InferPy Documentation

Release 1.0

Rafael Cabañas

May 08, 2018

Quick Start

develop branch

InferPy is a high-level API for probabilistic modeling written in Python and capable of running on top of Edward and Tensorflow. InferPy's API is strongly inspired by Keras and it has a focus on enabling flexible data processing, easy-to-code probablistic modelling, scalable inference and robust model validation.

Use InferPy is you need a probabilistic programming language that:

- Allows easy and fast prototyping of hierarchical probabilistic models with a simple and user friendly API inspired by Keras.
- Defines probabilistic models with complex probabilistic constructs containing deep neural networks.
- Automatically creates computational efficient batched models without the need to deal with complex tensor operations.
- Run seamlessly on CPU and GPU by relying on Tensorflow.

CHAPTER 1

InferPy is to Edward what Keras is to Tensorflow

InferPy's aim is to be to Edward what Keras is to Tensorflow. Edward is a general purpose probabilistic programing language, like Tensorflow is a general computational engine. But this generality comes a at price. Edward's API is verbose and is based on distributions over Tensor objects, which are n-dimensional arrays with complex semantics operations. Probability distributions over Tensors are powerful abstractions but it is not easy to operate with them. InferPy's API is no so general like Edward's API but still covers a wide range of powerful and widely used probabilistic models, which can contain complex probability constructs containing deep neural networks.

1.1 Getting Started: 30 seconds to InferPy

The core data structures of InferPy is a **probabilistic model**, defined as a set of **random variables** with a conditional independence structure. Like in Edward, a **random varible** is an object parameterized by a set of tensors.

Let's look at a simple (Bayesian) **probabilistic component analysis** model. Graphically the model can be defined as follows,

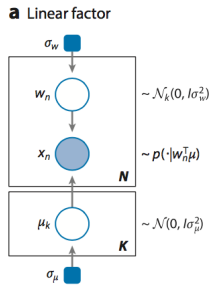

Fig. 1: Bayesian PCA

We start defining the **prior** of the global parameters,

```
import numpy as np
import inferpy as inf
from inferpy.models import Normal, InverseGamma, Dirichlet

# K defines the number of components.
K=10

# d defines the number of dimensions
d=20

#Prior for the principal components
with inf.replicate(size = K)
    mu = Normal(loc = 0, scale = 1, dim = d)
```

InferPy supports the definition of **plateau notation** by using the construct `with inf.replicate(size = K)`, which replicates K times the random variables enclosed within this anotator. Every replicated variable is assumed to be **independent**.

This `with inf.replicate(size = N)` construct is also useful when defining the model for the data:

```
# Number of observations
N = 1000

#data Model
with inf.replicate(size = N):
    # Latent representation of the sample
    w_n = Normal(loc = 0, scale = 1, dim = K)
    # Observed sample. The dimensionality of mu is [K,d].
    x = Normal(loc = mu0 + inf.matmul(w_n,mu), scale = 1.0, observed = true)
```

As commented above, the variable `w_n` and `x_n` are surrounded by a `with` statement to inidicate that the defined random variables will be reapeatedly used in each data sample. In this case, every replicated variable is conditionally idependent given the variables mu and sigma defined outside the `with` statement.

Once the random variables of the model are defined, the probablitic model itself can be created and compiled. The probabilistic model defines a joint probability distribuiton over all these random variables.

```
from inferpy import ProbModel

# Define the model
pca = ProbModel(vars = [mu,w_n,x_n])

# Compile the model
pca.compile(infMethod = 'KLqp')
```

During the model compilation we specify different inference methods that will be used to learn the model.

```
from inferpy import ProbModel

# Define the model
pca = ProbModel(vars = [mu,w_n,x_n])

# Compile the model
pca.compile(infMethod = 'MCMC')
```

The inference method can be further configure. But, as in Keras, a core principle is to try make things reasonbly simple, while allowing the user the full control if needed.

```
from keras.optimizers import SGD

# Define the model
pca = ProbModel(vars = [mu,w_n,x_n])

# Define the optimiser
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

# Define the inference method
infklqp = inf.inference.KLqp(optimizer = sgd, loss="ELBO")

# Compile the model
pca.compile(infMethod = infklqp)
```

Every random variable object is equipped with methods such as `log_prob()` and `sample()`. Similarly, a probabilistic model is also equipped with the same methods. Then, we can sample data from the model anbd compute the log-likelihood of a data set:

```
# Sample data from the model
data = pca.sample(size = 100)

# Compute the log-likelihood of a data set
log_like = probmodel.log_prob(data)
```

Of course, you can fit your model with a given data set:

```
# Fit the model with the given data
pca.fit(data_training, epochs=10)
```

Update your probablistic model with new data using the Bayes' rule:

```
# Update the model with the new data
pca.update(new_data)
```

Query the posterior over a given random varible:

```
# Compute the posterior of a given random variable
mu_post = pca.posterior(mu)
```

Evaluate your model according to a given metric:

```
# Evaluate the model on given test data set using some metric
log_like = pca.evaluate(test_data, metrics = ['log_likelihood'])
```

Or compute predicitons on new data

```
# Make predictions over a target var
latent_representation = pca.predict(test_data, targetvar = w_n)
```

1.2 Guiding Principles

- InferPy's probability distribuions are mainly inherited from TensorFlow Distribuitons package. .. InferPy's API is fully compatible with tf.distributions' API. The 'shape' argument was added as a simplifing option when defining multidimensional distributions.

- InferPy directly relies on top of Edward's inference engine and includes all the inference algorithms included in this package. As Edward's inference engine relies on TensorFlow computing engine, InferPy also relies on it too.
- InferPy seamsly process data contained in a numpy array, Tensorflow's tensor, Tensorflow's Dataset (tf.Data API), Pandas' DataFrame or Apache Spark's DataFrame.
- InferPy also includes novel distributed statistical inference algorithms by combining Tensorflow and Apache Spark computing engines.

1.3 Guide to Building Probabilistic Models

1.3.1 Getting Started with Probabilistic Models

InferPy focuses on *hirearchical probabilistic models* which usually are structured in two different layers:

- A **prior model** defining a joint distribution $p(\theta)$ over the global parameters of the model, θ.
- A **data or observation model** defining a joint conditional distribution $p(x, h|\theta)$ over the observed quantities x and the the local hidden variables h governing the observation x. This data model should be specified in a single-sample basis. There are many models of interest without local hidden variables, in that case we simply specify the conditional $p(x|\theta)$. More flexible ways of defining the data model can be found in ?.

A Bayesian PCA model has the following graphical structure,

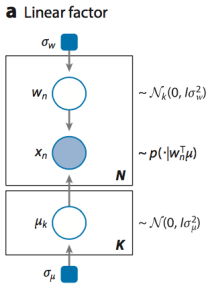

Fig. 2: Bayesian PCA

The **prior model** is the variable μ. The **data model** is the part of the model surrounded by the box indexed by **N**.

And this is how this Bayesian PCA model is denfined in InferPy:

```
import numpy as np
import inferpy as inf
from inferpy.models import Normal, InverseGamma, Dirichlet

import numpy as np
import inferpy as inf
from inferpy.models import Normal, InverseGamma, Dirichlet

# Define the probabilistic model
with inf.ProbModel() as pca:
   # K defines the number of components.
   K=10

   #Prior for the principal components
```

(continues on next page)

(continued from previous page)

```
with inf.replicate(size = K)
    mu = Normal(loc = 0, scale = 1, dime = d)

# Number of observations
N = 1000

#data Model
with inf.replicate(size = N):
    # Latent representation of the sample
    w_n = Normal(loc = 0, scale = 1, dim = K)
    # Observed sample. The dimensionality of mu is [K,d].
    x = Normal(loc = inf.matmul(w_n,mu), scale = 1.0, observed = true)

#compile the probabilistic model
pca.compile()
```

The `with inf.replicate(size = N)` sintaxis is used to replicate the random variables contained within this construct. It follows from the so-called *plateau notation* to define the data generation part of a probabilistic model. Every replicated variable is **conditionally idependent** given the previous random variables (if any) defined outside the **with** statement.

1.3.2 Random Variables

Following Edward's approach, a random variable x is an object parametrized by a tensor θ (i.e. a TensorFlow's tensor or numpy's ndarray). The number of random variables in one object is determined by the dimensions of its parameters (like in Edward) or by the 'shape' or 'dim' argument (inspired by PyMC3 and Keras):

```
# matrix of [1, 5] univariate standard normals
x  = Normal(loc = 0, scale = 1, dim = 5)

# matrix of [1, 5] univariate standard normals
x  = Normal(loc = np.zeros(5), scale = np.ones(5))

# matrix of [1,5] univariate standard normals
x = Normal (loc = 0, scale = 1, shape = [1,5])
```

The `with inf.replicate(size = N)` sintaxis can also be used to define multi-dimensional objects:

```
# matrix of [10,5] univariate standard normals
with inf.replicate(size = 10)
    x = Normal (loc = 0, scale = 1, dim = 5)
```

Multivariate distributions can be defined similarly. Following Edward's approach, the multivariate dimension is the innermost (right-most) dimension of the parameters.

```
# Object with five K-dimensional multivariate normals, shape(x) = [5,K]
x  = MultivariateNormal(loc = np.zeros([5,K]), scale = np.ones([5,K,K]))

# Object with five K-dimensional multivariate normals, shape(x) = [5,K]
x = MultivariateNormal (loc = np.zeros(K), scale = np.ones([K,K]), shape = [5,K])
```

The argument `observed = true` in the constructor of a random variable is used to indicate whether a variable is observable or not.

1.3.3 Probabilistic Models

A **probabilistic model** defines a joint distribution over observable and non-observable variables, $p(\theta, \mu, \sigma, z_n, x_n)$ for the running example. The variables in the model are the ones defined using the `with inf.ProbModel() as pca:` construct. Alternatively, we can also use a builder,

```
from inferpy import ProbModel
pca = ProbModel(vars = [mu,w_n,x_n])
pca.compile()
```

The model must be **compiled** before it can be used.

Like any random variable object, a probabilistic model is equipped with methods such as `log_prob()` and `sample()`. Then, we can sample data from the model and compute the log-likelihood of a data set:

```
data = probmodel.sample(size = 1000)
log_like = probmodel.log_prob(data)
```

Folowing Edward's approach, a random variable x is associated to a tensor x^* in the computational graph handled by TensorFlow, where the computations takes place. This tensor x^* contains the samples of the random variable x, i.e. $x^* \sim p(x|\theta)$. In this way, random variables can be involved in expressive deterministic operations. For example, the following piece of code corresponds to a zero inflated linear regression model

```
#Prior
w = Normal(0, 1, dim=d)
w0 = Normal(0, 1)
p = Beta(1,1)

#Likelihood model
with inf.replicate(size = 1000):
    x = Normal(0,1000, dim=d, observed = true)
    h = Binomial(p)
    y0 = Normal(w0 + inf.matmul(x,w, transpose_b = true), 1),
    y1 = Delta(0.0)
    y = Deterministic(h*y0 + (1-h)*y1, observed = true)

probmodel = ProbModel(vars = [w,w0,p,x,h,y0,y1,y])
probmodel.compile()
data = probmodel.sample(size = 1000)
probmodel.fit(data)
```

A special case, it is the inclusion of deep neural networks within our probabilistic model to capture complex non-linear dependencies between the random variables. This is extensively treated in the the Guide to Bayesian Deep Learning.

Finally, a probablistic model have the following methods:

- `probmodel.summary()`: prints a summary representation of the model.
- `probmodel.get_config()`: returns a dictionary containing the configuration of the model. The model can be reinstantiated from its config via:

```
config = probmodel.get_config()
probmodel = ProbModel.from_config(config)
```

- `model.to_json()`: returns a representation of the model as a JSON string. Note that the representation does not include the weights, only the architecture. You can reinstantiate the same model (with reinitialized weights) from the JSON string via: ‘‘‘python from models import model_from_json

```
json_string = model.to_json()
model = model_from_json(json_string)
```

1.3.4 Supported Probability Distributions

1.4 Guide to Approximate Inference

1.4.1 Getting Started with Approximate Inference

The API defines the set of algorithms and methods used to perform inference in a probabilistic model $p(x, z, \theta)$ (where x are the observations, z the local hidden variibles, and θ the global parameters of the model). More precisely, the inference problem reduces to compute the posterior probability over the latent variables given a data sample $p(z, \theta|x_{train})$, because by looking at these posteriors we can uncover the hidden structure in the data. For the running example, the posterior over the local hidden variables $p(w_n|x_{train})$ tell us the latent vector representation of the sample x_n, while the posterior over the global variables $p(\mu|x_{train})$ tells us which is the affine transformation between the latent space and the observable space.

InferPy inherits Edward's approach an consider approximate inference solutions,

$$q(z, \theta) \approx p(z, \theta|x_{train})$$

in which the task is to approximate the posterior $p(z, \theta|x_{train})$ using a family of distributions, $q(z, \theta; \lambda)$, indexed by a parameter vector λ.

A probabilistic model in InferPy should be compiled before we can access these posteriors,

```
pca = ProbModel(vars = [mu,w_n,x_n])
pca.compile(infMethod = 'KLqp')
pca.fit(x_train)
posterior_mu = pca.posterior(mu)
```

The compilation process allows to choose the inference algorithm through the `infMethod` argument. In the above example we use `'Klqp'`. Other inference algorithms include: `'NUTS'`, `'MCMC'`, `'KLpq'`, etc. Look at `?` for a detailed description of the available inference algorithms.

Following InferPy guiding principles, users can further configure the inference algorithm. First, they can define a model 'Q' for approximating the posterior distribution,

```
pca = ProbModel(vars = [mu,w_n,x_n])

q_mu = inf.inference.Q.Normal(bind = mu, initializer='random_unifrom')
q_w_n = inf.inference.Q.Normal(bind = w_n, initializer='random_unifrom')

qmodel = QModel(vars = [q_mu,q_w_n])

pca.compile(infMethod = 'KLqp', Q = qmodel)
pca.fit(x_train)
posterior_mu = pca.posterior(mu)
```

In the 'Q' model we should include a q distribution for every non observed variable in the 'P' model. Otherwise, an error will be raised during model compilation.

By default, the posterior **q** belongs to the same distribution family than **p** , but in the above example we show how we can change that (e.g. we set the posterior over **mu** to obtain a point mass estimate instead of the Gaussian approximation used by default). We can also configure how these **q's** are initialized using any of the Keras's initializers.

Inspired by Keras semantics, we can furhter configure the inference algorithm,

```
pca = ProbModel(vars = [mu,w_n,x_n])

q_mu = inf.inference.Q.PointMass(bind = mu, initializer='zeroes')
q_w_n = inf.inference.Q.Normal(bind = w_n, initializer='random_unifrom')

qmodel = QModel(vars = [q_mu,q_w_n])

sgd = keras.optimizers.SGD(lr=0.01, momentum=0.9, nesterov=True)
infkl_qp = inf.inference.KLqp(Q = qmodel, optimizer = sgd, loss="ELBO")
pca.compile(infMethod = infkl_qp)

pca.fit(x_train)
posterior_mu = pca.posterior(mu)
```

Have a look at Inference Zoo to explore other configuration options.

1.4.2 Compositional Inference

InferPy directly builds on top of Edward's compositionality idea to design complex infererence algorithms.

```
pca = ProbModel(vars = [mu,w_n,x_n])

q_mu = inf.inference.Q.PointMass(bind = mu, initializer='zeroes')
q_w_n = inf.inference.Q.Normal(bind = w_n, initializer='random_unifrom')

qlocal = QModel(vars = [q_w_n])
qglobal = QModel(vars = [mu])

infkl_qp = inf.inference.KLqp(Q = qlocal, optimizer = 'sgd', innerIter = 10)
infMAP = inf.inference.MAP(Q = qglobal, optimizer = 'sgd')

sgd = keras.optimizers.SGD(lr=0.01, momentum=0.9, nesterov=True)
infkl_qp = inf.inference.KLqp(Q = qmodel, optimizer = sgd, loss="ELBO")
probmodel.compile(infMethod = [infkl_qp,infMAP])

pca.fit(x_train)
posterior_mu = pca.posterior(mu)
```

With the above sintaxis, we perform a variational EM algorithm, where the E step is repeated 10 times for every MAP step.

More flexibility is also available by defining how each mini-batch is processed by the inference algorithm. The following piece of code is equivalent to the above one,

```
pca = ProbModel(vars = [mu,w_n,x_n])

q_mu = inf.inference.Q.PointMass(bind = mu, initializer='zeroes')
q_w_n = inf.inference.Q.Normal(bind = w_n, initializer='random_unifrom')

qlocal = QModel(vars = [q_w_n])
qglobal = QModel(vars = [mu])

infkl_qp = inf.inference.KLqp(Q = qlocal, optimizer = 'sgd', innerIter = 10)
infMAP = inf.inference.MAP(Q = qglobal, optimizer = 'sgd')

```

(continues on next page)

(continued from previous page)

```
emAlg = lambda (infMethod, dataBatch):
   for _ in range(10)
       infMethod[0].update(data = dataBatch)

   infMethod[1].update(data = dataBatch)
   return

pca.compile(infMethod = [infkl_qp,infMAP], ingAlg = emAlg)

pca.fit(x_train, EPOCHS = 10)
posterior_mu = pca.posterior(mu)
```

Have a look again at Inference Zoo to explore other complex compositional options.

1.4.3 Supported Inference Methods

1.5 Guide to Bayesian Deep Learning

InferPy inherits Edward's approach for representing probabilistic models as (stochastic) computational graphs. As describe above, a random variable x is associated to a tensor x^* in the computational graph handled by TensorFlow, where the computations takes place. This tensor x^* contains the samples of the random variable x, i.e. $x^* \sim p(x|\theta)$. In this way, random variables can be involved in complex deterministic operations containing deep neural networks, math operations and another libraries compatible with Tensorflow (such as Keras).

Bayesian deep learning or deep probabilistic programming enbraces the idea of employing deep neural networks within a probabilistic model in order to capture complex non-linear dependencies between variables.

InferPy's API gives support to this powerful and flexible modelling framework. Let us start by showing how a variational autoencoder over binary data can be defined by mixing Keras and InferPy code.

```
from keras.models import Sequential
from keras.layers import Dense, Activation

M = 1000
dim_z = 10
dim_x = 100

#Define the decoder network
input_z  = keras.layers.Input(input_dim = dim_z)
layer = keras.layers.Dense(256, activation = 'relu')(input_z)
output_x = keras.layers.Dense(dim_x)(layer)
decoder_nn = keras.models.Model(inputs = input, outputs = output_x)

#define the generative model
with inf.replicate(size = N)
 z = Normal(0,1, dim = dim_z)
 x = Bernoulli(logits = decoder_nn(z.value()), observed = true)

#define the encoder network
input_x  = keras.layers.Input(input_dim = d_x)
layer = keras.layers.Dense(256, activation = 'relu')(input_x)
output_loc = keras.layers.Dense(dim_z)(layer)
output_scale = keras.layers.Dense(dim_z, activation = 'softplus')(layer)
encoder_loc = keras.models.Model(inputs = input, outputs = output_mu)
```

(continues on next page)

(continued from previous page)

```
encoder_scale = keras.models.Model(inputs = input, outputs = output_scale)

#define the Q distribution
q_z = Normal(loc = encoder_loc(x.value()), scale = encoder_scale(x.value()))

#compile and fit the model with training data
probmodel.compile(infMethod = 'KLqp', Q = {z : q_z})
probmodel.fit(x_train)

#extract the hidden representation from a set of observations
hidden_encoding = probmodel.predict(x_pred, targetvar = z)
```

In this case, the parameters of the encoder and decoder neural networks are automatically managed by Keras. These parameters are them treated as model parameters and not exposed to the user. In consequence, we can not be Bayesian about them by defining specific prior distributions. In this example (?) , we show how we can avoid that by introducing extra complexity in the code.

Other examples of probabilisitc models using deep neural networks are: - Bayesian Neural Networks - Mixture Density Networks - ...

We can also define a Keras model whose input is an observation and its output its the expected value of the posterior over the hidden variables, $E[p(z|x)]$, by using the method `toKeras`, as a way to create more expressive models.

```
from keras.layers import Conv2D, MaxPooling2D, Flatten
from keras.layers import Input, LSTM, Embedding, Dense
from keras.models import Model, Sequential

#We define a Keras' model whose input is data sample 'x' and the output is the␣
↪encoded vector E[p(z|x)]
variational_econder_keras = probmodel.toKeras(targetvar = z)

vision_model = Sequential()
vision_model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
vision_model.add(Conv2D(64, (3, 3), activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Flatten())

# Now let's get a tensor with the output of our vision model:
encoded_image = vision_model(input_x)

# Let's concatenate the vae vector and the convolutional image vector:
merged = keras.layers.concatenate([variational_econder_keras, encoded_image])

# And let's train a logistic regression over 100 categories on top:
output = Dense(100, activation='softmax')(merged)

# This is our final model:
classifier = Model(inputs=[input_x], outputs=output)

# The next stage would be training this model on actual data.
```

1.6 Guide to Model Validation

Model validation try to assess how faifhfully the inferered probabilistic model represents and explain the observed data.

The main tool for model validation consists on analyzing the posterior predictive distribution,

$p(y_{test}, x_{test}|y_{train}, x_{train}) = \int p(y_{test}, x_{test}|z, \theta)p(z, \theta|y_{train}, x_{train})dzd\theta$

This posterior predictive distribution can be used to measure how well the model fits an independent dataset using the test marginal log-likelihood, $\ln p(y_{test}, x_{test}|y_{train}, x_{train})$,

```
log_like = probmodel.evaluate(test_data, metrics = ['log_likelihood'])
```

In other cases, we may need to evalute the predictive capacity of the model with respect to some target variable y,

$p(y_{test}|x_{test}, y_{train}, x_{train}) = \int p(y_{test}|x_{test}, z, \theta)p(z, \theta|y_{train}, x_{train})dzd\theta$

So the metrics can be computed with respect to this target variable by using the 'targetvar' argument,

```
log_like, accuracy, mse = probmodel.evaluate(test_data, targetvar = y, metrics = [
↪'log_likelihood', 'accuracy', 'mse'])
```

So, the log-likelihood metric as well as the accuracy and the mean square error metric are computed by using the predictive posterior $p(y_{test}|x_{test}, y_{train}, x_{train})$.

Custom evaluation metrics can also be defined,

```
def mean_absolute_error(posterior, observations, weights=None):
    predictions = tf.map_fn(lambda x : x.getMean(), posterior)
    return tf.metrics.mean_absolute_error(observations, predictions, weights)

mse, mae = probmodel.evaluate(test_data, targetvar = y, metrics = ['mse', mean_
↪absolute_error])
```

1.7 Guide to Data Handling

```
import numpy as np
import inferpy as inf
from inferpy.models import Normal, InverseGamma, Dirichlet

#We first define the probabilistic model
with inf.ProbModel() as mixture_model:
    # K defines the number of components.
    K=10
    #Prior for the means of the Gaussians
    mu = Normal(loc = 0, scale = 1, shape=[K,d])
    #Prior for the precision of the Gaussians
    invgamma = InverseGamma(concentration = 1, rate = 1, shape=[K,d])
    #Prior for the mixing proportions
    theta = Dirichlet(np.ones(K))

    # Number of observations
    N = 1000
    #data Model
    with inf.replicate(size = N, batch_size = 100)
        # Sample the component indicator of the mixture. This is a latent variable␣
↪that can not be observed
        z_n = Multinomial(probs = theta)
        # Sample the observed value from the Gaussian of the selected component.
        x_n = Normal(loc = tf.gather(mu,z_n), scale = tf.gather(invgamma,z_n),␣
↪observed = true)
```

(continues on next page)

(continued from previous page)

```
#compile the probabilistic model
mixture_model.compile(infAlg = 'klqp')

#fit the model with data
mixture_model.fit(data)
```

1.8 Probabilistic Model Zoo

1.8.1 Bayesian Linear Regression

```
# Shape = [1,d]
w = Normal(0, 1, dim=d)
# Shape = [1,1]
w0 = Normal(0, 1)

with inf.replicate(size = N):
    # Shape = [N,d]
    x = Normal(0,1, dim=d, observed = true)
    # Shape = [1,1] + [N,d]@[d,1] = [1,1] + [N,1] = [N,1] (by broadcasting)
    y = Normal(w0 + tf.matmul(x,w, transpose_b = true ), 1, observed = true)

model = ProbModel(vars = [w0,w,x,y])

data = model.sample(size=N)

log_prob = model.log_prob(sample)

model.compile(infMethod = 'KLqp')

model.fit(data)

print(probmodel.posterior([w0,w]))
```

1.8.2 Zero Inflated Linear Regression

```
# Shape = [1,d]
w = Normal(0, 1, dim=d)
# Shape = [1,1]
w0 = Normal(0, 1)

# Shape = [1,1]
p = Beta(1,1)

with inf.replicate(size = N):
    # Shape [N,d]
    x = Normal(0,1000, dim=d, observed = true)
    # Shape [N,1]
    h = Binomial(p)
```

(continues on next page)

(continued from previous page)

```
    # Shape [1,1] + [N,d]@[d,1] = [1,1] + [N,1] = [N,1] (by broadcasting)
    y0 = Normal(w0 + inf.matmul(x,w, transpose_b = true ), 1),
    # Shape [N,1]
    y1 = Delta(0.0)
    # Shape [N,1]*[N,1] + [N,1]*[N,1] = [N,1]
    y = Deterministic(h*y0 + (1-h)*y1, observed = true)

model = ProbModel(vars = [w0,w,p,x,h,y0,y1,y])

data = model.sample(size=N)

log_prob = model.log_prob(sample)

model.compile(infMethod = 'KLqp')

model.fit(data)

print(probmodel.posterior([w0,w]))
```

1.8.3 Bayesian Logistic Regression

```
# Shape = [1,d]
w = Normal(0, 1, dim=d)
# Shape = [1,1]
w0 = Normal(0, 1)

with inf.replicate(size = N):
    # Shape = [N,d]
    x = Normal(0,1, dim=d, observed = true)
    # Shape = [1,1] + [N,d]@[d,1] = [1,1] + [N,1] = [N,1] (by broadcasting)
    y = Binomial(logits = w0 + tf.matmul(x,w, transpose_b = true), observed = true)

model = ProbModel(vars = [w0,w,x,y])

data = model.sample(size=N)

log_prob = model.log_prob(sample)

model.compile(infMethod = 'KLqp')

model.fit(data)

print(probmodel.posterior([w0,w]))
```

1.8.4 Bayesian Multinomial Logistic Regression

```
# Number of classes
K=10
```

(continues on next page)

(continued from previous page)

```
with inf.replicate(size = K):
    # Shape = [K,d]
    w = Normal(0, 1, dim=d)
    # Shape = [K,1]
    w0 = Normal(0, 1])

with inf.replicate(size = N):
    # Shape = [N,d]
    x = Normal(0,1, dim=d, observed = true)
    # Shape = [1,K] + [N,d]@[d,K] = [1,K] + [N,K] = [N,K] (by broadcasting)
    y = Multinmial(logits = tf.transpose(w0) + tf.matmul(x,w, transpose_b = true),␣
↪observed = true)

model = ProbModel(vars = [w0,w,x,y])

data = model.sample(size=N)

log_prob = model.log_prob(sample)

model.compile(infMethod = 'KLqp')

model.fit(data)

print(probmodel.posterior([w0,w]))
```

1.8.5 Mixture of Gaussians

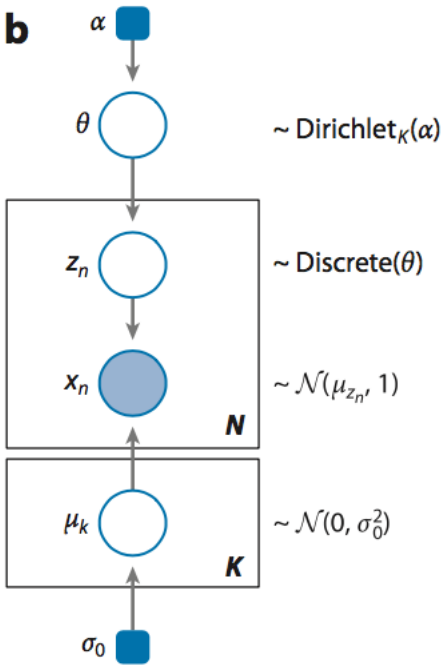

Fig. 3: Mixture of Gaussians

Version A

```
d=3
K=10
N=1000
#Prior
with inf.replicate(size = K):
    #Shape [K,d]
    mu = Normal(loc = 0, scale =1, dim=d)
    #Shape [K,d]
    sigma = InverseGamma(concentration = 1, rate = 1, dim=d)

# Shape [1,K]
p = Dirichlet(np.ones(K))

#Data Model
with inf.replicate(size = N):
    # Shape [N,1]
    z_n = Multinomial(probs = p)
    # Shape [N,d]
    x_n = Normal(loc = tf.gather(mu,z_n), scale = tf.gather(sigma,z_n), observed =␣
↪true)

model = ProbModel(vars = [p,mu,sigma,z_n, x_n])

data = model.sample(size=N)

log_prob = model.log_prob(sample)

model.compile(infMethod = 'KLqp')

model.fit(data)

print(probmodel.posterior([mu,sigma]))
```

Version B

```
d=3
K=10
N=1000
#Prior
mu = Normal(loc = 0, scale =1, shape = [K,d])
sigma = InverseGamma(concentration = 1, rate = 1, shape = [K,d])

# Shape [1,K]
p = Dirichlet(np.ones(K))

#Data Model
z_n = Multinomial(probs = p, shape = [N,1])
# Shape [N,d]
x_n = Normal(loc = tf.gather(mu,z_n), scale = tf.gather(sigma,z_n), observed = true)

probmodel = ProbModel(vars = [p,mu,sigma,z_n, x_n])

data = probmodel.sample(size=N)

log_prob = probmodel.log_prob(sample)
```

(continues on next page)

(continued from previous page)

```
probmodel.compile(infMethod = 'KLqp')

probmodel.fit(data)

print(probmodel.posterior([mu,sigma]))
```

1.8.6 Linear Factor Model (PCA)

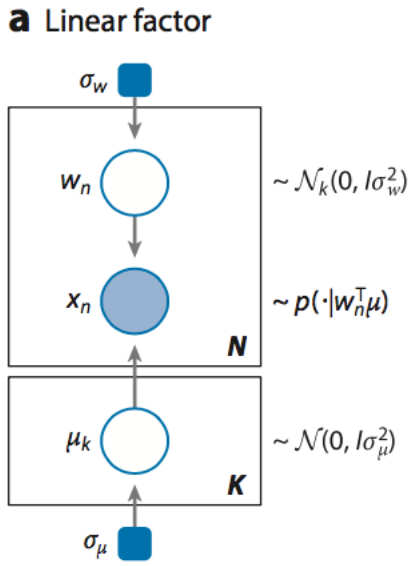

Fig. 4: Linear Factor Model

```
K = 5
d = 10
N=200

with inf.replicate(size = K)
    # Shape [K,d]
    mu = Normal(0,1, dim = d)

# Shape [1,d]
mu0 = Normal(0,1, dim = d)

sigma = 1.0

with inf.replicate(size = N):
    # Shape [N,K]
    w_n = Normal(0,1, dim = K)
    # inf.matmul(w_n,mu) has shape [N,K] x [K,d] = [N,d] by broadcasting mu.
    # Shape [1,d] + [N,d] = [N,d] by broadcasting mu0
    x = Normal(mu0 + inf.matmul(w,mu), sigma, observed = true)

probmodel = ProbModel([mu,mu0,w_n,x])
```

(continues on next page)

(continued from previous page)

```
data = probmodel.sample(size=N)

log_prob = probmodel.log_prob(sample)

probmodel.compile(infMethod = 'KLqp')

probmodel.fit(data)

print(probmodel.posterior([mu,mu0]))
```

1.8.7 PCA with ARD Prior (PCA)

```
K = 5
d = 10
N=200

with inf.replicate(size = K)
    # Shape [K,d]
    alpha = InverseGamma(1,1, dim = d)
    # Shape [K,d]
    mu = Normal(0,1, dim = d)

# Shape [1,d]
mu0 = Normal(0,1, dim = d)

# Shape [1,1]
sigma = InverseGamma(1,1, dim = 1)

with inf.replicate(size = N):
    # Shape [N,K]
    w_n = Normal(0,1, dim = K)
    # inf.matmul(w_n,mu) has shape [N,K] x [K,d] = [N,d] by broadcasting mu.
    # Shape [1,d] + [N,d] = [N,d] by broadcasting mu0
    x = Normal(mu0 + inf.matmul(w,mu), sigma, observed = true)

probmodel = ProbModel([alpha,mu,mu0,sigma,w_n,x])

data = probmodel.sample(size=N)

log_prob = probmodel.log_prob(sample)

probmodel.compile(infMethod = 'KLqp')

probmodel.fit(data)

print(probmodel.posterior([alpha,mu,mu0,sigma]))
```

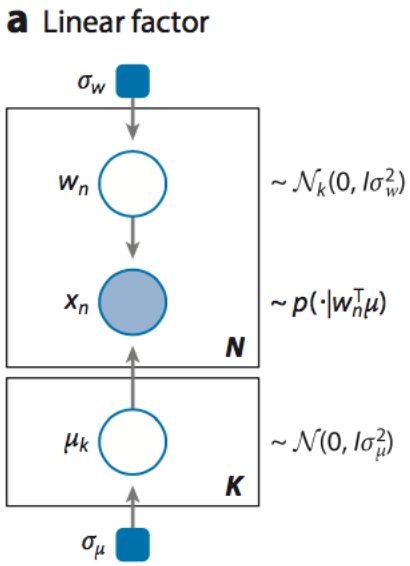

Fig. 5: Mixed Membership Model

1.8.8 Mixed Membership Model

```
K = 5
d = 10
N=200
M=50

with inf.replicate(size = K)
    #Shape = [K,d]
    mu = Normal(0,1, dim = d)
    #Shape = [K,d]
    sigma = InverseGamma(1,1, dim = d)

with inf.replicate(size = N):
    #Shape = [N,K]
    theta_n = Dirichlet(np.ones(K))
    with inf.replicate(size = M):
        # Shape [N*M,1]
        z_mn = Multinomial(theta_n)
        # Shape [N*M,d]
        x = Normal(tf.gather(mu,z_mn), tf.gather(sigma,z_mn), observed = true)

probmodel = ProbModel([mu,sigma,theta_n,z_mn,x])

data = probmodel.sample(size=N)

log_prob = probmodel.log_prob(sample)

probmodel.compile(infMethod = 'KLqp')

probmodel.fit(data)

print(probmodel.posterior([mu,sigma]))
```

1.8.9 Latent Dirichlet Allocation

```
K = 5 # Number of topics
d = 1000 # Size of vocabulary
N=200 # Number of documents in the corpus
M=50 # Number of words in each document

with inf.replicate(size = K)
    #Shape = [K,d]
    dir = Dirichlet(np.ones(d)*0.1)

with inf.replicate(size = N):
    #Shape = [N,K]
    theta_n = Dirichlet(np.ones(K))
    with inf.replicate(size = M):
        # Shape [N*M,1]
        z_mn = Multinomial(theta_n)
        # Shape [N*M,d]
        x = Multinomial(tf.gather(dir,z_mn), tf.gather(dir,z_mn), observed = true)

probmodel = ProbModel([dir,theta_n,z_mn,x])

data = probmodel.sample(size=N)

log_prob = probmodel.log_prob(sample)

probmodel.compile(infMethod = 'KLqp')

probmodel.fit(data)

print(probmodel.posterior(dir))
```

1.8.10 Matrix Factorization

Version A

```
N=200
M=50
K=5

with inf.replicate(name = 'A', size = M)
    # Shape [M,K]
    gamma_m = Normal(0,1, dim = K)

with inf.replicate(name = 'B', size = N):
    # Shape [N,K]
    w_n = Normal(0,1, dim = K)

with inf.replicate(compound = ['A', 'B']):
    # x_mn has shape [N,K] x [K,M] = [N,M]
```

(continues on next page)

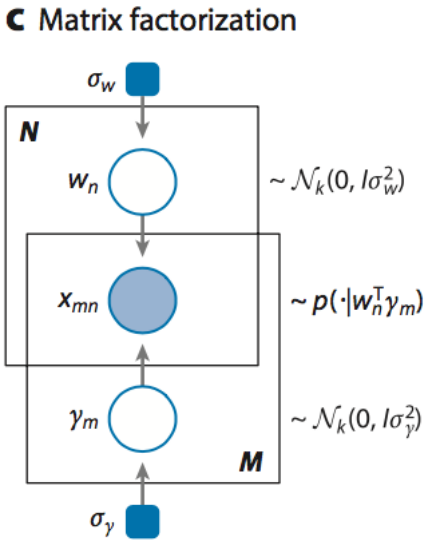

Fig. 6: Matrix Factorization Model

(continued from previous page)

```
    x_nm = Normal(tf.matmul(w_n,gamma_m, transpose_b = true), 1, observed = true)


probmodel = ProbModel([w_n,gamma_m,x_nm])

data = probmodel.sample(size=N)

log_prob = probmodel.log_prob(sample)

probmodel.compile(infMethod = 'KLqp')

probmodel.fit(data)

print(probmodel.posterior([w_n,gamma_m]))
```

Version B

```
N=200
M=50
K=5

# Shape [M,K]
gamma_m = Normal(0,1, shape = [M,K])

# Shape [N,K]
w_n = Normal(0,1, shape = [N,K])

# x_mn has shape [N,K] x [K,M] = [N,M]
x_nm = Normal(tf.matmul(w_n,gamma_m, transpose_b = true), 1, observed = true)

probmodel = ProbModel([w_n,gamma_m,x_nm])

data = probmodel.sample(size=N)
```

(continues on next page)

(continued from previous page)

```
log_prob = probmodel.log_prob(sample)

probmodel.compile(infMethod = 'KLqp')

probmodel.fit(data)

print(probmodel.posterior([w_n,gamma_m]))
```

1.8.11 Linear Mixed Effect Model

```
N = 1000 # number of observations
n_s = 100 # number of students
n_d = 10 # number of instructor
n_dept = 10 # number of departments

eta_s = Normal(0,1, dim = n_s)
eta_d = Normal(0,1, dim = n_d)
eta_dept = Normal(0,1, dim = n_dept)
mu = Normal(0,1)
mu_service = Normal(0,1)

with inf.replicate( size = N):
    student = Multinomial(probs = np.rep(1,n_s)/n_s, observed = true)
    instructor = Multinomial(probs = np.rep(1,n_d)/n_d, observed = true)
    department = Multinomial(probs = np.rep(1,n_dept)/n_dept, observed = true)
    service = Binomial (probs = 0.5, observed = true)
    y = Normal (tf.gather(eta_s,student)
                + bs.gather(eta_d,instructor)
                + bs.gather(eta_dept,department)
                +  mu + mu_service*service, 1, observed = true)

#vars = 'all' automatically add all previously created random variables
probmodel = ProbModel(vars = 'all')

data = probmodel.sample(size=N)

log_prob = probmodel.log_prob(sample)

probmodel.compile(infMethod = 'KLqp')

probmodel.fit(data)

#When no argument is given to posterior, return all non-replicated random varibles
print(probmodel.posterior())
```

1.8.12 Bayesian Neural Network Classifier

```
d = 10   # number of features
N = 1000 # number of observations

def neural_network(x):
  h = tf.tanh(tf.matmul(x, W_0) + b_0)
  h = tf.tanh(tf.matmul(h, W_1) + b_1)
  h = tf.matmul(h, W_2) + b_2
  return tf.reshape(h, [-1])

W_0 = Normal(0,1, shape = [d,10])
W_1 = Normal(0,1, shape = [10,10])
W_2 = Normal(0,1, shape = [10,1])

b_0 = Normal(0,1, shape = [1,10])
b_1 = Normal(0,1, shape = [1,10])
b_2 = Normal(0,1, shape = [1,1])


with inf.replicate(size = N):
    x = Normal(0,1, dim = d, observed = true)
    y = Bernoulli(logits=neural_network(x), observed = true)

#vars = 'all' automatically add all previously created random variables
probmodel = ProbModel(vars = 'all')

data = probmodel.sample(size=N)

log_prob = probmodel.log_prob(sample)

probmodel.compile(infMethod = 'KLqp')

probmodel.fit(data)

#When no argument is given to posterior, return all non-replicated random varibles
print(probmodel.posterior())
```

1.8.13 Variational Autoencoder

```
from keras.models import Sequential
from keras.layers import Dense, Activation

M = 1000
dim_z = 10
dim_x = 100

#Define the decoder network
input_z  = keras.layers.Input(input_dim = dim_z)
layer = keras.layers.Dense(256, activation = 'relu')(input_z)
output_x = keras.layers.Dense(dim_x)(layer)
decoder_nn = keras.models.Model(inputs = input, outputs = output_x)

#define the generative model
with inf.replicate(size = N)
 z = Normal(0,1, dim = dim_z)
```

(continues on next page)

(continued from previous page)

```
 x = Bernoulli(logits = decoder_nn(z.value()), observed = true)

#define the encoder network
input_x  = keras.layers.Input(input_dim = d_x)
layer = keras.layers.Dense(256, activation = 'relu')(input_x)
output_loc = keras.layers.Dense(dim_z)(layer)
output_scale = keras.layers.Dense(dim_z, activation = 'softplus')(layer)
encoder_loc = keras.models.Model(inputs = input, outputs = output_mu)
encoder_scale = keras.models.Model(inputs = input, outputs = output_scale)

#define the Q distribution
q_z = Normal(loc = encoder_loc(x.value()), scale = encoder_scale(x.value()))

#compile and fit the model with training data
probmodel.compile(infMethod = 'KLqp', Q = {z : q_z})
probmodel.fit(x_train)

#extract the hidden representation from a set of observations
hidden_encoding = probmodel.predict(x_pred, targetvar = z)
```

1.9 inferpy package

1.9.1 Subpackages

inferpy.models package

Submodules

inferpy.models.normal module

The Normal (Gaussian) distribution class.

class inferpy.models.normal.**Normal**(*loc*, *scale*, *dim=None*, *name='inf_Normal'*)

Bases: object

Class implementing the Normal distribution with location *loc*, *scale* and *dim* parameters.

The probability density of the normal distribution is,

$$f(x|\mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where

- μ is the mean or expectation of the distribution (i.e. *location*),
- σ is the standard deviation (i.e. *scale*), and
- σ^2 is the variance.

The Normal distribution is a member of the location-scale family.

This class allows the definition of a variable normal distributed of any dimension. Each of the dimensions are independent. For example:

```
import inferpy as inf

# define a 2-dimension Normal distribution of 2·3=6 batches
with inf.replicate(size=2):
        with inf.replicate(size=3):
                x = inf.models.Normal(loc=0., scale=1., dim=2)


# print its parameters
print(x.loc)
print(x.scale)


# the shape of the distribution is (6,2)
print(x.shape)

# get a sample
sample_x = x.sample([4,10])

# the shape of the sample is (4, 10, 6, 2)
print(sample_x.shape)

# get the underlying distribution Edward object
ed_x = x.dist
```

__init__ (*loc*, *scale*, *dim=None*, *name='inf_Normal'*)
Construct Normal distributions

The parameters *loc* and *scale* must be shaped in a way that supports broadcasting (e.g. *loc* + *scale* is a valid operation). If dim is specified, it should be consistent with the lengths of *loc* and *scale*

Parameters

- **loc** (*float*) – scalar or vector indicating the mean of the distribution at each dimension.
- **scale** (*float*) – scalar or vector indicating the stddev of the distribution at each dimension.
- **dim** (*int*) – optional scalar indicating the number of dimensions

Raises ValueError: if the parameters are not consistent AttributeError: if any of the properties is changed once the object is constructed

batches
Number of batches of the variable

dim
Dimensionality of variable

dist
Underlying Edward object

loc
Distribution parameter for the mean.

sample (*v*)
Method for obaining a sample of shape v

scale
Distribution parameter for standard deviation.

shape
shape of the variable, i.e. (batches, dim)

Module contents

1.9.2 Submodules

1.9.3 inferpy.replicate module

Module with the replication functionality.

class inferpy.replicate.**replicate**(*size*)
Bases: object

Class implementing the Plateau notation

The plateau notation is used to replicate the random variables contained within this construct. Every replicated variable is conditionally idependent given the previous random variables (if any) defined outside the with statement. The `with inf.replicate(size = N)` sintaxis is used to replicate N times the contained definitions. For example:

```
import inferpy as inf


with inf.replicate(size=50):
    # Define some random variables here
    print("Variable replicated " + str(inf.replicate.get_total_size()) + " times")


with inf.replicate(size=10):
    # Define some random variables here
    print("Variable replicated " + str(inf.replicate.get_total_size()) + " times")

    with inf.replicate(size=2):
        # Define some random variables here
        print("Variable replicated " + str(inf.replicate.get_total_size()) + "␣
↪times")
        print(inf.replicate.in_replicate())

# Define some random variables here
print("Variable replicated " + str(inf.replicate.get_total_size()) + " times")


print(inf.replicate.in_replicate())
```

The number of times that indicated with input argument `size`. Note that nested replicate constructs can be defined as well. At any moment, the product of all the nested replicate constructs can be obtained by invoking the static method `get_total_size()`.

Note: Defining a variable inside the construct replicate with size equal to 1, that is, `inf.replicate(size=1)` is equivalent to defining outside any replicate construct.

__init__(*size*)
Initializes the replicate construct

Parameters **size** (*int*) – number of times that the variables contained are replicated.

static get_total_size()
Static method that returns the product of the sizes of all the nested replicate constructs

Returns Integer with the product of sizes

static in_replicate()
Check if a replicate construct has been initialized

Returns True if the method is inside a construct replicate (of size different to 1). Otherwise False is return

static print_total_size()
Static that prints the total size

1.9.4 inferpy.version module

Version of inferpy

This module contains a constant string with the version of inferpy, i.e.:

```
import inferpy as inf
print(inf.VERSION)
```

1.9.5 Module contents

Python Module Index

Index

Symbols

B

D

G

I

L

N

P

R

S