, *d2=None*)
Add additional loss due to KL divergences.

**aleatoric_sample**(*x=None*, *n=1000*, *batch_size=None*)
Draw samples of the model's estimate given x, including only aleatoric uncertainty (uncertainty due to noise)

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples,x.shape[0],...)

**Return type** ndarray

**bayesian_update**()
Perform a Bayesian update of all *Parameters* in this module. Sets the prior to the current variational posterior for all parameters.

**calibration_curve**(*x*, *y=None*, *split_by=None*, *bins=10*, *plot=True*, *batch_size=None*)
Plot and return the categorical calibration curve.

Plots and returns the calibration curve (estimated probability of outcome vs the true probability of that outcome).

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **split_by** (*int*) – Draw the calibration curve independently for datapoints with each unique value in *x[:,split_by]* (a categorical column).
- **bins** (int, list of float, or ndarray) – Bins used to compute the curve. If an integer, will use *bins* evenly-spaced bins from 0 to 1. If a vector, *bins* is the vector of bin edges.
- **plot** (*bool*) – Whether to plot the curve
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- **#TODO** (*split by continuous cols as well? Then will need to define bins or edges too*) –
- **TODO** (*Docs...*) –

**dumps**()
Serialize module object to bytes

**elbo_loss**(*x_data*, *y_data*, *n: int*, *n_mc: int*)
Compute the negative ELBO, scaled to a single sample.

**Parameters**

- **x_data** – The independent variable values (or None if this is a generative model)
- **y_data** – The dependent variable values
- **n** (*int*) – Total number of datapoints in the dataset
- **n_mc** (*int*) – Number of MC samples we're taking from the posteriors

**epistemic_sample**(*x=None*, *n=1000*, *batch_size=None*)
Draw samples of the model's estimate given x, including only epistemic uncertainty (uncertainty due to uncertainty as to the model's parameter values)

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").

- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predicted mean distribution. Size (num_samples, x.shape[0], ...)

**Return type** ndarray

**fit** (*x*, *y=None*, *batch_size: int = 128*, *epochs: int = 200*, *shuffle: bool = False*, *optimizer=None*, *optimizer_kwargs: dict = {}*, *lr: Optional[float] = None*, *flipout: bool = True*, *num_workers: Optional[int] = None*, *callbacks: List[probflow.utils.base.BaseCallback] = []*, *eager: bool = False*, *n_mc: int = 1*)
Fit the model to data

TODO

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values (or, if fitting a generative model, the dependent variable values). Should be of shape (Nsamples,...)
- **y** (None or ndarray or DataFrame or Series) – Dependent variable values (or, if fitting a generative model, None). Should be of shape (Nsamples,...). Default = None
- **batch_size** (*int*) – Number of samples to use per minibatch. Default = 128
- **epochs** (*int*) – Number of epochs to train the model. Default = 200
- **shuffle** (*bool*) – Whether to shuffle the data each epoch. Note that this is ignored if x is a *DataGenerator* Default = True
- **optimizer** (None or a backend-specific optimizer) – What optimizer to use for optimizing the variational posterior distributions' variables. When the backend is TensorFlow the default is to use adam (tf.keras.optimizers.Adam). When the backend is PyTorch the default is to use TODO
- **optimizer_kwargs** (*dict*) – Keyword arguments to pass to the optimizer. Default is an empty dict.
- **lr** (*float*) – Learning rate for the optimizer. Note that the learning rate can be updated during training using the set_learning_rate method. Default is $\exp(-\log_{10}(N_p N_b))$, where $N_p$ is the number of parameters in the model, and $N_b$ is the number of samples per batch (batch_size).
- **flipout** (*bool*) – Whether to use flipout during training where possible Default = True
- **num_workers** (*None or int > 0*) – Number of parallel processes to run for loading the data. If None, will not use parallel processes. If an integer, will use a process pool with that many processes. Note that this parameter is ignored if a *DataGenerator* is passed as x. Default = None
- **callbacks** (*List[BaseCallback]*) – List of callbacks to run while training the model. Default is [], i.e. no callbacks.
- **eager** (*bool*) – Whether to use eager execution. If False, will use tf.function (for TensorFlow) or tracing (for PyTorch) to optimize the model fitting. Note that even if eager=True, you can still use eager execution when using the model after it is fit. Default = False
- **n_mc** (*int*) – Number of monte carlo samples to take from the variational posteriors per minibatch. The default is to just take one per batch. Using a smaller number of MC

samples is faster, but using a greater number of MC samples will decrease the variance of the gradients, leading to more stable parameter optimization.

### Example

See the user guide section on *Fitting a Model*.

**get_elbo**()
Get the current ELBO on training data

**kl_loss**()
Compute the sum of the Kullback-Leibler divergences between priors and their variational posteriors for all *Parameters* in this *Module* and its sub-Modules.

**kl_loss_batch**()
Compute the sum of additional Kullback-Leibler divergences due to data in this batch

**log_likelihood**(*x_data*, *y_data*)
Compute the sum log likelihood of the model given a batch of data

**log_prob**(*x*, *y=None*, *individually=True*, *distribution=False*, *n=1000*, *batch_size=None*)
Compute the log probability of *y* given the model

TODO: Docs...

**Parameters**

- **x** (`ndarray` or `DataFrame` or `Series` or Tensor) – Independent variable values of the dataset to evaluate (aka the "features").
- **y** (`ndarray` or `DataFrame` or `Series` or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **individually** (*bool*) – If `individually` is True, returns log probability for each sample individually, so return shape is `(x.shape[0], ?)`. If `individually` is False, returns sum of all log probabilities, so return shape is `(1, ?)`.
- **distribution** (*bool*) – If `distribution` is True, returns log probability posterior distribution (`n` samples from the model), so return shape is `(?, n)`. If `distribution` is False, returns log posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is `(?, 1)`.
- **n** (*int*) – Number of samples to draw for each distribution if `distribution=True`.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns log_probs** – Log probabilities. Shape is determined by `individually`, `distribution`, and `n` kwargs.

**Return type** `ndarray`

**metric**(*metric*, *x*, *y=None*, *batch_size=None*)
Compute a metric of model performance

TODO: docs

TODO: note that this doesn't work w/ generative models

**Parameters**

- **metric** (*str or callable*) – Metric to evaluate. Available metrics:
  - 'lp': log likelihood sum

- 'log_prob': log likelihood sum
- 'accuracy': accuracy
- 'acc': accuracy
- 'mean_squared_error': mean squared error
- 'mse': mean squared error
- 'sum_squared_error': sum squared error
- 'sse': sum squared error
- 'mean_absolute_error': mean absolute error
- 'mae': mean absolute error
- 'r_squared': coefficient of determination
- 'r2': coefficient of determination
- 'recall': true positive rate
- 'sensitivity': true positive rate
- 'true_positive_rate': true positive rate
- 'tpr': true positive rate
- 'specificity': true negative rate
- 'selectivity': true negative rate
- 'true_negative_rate': true negative rate
- 'tnr': true negative rate
- 'precision': precision
- 'f1_score': F-measure
- 'f1': F-measure
- callable: a function which takes (y_true, y_pred)

• **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* to generate both x and y.

• **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").

• **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns**

**Return type** TODO

**property modules**
A list of sub-Modules in this *Module*, including itself.

**property n_parameters**
Get the number of independent parameters of this module

**property n_variables**
Get the number of underlying variables in this module

**property parameters**
A list of *Parameters* in this *Module* and its sub-Modules.

**posterior_ci** (*params=None*, *ci=0.95*, *n=10000*)
Posterior confidence intervals

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get the confidence intervals for all parameters in the model.
- **ci** (*float*) – Confidence interval for which to compute the upper and lower bounds. Must be between 0 and 1. Default = 0.95
- **n** (*int*) – Number of samples to draw from the posterior distributions for computing the confidence intervals Default = 10,000

**Returns** Confidence intervals of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain tuples. The first element of each tuple is the lower bound, and the second element is the upper bound. Or just a single tuple if params was a str

**Return type** dict

**posterior_mean** (*params=None*)
Get the mean of the posterior distribution(s)

TODO: Docs... params is a list of strings of params to plot

**Parameters** **params** (*str or List[str] or None*) – Parameter name(s) for which to compute the means. Default is to get the mean for all parameters in the model.

**Returns** Means of the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain `ndarrays` with the posterior means. The `ndarrays` are the same size as each parameter. Or just the `ndarray` if `params` was a str.

**Return type** dict

**posterior_plot** (*params=None*, *cols=1*, ***kwargs*)
Plot posterior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the posterior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.posterior_plot()*

**posterior_sample** (*params=None*, *n=10000*)
Draw samples from parameter posteriors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or List[str] or None*) – Parameter name(s) to sample. Default is to get a sample for all parameters in the model.

- **num_samples** (*int*) – Number of samples to take from each posterior distribution. Default = 1000

**Returns** Samples from the parameter posterior distributions. A dictionary where the keys contain the parameter names and the values contain ndarrays with the posterior samples. The ndarrays are of size (num_samples, param.shape). Or just the ndarray if params was a str.

**Return type** dict

**pred_dist_plot**(*x*, *n=10000*, *cols=1*, *batch_size=None*, ***kwargs*)

Plot posterior predictive distribution from the model given x.

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model given x. Default = 10000
- **cols** (*int*) – Divide the subplots into a grid with this many columns (if individually=True.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).
- ****kwargs** – Additional keyword arguments are passed to *plot_categorical_dist()*

**predict**(*x=None*, *method='mean'*, *batch_size=None*)

Predict dependent variable using the model

TODO... using maximum a posteriori param estimates etc

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **method** (*str*) – Method to use for prediction. If 'mean', uses the mean of the predicted target distribution as the prediction. If 'mode', uses the mode of the distribution.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Predicted y-value for each sample in x. Of size (x.shape[0], y.shape[0], ..., y.shape[-1])

**Return type** ndarray

### Examples

TODO: Docs...

**predictive_sample**(*x=None*, *n=1000*, *batch_size=None*)

Draw samples from the posterior predictive distribution given x

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features").
- **n** (*int*) – Number of samples to draw from the model per datapoint.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** Samples from the predictive distribution. Size (num_samples, x.shape[0], ...)

**Return type** ndarray

**prior_plot** (*params=None*, *cols=1*, ***kwargs*)

Plot prior distributions of the model's parameters

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*str or list or None*) – List of names of parameters to plot. Default is to plot the prior of all parameters in the model.
- **cols** (*int*) – Divide the subplots into a grid with this many columns.
- **kwargs** – Additional keyword arguments are passed to *Parameter.prior_plot()*

**prior_sample** (*params=None*, *n=10000*)

Draw samples from parameter priors

TODO: Docs... params is a list of strings of params to plot

**Parameters**

- **params** (*list*) – List of parameter names to sample. Each element should be a str. Default is to sample priors of all parameters in the model.
- **n** (*int*) – Number of samples to take from each prior distribution. Default = 10000

**Returns** Samples from the parameter prior distributions. A dictionary where the keys contain the parameter names and the values contain ndarrays with the prior samples. The ndarrays are of size (n,param.shape).

**Return type** dict

**prob** (*x*, *y=None*, ***kwargs*)

Compute the probability of y given the model

TODO: Docs...

**Parameters**

- **x** (ndarray or DataFrame or Series or Tensor or *DataGenerator*) – Independent variable values of the dataset to evaluate (aka the "features"). Or a *DataGenerator* for both x and y.
- **y** (ndarray or DataFrame or Series or Tensor) – Dependent variable values of the dataset to evaluate (aka the "target").
- **individually** (*bool*) – If individually is True, returns probability for each sample individually, so return shape is (x.shape[0], ?). If individually is False, returns product of all probabilities, so return shape is (1, ?).
- **distribution** (*bool*) – If distribution is True, returns posterior probability distribution (n samples from the model), so return shape is (?, n). If distribution is False, returns posterior probabilities using the maximum a posteriori estimate for each parameter, so the return shape is (?, 1).

- **n** (*int*) – Number of samples to draw for each distribution if `distribution=True`.
- **batch_size** (*None or int*) – Compute using batches of this many datapoints. Default is *None* (i.e., do not use batching).

**Returns** **probs** – Probabilities. Shape is determined by `individually`, `distribution`, and `n` kwargs.

**Return type** `ndarray`

**reset_kl_loss**()
Reset additional loss due to KL divergences

**save**(*filename: str*)
Save module object to file

**Parameters** **filename** (*str*) – Filename for file to which to save this object

**set_kl_weight**(*w*)
Set the weight of the KL term's contribution to the ELBO loss

**set_learning_rate**(*lr*)
Set the learning rate used by this model's optimizer

**stop_training**()
Stop the training of the model

**summary**()
Show a summary of the model and its parameters.

TODO

TODO: though maybe this should be a method of module... model would have to add to it the observation dist

**train_step**(*x_data*, *y_data*)
Perform one training step

**property trainable_variables**
A list of trainable backend variables within this *Module*

## 3.8 Utils

The `utils` module contains utility classes, functions, and settings which ProbFlow uses internally. The sub-modules of `utils` are:

- *settings* - backend, datatype, and sampling settings
- *base* - abstract base classes for ProbFlow objects
- *ops* - backend-independent mathematical operations
- *casting* - backend-independent casting operations
- *initializers* - backend-independent variable initializer functions
- *io* - functions for loading and saving models
- *metrics* - functions for computing various model performance metrics
- *plotting* - functions for plotting distributions, posteriors, etc
- *torch_distributions* - manual implementations of missing torch dists

- *validation* - functions for data type validation

### 3.8.1 Settings

The utils.settings module contains global settings about the backend to use, what sampling method to use, the default device, and default datatype.

#### Backend

Which backend to use. Can be either TensorFlow 2.0 or PyTorch.

- *get_backend()*
- *set_backend()*

#### Datatype

Which datatype to use as the default for parameters. Depending on your model, you might have to set the default datatype to match the datatype of your data.

- *get_datatype()*
- *set_datatype()*

#### Samples

Whether and how many samples to draw from parameter posterior distributions. If `None`, the maximum a posteriori estimate of each parameter will be used. If an integer greater than 0, that many samples from each parameter's posterior distribution will be used.

- *get_samples()*
- *set_samples()*

#### Flipout

Whether to use Flipout where possible.

- *get_flipout()*
- *set_flipout()*

#### Static posterior sampling

Whether or not to use static posterior sampling (i.e., take a random sample from the posterior, but take the same random sample on repeated calls), and control the UUID of the current static sampling regime.

- *get_static_sampling_uuid()*
- *set_static_sampling_uuid()*

### Sampling context manager

A context manager which controls how *Parameters* sample from their variational distributions while inside the context manager.

- *Sampling*

probflow.utils.settings.**get_backend**()
Get which backend is currently being used.

**Returns** **backend** – The current backend

**Return type** str {'tensorflow' or 'pytorch'}

probflow.utils.settings.**set_backend**(*backend*)
Set which backend is currently being used.

**Parameters** **backend** (*str {'tensorflow' or 'pytorch'}*) – The backend to use

probflow.utils.settings.**get_datatype**()
Get the default datatype used for Tensors

**Returns** **dtype** – The current default datatype

**Return type** tf.dtype or torch.dtype

probflow.utils.settings.**set_datatype**(*datatype*)
Set the datatype to use for Tensors

**Parameters** **datatype** (*tf.dtype or torch.dtype*) – The default datatype to use

probflow.utils.settings.**get_samples**()
Get how many samples (if any) are being drawn from parameter posteriors

**Returns** **n** – Number of samples (if any) to draw from parameters' posteriors. Default = None (ie, use the Maximum a posteriori estimate)

**Return type** None or int > 0

probflow.utils.settings.**set_samples**(*samples*)
Set how many samples (if any) to draw from parameter posteriors

**Parameters** **samples** (*None or int > 0*) – Number of samples (if any) to draw from parameters' posteriors.

probflow.utils.settings.**get_flipout**()
Get whether flipout is currently being used where possible.

**Returns** **flipout** – Whether flipout is currently being used where possible while sampling during training.

**Return type** bool

probflow.utils.settings.**set_flipout**(*flipout*)
Set whether to use flipout where possible while sampling during training

**Parameters** **flipout** (*bool*) – Whether to use flipout where possible while sampling during training.

probflow.utils.settings.**get_static_sampling_uuid**()
Get the current static sampling UUID

probflow.utils.settings.**set_static_sampling_uuid**(*uuid_value*)
Set the current static sampling UUID

**class** probflow.utils.settings.**Sampling**(*n=None*, *flipout=None*, *static=None*)

Bases: object

Use sampling while within this context manager.

**Keyword Arguments**

- **n** (*None or int > 0*) – Number of samples (if any) to draw from parameters' posteriors. Default = 1
- **flipout** (*bool*) – Whether to use flipout where possible while sampling during training. Default = False

### Example

To use maximum a posteriori estimates of the parameter values, don't use the sampling context manager:

```
>>> import probflow as pf
>>> param = pf.Parameter()
>>> param()
[0.07226744]
>>> param() # MAP estimate is always the same
[0.07226744]
```

To use a single sample, use the sampling context manager with `n=1`:

```
>>> with pf.Sampling(n=1):
>>>     param()
[-2.2228503]
>>> with pf.Sampling(n=1):
>>>     param() #samples are different
[1.3473024]
```

To use multiple samples, use the sampling context manager and set the number of samples to take with the `n` keyword argument:

```
>>> with pf.Sampling(n=3):
>>>     param()
[[ 0.10457394]
 [ 0.14018342]
 [-1.8649881 ]]
>>> with pf.Sampling(n=5):
>>>     param()
[[ 2.1035051]
 [-2.641631 ]
 [-2.9091313]
 [ 3.5294306]
 [ 1.6596333]]
```

To use static samples - that is, to always return the same samples while in the same context manager - use the sampling context manager with the `static` keyword argument set to `True`:

```
>>> with pf.Sampling(static=True):
>>>     param()
[ 0.10457394]
>>>     param()  # repeated samples yield the same value
[ 0.10457394]
>>> with pf.Sampling(static=True):
```

(continues on next page)

(continued from previous page)

```
>>>     param()  # under a new context manager they yield new samples
[-2.641631]
>>>     param()  # but remain the same while under the same context
[-2.641631]
```

### 3.8.2 Base

The utils.base module contains abstract base classes (ABCs) for all of ProbFlow's classes.

**class** probflow.utils.base.**BaseDistribution**(**args*)

Bases: abc.ABC

Abstract base class for ProbFlow Distributions

**prob**(*y*)

Compute the probability of some data given this distribution

**log_prob**(*y*)

Compute the log probability of some data given this distribution

**cdf**(*y*)

Cumulative probability of some data along this distribution

**mean**()

Compute the mean of this distribution

Note that this uses the mode of distributions for which the mean is undefined (for example, a categorical distribution)

**mode**()

Compute the mode of this distribution

**sample**(*n=1*)

Generate a random sample from this distribution

**class** probflow.utils.base.**BaseParameter**(**args*)

Bases: abc.ABC

Abstract base class for ProbFlow Parameters

**abstract kl_loss**()

Compute the sum of the Kullback–Leibler divergences between this parameter's priors and its variational posteriors.

**abstract posterior_mean**()

Get the mean of the posterior distribution(s).

**abstract posterior_sample**()

Get the mean of the posterior distribution(s).

**abstract prior_sample**()

Get the mean of the posterior distribution(s).

**class** probflow.utils.base.**BaseModule**(**args*)

Bases: abc.ABC

Abstract base class for ProbFlow Modules

**class** probflow.utils.base.**BaseDataGenerator**(**args*)

Bases: abc.ABC

Abstract base class for ProbFlow DataGenerators

**on_epoch_start**()
Will be called at the start of each training epoch

**on_epoch_end**()
Will be called at the end of each training epoch

**abstract property n_samples**
Number of samples in the dataset

**abstract property batch_size**
Number of samples to generate each minibatch

**class** probflow.utils.base.**BaseCallback**(**args*)
Bases: abc.ABC

Abstract base class for ProbFlow Callbacks

**abstract on_epoch_start**()
Will be called at the start of each training epoch

**abstract on_epoch_end**()
Will be called at the end of each training epoch

**abstract on_train_end**()
Will be called at the end of training

### 3.8.3 Ops

The utils.ops module contains operations which run using the current backend.

- *kl_divergence()*
- expand_dims()
- squeeze()
- *ones()*
- *zeros()*
- *full()*
- *randn()*
- *rand_rademacher()*
- *shape()*
- *eye()*
- *sum()*
- *prod()*
- *mean()*
- *std()*
- *round()*
- *abs()*
- *square()*
- *sqrt()*

- *exp()*
- *relu()*
- *softplus()*
- *sigmoid()*
- *gather()*
- *cat()*
- *additive_logistic_transform()*
- *insert_col_of()*
- *new_variable()*
- *log_cholesky_transform()*
- *copy_tensor()*

---

probflow.utils.ops.**kl_divergence**(*P*, *Q*)
Compute the Kullback–Leibler divergence between two distributions.

**Parameters**

- **P** (tensorflow_probability.distributions.Distribution or torch.distributions.distribution) – The first distribution
- **Q** (tensorflow_probability.distributions.Distribution or torch.distributions.distribution) – The second distribution

**Returns** **kld** – The Kullback–Leibler divergence between P and Q (KL(P || Q))

**Return type** Tensor

probflow.utils.ops.**ones**(*shape*)
Tensor full of ones.

probflow.utils.ops.**zeros**(*shape*)
Tensor full of zeros.

probflow.utils.ops.**full**(*shape*, *value*)
Tensor full of some value.

probflow.utils.ops.**randn**(*shape*)
Tensor full of random values drawn from a standard normal.

probflow.utils.ops.**rand_rademacher**(*shape*)
Tensor full of random -1s or 1s (i.e. drawn from a Rademacher dist).

probflow.utils.ops.**shape**(*x*)
Get a list of integers representing this tensor's shape

probflow.utils.ops.**eye**(*dims*)
Identity matrix.

probflow.utils.ops.**sum**(*val*, *axis=- 1*, *keepdims=False*)
The sum.

probflow.utils.ops.**prod**(*val*, *axis=- 1*, *keepdims=False*)
The product.

probflow.utils.ops.**mean**(*val*, *axis=- 1*, *keepdims=False*)
The mean.

probflow.utils.ops.**std**(*val*, *axis=- 1*, *keepdims=False*)
The uncorrected sample standard deviation.

probflow.utils.ops.**round**(*val*)
Round to the closest integer

probflow.utils.ops.**abs**(*val*)
Absolute value

probflow.utils.ops.**square**(*val*)
Power of 2

probflow.utils.ops.**sqrt**(*val*)
The square root.

probflow.utils.ops.**exp**(*val*)
The natural exponent.

probflow.utils.ops.**relu**(*val*)
Linear rectification.

probflow.utils.ops.**softplus**(*val*)
Linear rectification.

probflow.utils.ops.**sigmoid**(*val*)
Sigmoid function.

probflow.utils.ops.**gather**(*vals*, *inds*, *axis=0*)
Gather values by index

probflow.utils.ops.**cat**(*vals*, *axis=0*)
Concatenate tensors

probflow.utils.ops.**additive_logistic_transform**(*vals*)
The additive logistic transformation

probflow.utils.ops.**insert_col_of**(*vals*, *val*)
Add a column of a value to the left side of a tensor

probflow.utils.ops.**new_variable**(*initial_values*)
Get a new variable with the current backend, and initialize it

probflow.utils.ops.**log_cholesky_transform**(*x*)
Perform the log cholesky transform on a vector of values.

This turns a vector of $\frac{N(N+1)}{2}$ unconstrained values into a valid $N \times N$ covariance matrix.

#### References

- Jose C. Pinheiro & Douglas M. Bates. Unconstrained Parameterizations for Variance-Covariance Matrices *Statistics and Computing*, 1996.

probflow.utils.ops.**copy_tensor**(*x*)
Copy a tensor, detaching it from the gradient/backend/etc/etc

### 3.8.4 Casting

The utils.casting module contains functions for casting back and forth betweeen Tensors and numpy arrays.

- *to_numpy()*
- *to_tensor()*
- to_default_dtype()
- make_input_tensor()

---

probflow.utils.casting.**to_numpy**(*x*)
Convert tensor to numpy array

probflow.utils.casting.**to_tensor**(*x*)
Make x a tensor if not already

### 3.8.5 Initializers

Initializers.

Functions to initialize posterior distribution variables.

- *xavier()* - Xavier initializer
- *scale_xavier()* - Xavier initializer scaled for scale parameters
- *pos_xavier()* - positive-only initizlier

---

probflow.utils.initializers.**xavier**(*shape*)
Xavier initializer

probflow.utils.initializers.**scale_xavier**(*shape*)
Xavier initializer for scale variables

probflow.utils.initializers.**pos_xavier**(*shape*)
Xavier initializer for positive variables

probflow.utils.initializers.**full_of**(*val*)
Get initializer which returns tensor full of single value

### 3.8.6 IO

Functions for saving and loading ProbFlow objects

probflow.utils.io.**dumps**(*obj*)
Serialize a probflow object to a json-safe string.

> **Note:** This removes the compiled _train_fn attribute of a *Model* which is either a TensorFlow or PyTorch compiled function to perform a single training step. Cloudpickle can't serialize it, and after de-serializing will just JIT re-compile if needed.

probflow.utils.io.**loads**(*s*)
Deserialize a probflow object from string

probflow.utils.io.**dump**(*obj*, *filename*)
Serialize a probflow object to file

**Note:** This removes the compiled `_train_fn` attribute of a *Model* which is either a TensorFlow or PyTorch compiled function to perform a single training step. Cloudpickle can't serialize it, and after de-serializing will just JIT re-compile if needed.

probflow.utils.io.**load**(*filename*)
Deserialize a probflow object from file

### 3.8.7 Metrics

Metrics.

Evaluation metrics

- `log_prob()`
- `acc()`
- `accuracy()`
- `mse()`
- `sse()`
- `mae()`

probflow.utils.metrics.**get_metric_fn**(*metric*)
Get a function corresponding to a metric string

### 3.8.8 Plotting

Plotting utilities.

TODO: more info...

probflow.utils.plotting.**approx_kde**(*data*, *bins=500*, *bw=0.075*)
A fast approximation to kernel density estimation.

probflow.utils.plotting.**get_next_color**(*def_color*, *ix*)
Get the next color in the color cycle

probflow.utils.plotting.**get_ix_label**(*ix*, *shape*)
Get a string representation of the current index

probflow.utils.plotting.**plot_dist**(*data*, *xlabel=''*, *style='fill'*, *bins=20*, *ci=0.0*, *bw=0.075*, *alpha=0.4*, *color=None*, *legend=True*)
Plot the distribution of samples.

**Parameters**

- **data** (*ndarray*) – Samples to plot. Should be of size (Nsamples,...)
- **xlabel** (*str*) – Label for the x axis
- **style** (*str*) – Which style of plot to create. Available types are:

  - `'fill'` - filled density plot (the default)
  - `'line'` - line density plot
  - `'hist'` - histogram
- **bins** (int or list or `ndarray`) – Number of bins to use for the histogram (if `kde=False`), or a list or vector of bin edges.
- **ci** (*float between 0 and 1*) – Confidence interval to plot. Default = 0.0 (i.e., not plotted)
- **bw** (*float*) – Bandwidth of the kernel density estimate (if using `style='line'` or `style='fill'`). Default is 0.075
- **alpha** (*float between 0 and 1*) – Transparency of the plot (if `style``='fill'`` or `'hist'`)
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle
- **legend** (*bool*) – Whether to show legends for plots with >1 distribution Default = True

`probflow.utils.plotting.`**plot_line**(*xdata*, *ydata*, *xlabel=''*, *ylabel=''*, *fmt='-'*, *color=None*)

Plot lines.

**Parameters**

- **xdata** (`ndarray`) – X values of points to plot. Should be vector of length `Nsamples`.
- **ydata** (`ndarray`) – Y vaules of points to plot. Should be of size `(Nsamples,...)`.
- **xlabel** (*str*) – Label for the x axis. Default is no x axis label.
- **ylabel** (*str*) – Label for the y axis. Default is no y axis label.
- **fmt** (*str or matplotlib linespec*) – Line marker to use. Default = `'-'` (a normal line).
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle

`probflow.utils.plotting.`**fill_between**(*xdata*, *lb*, *ub*, *xlabel=''*, *ylabel=''*, *alpha=0.3*, *color=None*)

Fill between lines.

**Parameters**

- **xdata** (`ndarray`) – X values of points to plot. Should be vector of length `Nsamples`.
- **lb** (`ndarray`) – Lower bound of fill. Should be of size `(Nsamples,...)`.
- **ub** (`ndarray`) – Upper bound of fill. Should be same size as lb.
- **xlabel** (*str*) – Label for the x axis. Default is no x axis label.
- **ylabel** (*str*) – Label for the y axis. Default is no y axis label.
- **fmt** (*str or matplotlib linespec*) – Line marker to use. Default = `'-'` (a normal line).
- **color** (*matplotlib color code or list of them*) – Color(s) to use to plot the distribution. See https://matplotlib.org/tutorials/colors/colors.html Default = use the default matplotlib color cycle

probflow.utils.plotting.**centered_text**(*text*)
Display text centered in the figure

probflow.utils.plotting.**plot_discrete_dist**(*x*)
Plot histogram of discrete variable

probflow.utils.plotting.**plot_categorical_dist**(*x*)
Plot histogram of categorical variable

probflow.utils.plotting.**plot_by**(*x*, *data*, *bins=30*, *func='mean'*, *plot=True*, *bootstrap=100*, *ci=0.95*, ***kwargs*)
Compute and plot some function func of data as a function of x.

**Parameters**

- **x** (ndarray) – Coordinates of data to plot
- **data** (ndarray) – Data to plot by bins of x
- **bins** (*int*) – Number of bins to bin x into
- **func** (*callable or str*) – Function to apply on elements of data in each x bin. Can be a callable or one of the following str:
  - 'count'
  - 'sum'
  - 'mean'
  - 'median'

  Default = 'mean'
- **plot** (*bool*) – Whether to plot data as a function of x Default = False
- **bootstrap** (*None or int > 0*) – Number of bootstrap samples to use for estimating the uncertainty of the true coverage.
- **ci** (*list of float between 0 and 1*) – Bootstrapped confidence interval percentiles of coverage to show.
- ****kwargs** – Additional arguments are passed to plt.plot or fill_between

**Returns**

- **x_o** (|*ndarray*|) – x bin centers
- **data_o** (|*ndarray*|) – func applied to data values in each x bin

### 3.8.9 Torch Distributions

Torch backend distributions

### 3.8.10 Validation

The utils.validation module contains functions for checking that inputs have the correct type.

- *ensure_tensor_like()*

---

probflow.utils.validation.**ensure_tensor_like**(*obj*, *name*)
Determine whether an object can be cast to a Tensor

The ProbFlow API consists of four primary modules:

- *distributions*
- *parameters*
- *modules*
- *models*

as well as a module which contains pre-built models for standard tasks:

- *applications*

and two other modules which allow for customization of the training process:

- *callbacks*
- *data*

The *distributions* module contains classes to instantiate probability distributions, which describe the likelihood of either a parameter or a datapoint taking any given value. Distribution objects are used to represent the predicted probability distribution of the data, and also the parameters' posteriors and priors.

The *parameters* module contains classes to instantiate parameters, which are values that characterize the behavior of a model. When fitting a model, we want to find the values of the parameters which best allow the model to explain the data. However, with Bayesian modeling we want not only to find the single best value for each parameter, but a probability distribution which describes how likely any given value of a parameter is to be the best or true value.

The *modules* module contains the *Module* abstract base class. Modules are building blocks which can be used to create probabilistic models. They can contain parameters, other information, and even other Modules. They take Tensor(s) as input, perform some computation them, and output a Tensor. A good example of a Module is a neural network layer, because it needs to contain parameters (the weights and biases), and it takes one Tensor as input and outputs a different Tensor. The *modules* module also contains several specific types of Modules, such as a *Dense* neural network layer, a *BatchNormalization* layer and an *Embedding* layer.

The *models* module contains abstract base classes for Models. Unlike Modules, Models encapsulate an entire probabilistic model: they take Tensor(s) as input and output a probability distribution. Like Modules, they can contain Parameters and Modules. They can be fit to data, and have many methods for inspecting the quality of the fit.

The *applications* module contains pre-built models for standard applications (e.g. linear regression, logistic regression, and multilayer dense neural networks) which are ready to be fit.

The *callbacks* module contains the *Callback* abstract base class, and several specific types of callbacks which can be used to control the training process, record metrics or parameter values over the course of training, stop training early, etc.

The *data* module contains the *DataGenerator* class, which can be used to feed data during training when the dataset is too large to fit into memory.

The `utils` module contains various utilities, generally not intended to be used by users, such as abstract base classes, casting functions, settings, etc.

CHAPTER

FOUR

# DEVELOPER GUIDE

At some point I'll fill this in a bit more, but for the time being:

## 4.1 Requirements

First make sure you've got the following installed:

- make
- git
- python3
- virtualenv

## 4.2 Setting up a development environment

To start up an environment to run and test ProbFlow, first make a fork of the ProbFlow github repository. Then, clone your fork to download the repository to your machine (this assumes you're connecting to github using ssh ):

```
git clone git@github.com:<your_github_username>/probflow.git
cd probflow
```

Then, to set up a development environment with tensorflow, run

```
make init-tensorflow
```

or alternatively to set up a dev environment with pytorch,

```
make init-tensorflow
```

The above command creates a new virtual environment called `venv`, activates that virtual environment, installs the requirements (including tensorflow or pytorch), dev requirements, and the ProbFlow package in editable mode from your version of the source code - see the `Makefile` for the commands it's running).

## 4.3 Tests

Then you can edit the source code, which is in `src/probflow`. The tests are in `tests`. To run the tensorflow tests, run

```
make test-tensorflow
```

and to run the PyTorch tests, run

```
make test-pytorch
```

If you get an error during the tests and want to debug, the tests are written using pytest, so to drop into the python debugger on errors, run:

```
. venv/bin/activate
pytest tests/test_you_want_to_run.py --pdb
```

## 4.4 Style

To run the autoformatting (using `isort` and `black`) and style checks (using `flake8`), run

```
make format
```

## 4.5 Documentation

To build the documentation locally (the docs are written for and built with Sphinx, this command creates html files in the `docs/_html` directory, the main page being `docs/_html/index.html`), run:

```
make docs
```

## 4.6 Contributing your changes

Then if you want to contribute your changes, make a pull request!

CHAPTER

FIVE

# BACKLOG

See the github projects page for a list of planned improvements (in the "To do" column, in order of priority from highest at the top to lowest at the bottom), as well as what's being worked on currently in the "In progress tab".

If you're interested in tackling one of them, I'd be thrilled! Pull requests are totally welcome! Also take a look at the *Developer Guide*.

## 5.1 Out of scope

Things which would conceivably be cool, but I think are out of scope for the package:

- Forms of inference aside from stochastic variational inference (e.g. MCMC, Stein methods, Langevin dynamics, Laplace approximation, importance sampling, expectation maximization, particle filters, finding purely the MAP/ML estimate, Bayesian estimation via dropout, etc...)
- Support for sequence models like RNNs and HMMs (though maybe someday will support that, but ProbFlow is aimed more towards supporting tabular/matrix data as opposed to sequence data. Note that you could probably hack together a HMM using tfd.HiddenMarkovModel or an RNN with, say, tf.keras.layers.recurrent_v2.cudnn_lstm or the like)
- Gaussian processes (again, maybe someday, but nonparametric models also don't quite fit in to the ProbFlow framework. Though again one could hack together a GP w/ ProbFlow, for example using tfd.GaussianProcess). See this issue for more info. If you want to use Gaussian processes with TensorFlow, I'd suggest GPflow, or GPyTorch for PyTorch.
- Bayesian networks (though again, could probably manually hack one together in ProbFlow, at least one with a fixed DAG structure where you just want to infer the weights)
- Bayesian model comparison (also maybe someday)
- Backends other than TensorFlow/TFP and PyTorch
- Automatic variational posterior generation (a la Pyro).
- Automatic reparameterizations

ProbFlow is a Python package for building probabilistic Bayesian models with TensorFlow or PyTorch, performing stochastic variational inference with those models, and evaluating the models' inferences. It provides both high-level *Modules* for building Bayesian neural networks, as well as low-level *Parameters* and *Distributions* for constructing custom Bayesian models.

It's very much still a work in progress.

- **Git repository:** http://github.com/brendanhasz/probflow
- **Documentation:** http://probflow.readthedocs.io

- **Bug reports:** http://github.com/brendanhasz/probflow/issues

CHAPTER

SIX

# GETTING STARTED

**ProbFlow** allows you to quickly and less painfully build, fit, and evaluate custom Bayesian models (or *ready-made* ones!) which run on top of either TensorFlow and TensorFlow Probability or PyTorch.

With ProbFlow, the core building blocks of a Bayesian model are parameters and probability distributions (and, of course, the input data). Parameters define how the independent variables (the features) predict the probability distribution of the dependent variables (the target).

For example, a simple Bayesian linear regression

$$y \sim \text{Normal}(wx + b, \sigma)$$

can be built by creating a ProbFlow Model. This is just a class which inherits *Model* (or *ContinuousModel* or *CategoricalModel* depending on the target type). The `__init__` method sets up the parameters, and the `__call__` method performs a forward pass of the model, returning the predicted probability distribution of the target:

TensorFlow

PyTorch

```
import probflow as pf
import tensorflow as tf

class LinearRegression(pf.ContinuousModel):

    def __init__(self):
        self.weight = pf.Parameter(name='weight')
        self.bias = pf.Parameter(name='bias')
        self.std = pf.ScaleParameter(name='sigma')

    def __call__(self, x):
        return pf.Normal(x*self.weight()+self.bias(), self.std())

model = LinearRegression()
```

```
import probflow as pf
import torch

class LinearRegression(pf.ContinuousModel):

    def __init__(self):
        self.weight = pf.Parameter(name='weight')
        self.bias = pf.Parameter(name='bias')
        self.std = pf.ScaleParameter(name='sigma')
```

(continues on next page)

(continued from previous page)

```
    def __call__(self, x):
        x = torch.tensor(x)
        return pf.Normal(x*self.weight()+self.bias(), self.std())

model = LinearRegression()
```

Then, the model can be fit using stochastic variational inference, in *one line*:

```
# x and y are Numpy arrays or pandas DataFrame/Series
model.fit(x, y)
```

You can generate predictions for new data:

```
# x_test is a Numpy array or pandas DataFrame
>>> model.predict(x_test)
[0.983]
```

Compute *probabilistic* predictions for new data, with 95% confidence intervals:

```
model.pred_dist_plot(x_test, ci=0.95)
```

Evaluate your model's performance using various metrics:

```
>>> model.metric('mse', x_test, y_test)
0.217
```

Inspect the posterior distributions of your fit model's parameters, with 95% confidence intervals:

```
model.posterior_plot(ci=0.95)
```

Investigate how well your model is capturing uncertainty by examining how accurate its predictive intervals are:

```
>>> model.pred_dist_coverage(ci=0.95)
0.903
```

and diagnose *where* your model is having problems capturing uncertainty:

```
model.coverage_by(ci=0.95)
```

ProbFlow also provides more complex modules, such as those required for building *Bayesian neural networks*. Also, you can mix ProbFlow with TensorFlow (or PyTorch!) code. For example, even a somewhat complex multi-layer Bayesian neural network like this:

Can be built and fit with ProbFlow in only a few lines:

TensorFlow

PyTorch

```
class DensityNetwork(pf.ContinuousModel):

    def __init__(self, units, head_units):
        self.core = pf.DenseNetwork(units)
        self.mean = pf.DenseNetwork(head_units)
        self.std  = pf.DenseNetwork(head_units)

    def __call__(self, x):
        z = tf.nn.relu(self.core(x))
        return pf.Normal(self.mean(z), tf.exp(self.std(z)))

# Create the model
model = DensityNetwork([x.shape[1], 256, 128], [128, 64, 32, 1])

# Fit it!
model.fit(x, y)
```

```
class DensityNetwork(pf.ContinuousModel):

    def __init__(self, units, head_units):
        self.core = pf.DenseNetwork(units)
        self.mean = pf.DenseNetwork(head_units)
        self.std  = pf.DenseNetwork(head_units)

    def __call__(self, x):
        x = torch.tensor(x)
        z = torch.nn.ReLU()(self.core(x))
        return pf.Normal(self.mean(z), torch.exp(self.std(z)))

# Create the model
model = DensityNetwork([x.shape[1], 256, 128], [128, 64, 32, 1])

# Fit it!
model.fit(x, y)
```

For convenience, ProbFlow also includes several *pre-built models* for standard tasks (such as linear regressions, logistic regressions, and multi-layer dense neural networks). For example, the above linear regression example could have been done with much less work by using ProbFlow's ready-made `LinearRegression` model:

```
model = pf.LinearRegression(x.shape[1])
model.fit(x, y)
```

And a multi-layer Bayesian neural net can be made easily using ProbFlow's ready-made `DenseRegression` model:

```
model = pf.DenseRegression([x.shape[1], 128, 64, 1])
model.fit(x, y)
```

Using parameters and distributions as simple building blocks, ProbFlow allows for the painless creation of more complicated Bayesian models like *generalized linear models*, *deep time-to-event models*, *neural matrix factorization* models, and *Gaussian mixture models*. You can even *mix probabilistic and non-probabilistic models*! Take a look at the *Examples* and the *User Guide* for more!

CHAPTER

SEVEN

# INSTALLATION

If you already have your desired backend installed (i.e. Tensorflow/TFP or PyTorch), then you can just do:

```
pip install probflow
```

Or, to install both ProbFlow and a specific backend,

TensorFlow CPU

TensorFlow GPU

PyTorch

```
pip install probflow[tensorflow]
```

```
pip install probflow[tensorflow_gpu]
```

```
pip install probflow[pytorch]
```

CHAPTER

EIGHT

# SUPPORT

Post bug reports, feature requests, and tutorial requests in GitHub issues.

CHAPTER

NINE

# CONTRIBUTING

Pull requests are totally welcome! Any contribution would be appreciated, from things as minor as pointing out typos to things as major as writing new applications and distributions. For info on how to set up a development environment and run the tests, see the *Developer Guide*.

CHAPTER

TEN

# WHY THE NAME, PROBFLOW?

Because it's a package for probabilistic modeling, and it was built on TensorFlow. ¯_()_/¯

# PYTHON MODULE INDEX

## p

# INDEX

## A



## B

## C

## D

## E



## F

## G



## H



## I



## K

## L

## M

## N

## O

## P

## R

## S

## T

## V



## W



## X

# Z