InferPy Documentation

Release 1.0

Rafael Cabañas

Sep 17, 2018

Quick Start

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

Use InferPy if 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.
- 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, without having to learn how to use Tensorflow.

CHAPTER 1

Getting Started:

1.1 Installation

Install InferPy from PyPI:

```
$ pip install inferpy
```

1.2 30 seconds to InferPy

The core data structures of InferPy is a **probabilistic model**, defined as a set of **random variables** with a conditional dependency structure. A **random varible** is an object parameterized by a set of Numpy's arrays.

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

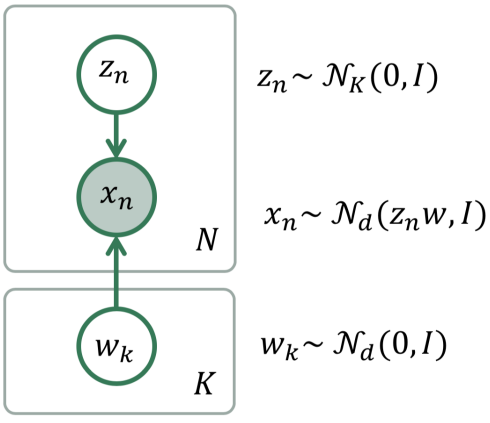


Fig. 1: Bayesian PCA

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

```
import inferpy as inf
from inferpy.models import Normal

# 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):
    w = Normal(loc = 0, scale = 1, dim = d)   # x.shape = [K,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

# define the generative model
with inf.replicate(size=N):
    z = Normal(0, 1, dim=K)   # z.shape = [N,K]
    x = Normal(inf.matmul(z,w), 1.0, observed=True, dim=d)   # x.shape = [N,d]
```

As commented above, the variables 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 variable **w** defined above.

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(varlist = [w,z,x])

# 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(varlist = [w,z,x])

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

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.

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 = pca.log_prob(data)
```

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

```
# compile and fit the model with training data
pca.compile()
pca.fit(data)

#extract the hidden representation from a set of observations
hidden_encoding = pca.posterior(z)
```

CHAPTER 2

Guiding Principles

2.1 Features

The main features of InferPy are listed below.

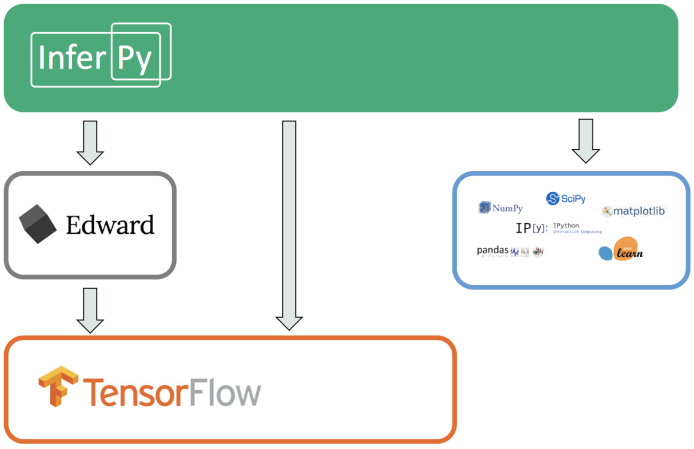
- The models that can be defined in Inferpy are those that can be defined using Edward, whose probability distribuions are mainly inherited from TensorFlow Distribuitons package.
- Edward's drawback is that for the model definition, the user has to manage complex multidimensional arrays called tensors. By contrast, in InferPy all the parameters in a model can be defined using the standard Python types (compatibility with Numpy is available as well).
- 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 seamlessly 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 computing engines.

2.2 Architecture

Given the previous considerations, we might summarize the InferPy architecture as follows.

Note that InferPy can be seen as an upper layer for working with probabilistic distributions defined over tensors. Most of the interaction is done with Edward: the definitions of the distributions, the inference. However, InferPy also interacts directly with Tensorflow in some operations that are hidden to the user, e.g. the manipulation of the tensors representing the parameters of the distributions.

An additional advantage of using Edward and Tensorflow as inference engine, is that all the paralelisation details are hidden to the user. Moreover, the same code will run either in CPUs or GPUs.



For some less important task, InferPy might also interact with other third-party software. For example, reading data is done with Pandas or the visualization tasks are leveraged to MatPlotLib.

CHAPTER 3

Guide to Building Probabilistic Models

3.1 Getting Started with Probabilistic Models

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

- A **prior model** defining a joint distribution $p(\mathbf{w})$ over the global parameters of the model. $\mathbf{w}$ can be a single random variable or a bunch of random variables with any given dependency structure.
- A **data or observation model** defining a joint conditional distribution $p(\mathbf{x}, \mathbf{z}|\mathbf{w})$ over the observed quantities $\mathbf{x}$ and the the local hidden variables $\mathbf{z}$ governing the observation $\mathbf{x}$. This data model is specified in a single-sample basis. There are many models of interest without local hidden variables, in that case, we simply specify the conditional $p(\mathbf{x}|\mathbf{w})$. Similarly, either $\mathbf{x}$ or $\mathbf{z}$ can be a single random variable or a bunch of random variables with any given dependency structure.

For example, a Bayesian PCA model has the following graphical structure,

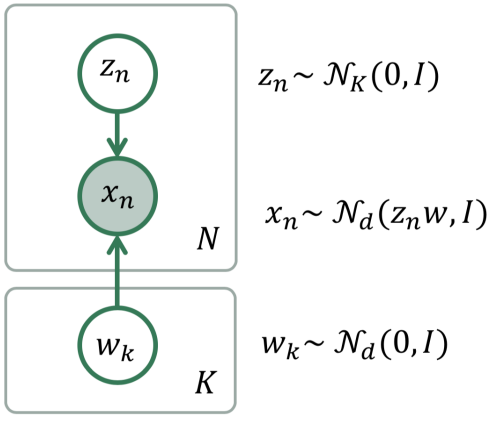


Fig. 1: Bayesian PCA

The **prior model** are the variables w_k. 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 edward as ed
import inferpy as inf
from inferpy.models import Normal

K, d, N = 5, 10, 200

# model definition
with inf.ProbModel() as m:
    #define the weights
    with inf.replicate(size=K):
        w = Normal(0, 1, dim=d)

    # define the generative model
    with inf.replicate(size=N):
        z = Normal(0, 1, dim=K)
        x = Normal(inf.matmul(z,w), 1.0, observed=True, dim=d)

m.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.

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 'dim' argument (inspired by PyMC3 and Keras):

```
import inferpy as inf
import tensorflow as tf
import numpy as np


# different ways of declaring 1 batch of 5 Normal distributions

x = inf.models.Normal(loc = 0, scale=1, dim=5)          # x.shape = [5]

x = inf.models.Normal(loc = [0, 0, 0, 0, 0], scale=1)   # x.shape = [5]

x = inf.models.Normal(loc = np.zeros(5), scale=1)       # x.shape = [5]

x = inf.models.Normal(loc = 0, scale=tf.ones(5))        # x.shape = [5]
```

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

```
with inf.replicate(size=10):
    x = inf.models.Normal(loc=0, scale=1, dim=5)        # x.shape = [10,5]
```

Following Edward's approach, the multivariate dimension is the innermost (right-most) dimension of the parameters.

Note that indexing is supported:

```
y = x[7,4]                                                    # y.shape = [1]

y2 = x[7]                                                     # y2.shape = [5]

y3 = x[7,:]                                                   # y2.shape = [5]

y4 = x[:,4]                                                   # y4.shape = [10]
```

Moreover, we may use indexation for defining new variables whose indexes may be other (discrete) variables:

```
z = inf.models.Categorical(logits = np.zeros(5))
yz = inf.models.Normal(loc=x[0,z], scale=1)              # yz.shape = [1]
```

Any random variable in InferPy contain the following (optional) input parameters in the constructor:

- `validate_args` : Python boolean indicating that possibly expensive checks with the input parameters are enabled. By default, it is set to `False`.
- `allow_nan_stats` : When `True`, the value "NaN" is used to indicate the result is undefined. Otherwise an exception is raised. Its default value is `True`.
- `name`: Python string with the name of the underlying Tensor object.
- `observed`: Python boolean which is used to indicate whether a variable is observable or not . The default value is `False`
- `dim`: dimension of the variable. The default value is `None`

Inferpy supports a wide range of probability distributions. Details of the specific arguments for each supported distributions are specified in the following sections. ‘

3.3 Probabilistic Models

A **probabilistic model** defines a joint distribution over observable and non-observable variables, $p(\mathbf{w}, \mathbf{z}, \mathbf{x})$ 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,

```
m = inf.ProbModel(varlist=[w,z,x])
m.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 `sample()`, `log_prob()` and `sum_log_prob()`. Then, we can sample data from the model and compute the log-likelihood of a data set:

```
data = m.sample(1000)
log_like = m.log_prob(data)
sum_log_like = m.sum_log_prob(data)
```

Random variables can be involved in expressive deterministic operations. Dependecies between variables are modelled by setting a given variable as a parameter of another variable. For example:

```
with inf.ProbModel() as m:
    theta = inf.models.Beta(0.5,0.5)
    z =  inf.models.Categorical(probs=[theta, 1-theta], name="z")
```

(continues on next page)

(continued from previous page)

```
m.sample()
```

Moreover, we might consider using the function `inferpy.case` as the parameter of other random variables:

```
# Categorical variable depending on another categorical variable

with inf.ProbModel() as m2:
    y =  inf.models.Categorical(probs=[0.4,0.6], name="y")
    x = inf.models.Categorical(probs=inf.case({y.equal(0): [0.0, 1.0],
                                               y.equal(1): [1.0, 0.0] }), name="x")
m2.sample()


# Categorical variable depending on a Normal distributed variable

with inf.ProbModel() as m3:
    a = inf.models.Normal(0,1, name="a")
    b = inf.models.Categorical(probs=inf.case({a>0: [0.0, 1.0],
                                               a<=0: [1.0, 0.0]}), name="b")
m3.sample()


# Normal distributed variable depending on a Categorical variable

with inf.ProbModel() as m4:
    d =  inf.models.Categorical(probs=[0.4,0.6], name="d")
    c = inf.models.Normal(loc=inf.case({d.equal(0): 0.,
                                        d.equal(1): 100.}), scale=1., name="c")
m4.sample()
```

Note that we might use the case function inside the replicate construct. The result will be a multi-batch random variable having the same distribution for each batch. When obtaining a sample from the model, each sample of a given batch in x is independent of the rest.

```
with inf.ProbModel() as m:
    y =  inf.models.Categorical(probs=[0.4,0.6], name="y")
    with inf.replicate(size=10):
        x = inf.models.Categorical(probs=inf.case({y.equal(0): [0.5, 0.5],
                                                   y.equal(1): [1.0, 0.0] }), name="x
↪")
m.sample()
```

We can also use the functions `inferpy.case_states` or `inferpy.gather` for defining the same model.

```
with inf.ProbModel() as m:
    y =  inf.models.Categorical(probs=[0.4,0.6], name="y")
    x = inf.models.Categorical(probs=inf.case_states(y, {0: [0.0, 1.0],
                                                         1: [1.0, 0.0] }), name="x")
m.sample()


with inf.ProbModel() as m:
    y =  inf.models.Categorical(probs=[0.4,0.6], name="y")
    with inf.replicate(size=10):
        x = inf.models.Categorical(probs=inf.gather([[0.5, 0.5], [1.0, 0.0]], y),␣
↪name="x")
```

(continues on next page)

(continued from previous page)

```
m.sample()
```

We can use the function `inferpy.case_states` with a list of variables (or multidimensional variables):

```
y =  inf.models.Categorical(probs=[0.5,0.5], name="y", dim=2)
p = inf.case_states(y, {(0,0): [1.0, 0.0, 0.0, 0.0], (0,1): [0.0, 1.0, 0.0, 0.0],
                        (1, 0): [0.0, 0.0, 1.0, 0.0], (1,1): [0.0, 0.0, 0.0, 1.0]} )

with inf.replicate(size=10):
    x = inf.models.Categorical(probs=p, name="x")

####

y =  inf.models.Categorical(probs=[0.5,0.5], name="y", dim=1)
z =  inf.models.Categorical(probs=[0.5,0.5], name="z", dim=1)


p = inf.case_states((y,z), {(0,0): [1.0, 0.0, 0.0, 0.0], (0,1): [0.0, 1.0, 0.0, 0.0],
                           (1, 0): [0.0, 0.0, 1.0, 0.0], (1,1): [0.0, 0.0, 0.0, 1.0]}
 ↪ )

with inf.replicate(size=10):
    x = inf.models.Categorical(probs=p, name="x")

####

p = inf.case_states([y,z], {(0,0): [1.0, 0.0, 0.0, 0.0], (0,1): [0.0, 1.0, 0.0, 0.0],
                           (1, 0): [0.0, 0.0, 1.0, 0.0], (1,1): [0.0, 0.0, 0.0, 1.0]}
 ↪ )

with inf.replicate(size=10):
    x = inf.models.Categorical(probs=p, name="x")
```

3.4 Supported Probability Distributions

Supported probability distributions are located in the package `inferpy.models`. All of them have `inferpy.models.RandomVariable` as superclass. A list with all the supported distributions can be obtained as as follows.

```
>>> inf.models.ALLOWED_VARS
['Bernoulli', 'Beta', 'Categorical', 'Deterministic', 'Dirichlet', 'Exponential',
↪'Gamma', 'InverseGamma', 'Laplace', 'Multinomial', 'Normal', 'Poisson', 'Uniform']
```

3.4.1 Bernoulli

Binary distribution which takes the value 1 with probability p and the value with $1 - p$. Its probability mass function is

$$p(x; p) = \begin{cases} p & \text{if } x = 1 \\ 1 - p & \text{if } x = 0 \end{cases}$$

An example of definition in InferPy of a random variable following a Bernoulli distribution is shown below. Note that the input parameter `probs` corresponds to p in the previous equation.

```
x = inf.models.Bernoulli(probs=0.5)

# or

x = inf.models.Bernoulli(logits=0)
```

This distribution can be initialized by indicating the logit function of the probability, i.e., $logit(p) = log(\frac{p}{1-p})$.

3.4.2 Beta

Continuous distribution defined in the interval $[0, 1]$ and parametrized by two positive shape parameters, denoted α and β.

$$p(x; \alpha, \beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}$$

where *B* is the beta function

$$B(\alpha, \beta) = \int_0^1 t^{\alpha-1}(1-t)^{\beta-1}dt$$

The definition of a random variable following a Beta distribution is done as follows.

```
x = inf.models.Beta(concentration0=0.5, concentration1=0.5)

# or simply:

x = inf.models.Beta(0.5,0.5)
```

Note that the input parameters `concentration0` and `concentration1` correspond to the shape parameters α and β respectively.

3.4.3 Categorical

Discrete probability distribution that can take k possible states or categories. The probability of each state is separately defined:

$$p(x; \mathbf{p}) = p_i$$

where $\mathbf{p} = (p_1, p_2, \ldots, p_k)$ is a k-dimensional vector with the probability associated to each possible state.

The definition of a random variable following a Categorical distribution is done as follows.

```
x = inf.models.Categorical(probs=[0.5,0.5])

# or

x = inf.models.Categorical(logits=[0,0])
```

3.4.4 Deterministic

The deterministic distribution is a probability distribution in a space (continuous or discrete) that always takes the same value k_0. Its probability density (or mass) function can be defined as follows.

$$p(x; k_0) = \begin{cases} 1 & \text{if } x = k_0 \\ 0 & \text{if } x \neq k_0 \end{cases}$$

The definition of a random variable following a Beta distribution is done as follows:

```
x = inf.models.Deterministic(loc=5)
```

where the input parameter `loc` corresponds to the value k_0.

3.4.5 Dirichlet

Dirichlet distribution is a continuous multivariate probability distribution parmeterized by a vector of positive reals $(\alpha_1, \alpha_2, \ldots, \alpha_k)$. It is a multivariate generalization of the beta distribution. Dirichlet distributions are commonly used as prior distributions in Bayesian statistics. The Dirichlet distribution of order $k \geq 2$ has the following density function.

$$p(x_1, x_2, \ldots x_k; \alpha_1, \alpha_2, \ldots, \alpha_k) = \frac{\Gamma\left(\sum_i \alpha_i\right)}{\prod_i \Gamma(\alpha_i)} \prod_{i=1}^{k} x_i^{\alpha_i - 1}$$

The definition of a random variable following a Beta distribution is done as follows:

```
x = inf.models.Dirichlet(concentration=[5,1])

# or simply:

x = inf.models.Dirichlet([5,1])
```

where the input parameter `concentration` is the vector $(\alpha_1, \alpha_2, \ldots, \alpha_k)$.

3.4.6 Exponential

The exponential distribution (also known as negative exponential distribution) is defined over a continuous domain and describes the time between events in a Poisson point process, i.e., a process in which events occur continuously and independently at a constant average rate. Its probability density function is

$$p(x; \lambda) = \begin{cases} \lambda e^{-\lambda x} & \text{if } x \geq 0 \\ 0 & \text{if } x < k_0 \end{cases}$$

where $\lambda > 0$ is the rate or inverse scale.

The definition of a random variable following a exponential distribution is done as follows:

```
x = inf.models.Exponential(rate=1)

# or simply

x = inf.models.Exponential(1)
```

where the input parameter `rate` corresponds to the value λ.

3.4.7 Gamma

The Gamma distribution is a continuous probability distribution parametrized by a concentration (or shape) parameter $\alpha > 0$, and an inverse scale parameter $\lambda > 0$ called rate. Its density function is defined as follows.

$$p(x; \alpha, \beta) = \frac{\beta^\alpha x^{\alpha-1} e^{\beta x}}{\Gamma(\alpha)}$$

for $x > 0$ and where $\Gamma(\alpha)$ is the gamma function.

The definition of a random variable following a gamma distribution is done as follows:

```
x = inf.models.Gamma(concentration=3, rate=2)
```

where the input parameters `concentration` and `rate` corespond to α and β respectively.

3.4.8 Inverse-gamma

The Inverse-gamma distribution is a continuous probability distribution which is the distribution of the reciprocal of a variable distributed according to the gamma distribution. It is also parametrized by a concentration (or shape) parameter $\alpha > 0$, and an inverse scale parameter $\lambda > 0$ called rate. Its density function is defined as follows.

$$p(x; \alpha, \beta) = \frac{\beta^\alpha x^{-\alpha-1} e^{-\frac{\beta}{x}}}{\Gamma(\alpha)}$$

for $x > 0$ and where $\Gamma(\alpha)$ is the gamma function.

The definition of a random variable following a inverse-gamma distribution is done as follows:

```
x = inf.models.InverseGamma(concentration=3, rate=2)
```

where the input parameters `concentration` and `rate` corespond to α and β respectively.

3.4.9 Laplace

The Laplace distribution is a continuous probability distribution with the following density function

$$p(x; \mu, \sigma) = \frac{1}{2\sigma} exp\left(-\frac{|x-\mu|}{\sigma}\right)$$

The definition of a random variable following a Beta distribution is done as follows:

```
x = inf.models.Laplace(loc=0, scale=1)

# or simply

x = inf.models.Laplace(0,1)
```

where the input parameter `loc` and `scale` correspond to μ and σ respectively.

3.4.10 Multinomial

The multinomial is a discrete distribution which models the probability of counts resulting from repeating n times an experiment with k possible outcomes. Its probability mass function is defined below.

$$p(x_1, x_2, \ldots x_k; \mathbf{p}) = \frac{n!}{\prod_{i=1}^k x_i} \prod_{i=1}^k p_i^{x^i}$$

where $\mathbf{p}$ is a k-dimensional vector defined as $\mathbf{p} = (p_1, p_2, \ldots, p_k)$ with the probability associated to each possible outcome.

The definition of a random variable following a multinomial distribution is done as follows:

```
x = inf.models.Multinomial(total_count=4, probs=[0.5,0.5])

# or

x = inf.models.Multinomial(total_count=4, logits=[0,0])
```

3.4.11 Multivariate-Normal

A multivariate-normal (or Gaussian) defines a set of normal-distributed variables which are assumed to be idependent. In other words, the covariance matrix is diagonal.

A single multivariate-normal distribution defined on $\mathbb{R}^2$ can be defined as follows.

```
x = inf.models.MultivariateNormalDiag(
    loc=[1., -1],
    scale_diag=[1, 2.]
)
```

3.4.12 Normal

The normal (or Gaussian) distribution is a continuous probability distribution with the following density function

$$p(x; \mu, \sigma) = \frac{1}{2\sigma} exp\left(-\frac{|x-\mu|}{\sigma}\right)$$

where μ is the mean or expectation of the distribution, σ is the standard deviation, and σ^2 is the variance.

A normal distribution can be defined as follows.

```
x = inf.models.Normal(loc=0, scale=1)

# or

x = inf.models.Normal(0,1)
```

where the input parameter `loc` and `scale` correspond to μ and σ respectively.

3.4.13 Poisson

The Poisson distribution is a discrete probability distribution for modeling the number of times an event occurs in an interval of time or space. Its probability mass function is

$$p(x; \lambda) = e^{-\lambda} \frac{\lambda^x}{x!}$$

where λ is the rate or number of events per interval.

A Poisson distribution can be defined as follows.

```
x = inf.models.Poisson(rate=4)

# or

x = inf.models.Poisson(4)
```

3.4.14 Uniform

The continuous uniform distribution or rectangular distribution assings the same probability to any x in the interval $[a, b]$.

$$p(x; a, b) = \begin{cases} \frac{1}{b-a} & \text{if } x \in [a, b] \\ 0 & \text{if } x \notin [a, b] \end{cases}$$

A uniform distribution can be defined as follows.

```
x = inf.models.Uniform(low=1, high=3)

# or

inf.models.Uniform(1,3)
```

where the input parameters `low` and `high` correspond to the lower and upper bounds of the interval $[a, b]$.

CHAPTER 4

Guide to Approximate Inference

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,

```
m.compile(infMethod="KLqp")
m.fit(x_train)
m.posterior(z)
```

The compilation process allows to choose the inference algorithm through the `infMethod` argument. In the above example we use `'Klqp'`.

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

```
qw = inf.Qmodel.Normal(w)
qz = inf.Qmodel.Normal(z)

qmodel = inf.Qmodel([qw, qz])
```

(continues on next page)

(continued from previous page)

```
m.compile(infMethod="KLqp", Q=qmodel)
m.fit(x_train)
m.posterior(z)
```

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.

4.2 Compositional Inference

Note: not implemented yet

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.

4.3 Supported Inference Methods

CHAPTER 5

Guide to Model Validation

Note: not implemented yet

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])
```

CHAPTER 6

Guide to Data Handling

InferPy leverages existing Pandas functionality for reading data. As a consequence, InferPy can learn from datasets in any file format handled by Pandas. This is possible because the method `inferpy.ProbModel.fit(data)` accepts as input argument a Pandas DataFrame.

In the following code fragment, an example of learning a model from a CVS file is shown:

```
import inferpy as inf
import pandas as pd


data = pd.read_csv("inferpy/datasets/test.csv")
N = len(data)

with inf.ProbModel() as m:

    thetaX = inf.models.Normal(loc=0., scale=1.)
    thetaY = inf.models.Normal(loc=0., scale=1.)


    with inf.replicate(size=N):
        x = inf.models.Normal(loc=thetaX, scale=1., observed=True, name="x")
        y = inf.models.Normal(loc=thetaY, scale=1., observed=True, name="y")


m.compile()
m.fit(data)

m.posterior([thetaX, thetaY])
```

CHAPTER 7

Probabilistic Model Zoo

7.1 Bayesian Linear Regression

```
import edward as ed
import inferpy as inf
from inferpy.models import Normal
import numpy as np

d, N =  5, 20000

# model definition
with inf.ProbModel() as m:

    #define the weights
    w0 = Normal(0,1)
    w = Normal(0, 1, dim=d)

    # define the generative model
    with inf.replicate(size=N):
        x = Normal(0, 1, observed=True, dim=d)
        y = Normal(w0 + inf.dot(x,w), 1.0, observed=True)


# toy data generation
x_train = inf.models.Normal(loc=10, scale=5, dim=d).sample(N)
y_train = np.matmul(x_train, np.array([10,10,0.1,0.5,2]).reshape((d,1))) \
          + inf.models.Normal(loc=0, scale=5, dim=1).sample(N)


data = {x.name: x_train, y.name: y_train}


# compile and fit the model with training data
m.compile()
```

(continues on next page)

(continued from previous page)

```
m.fit(data)

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

7.2 Bayesian Logistic Regression

```
import edward as ed
import inferpy as inf
from inferpy.models import Normal, Bernoulli, Categorical

d, N =  10, 500

# model definition
with inf.ProbModel() as m:

    #define the weights
    w0 = Normal(0,1)
    w = Normal(0, 1, dim=d)

    # define the generative model
    with inf.replicate(size=N):
        x = Normal(0, 1, observed=True, dim=d)
        y = Bernoulli(logits=w0+inf.dot(x, w), observed=True)


# toy data generation
x_train = Normal(loc=0, scale=1, dim=d).sample(N)
y_train = Bernoulli(probs=0.4).sample(N)
data = {x.name: x_train, y.name: y_train}

# compile and fit the model with training data
m.compile()
m.fit(data)

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

7.3 Bayesian Multinomial Logistic Regression

```
import edward as ed
import inferpy as inf
from inferpy.models import Normal, Bernoulli, Categorical
import numpy as np
```

(continues on next page)

(continued from previous page)

```
d, N =  10, 500

#number of classes
K = 3

# model definition
with inf.ProbModel() as m:

    #define the weights
    w0 = Normal(0,1, dim=K)

    with inf.replicate(size=d):
        w = Normal(0, 1, dim=K)

    # define the generative model
    with inf.replicate(size=N):
        x = Normal(0, 1, observed=True, dim=d)
        p = w0 + inf.matmul(x, w)
        y = Bernoulli(logits = p, observed=True)


y.shape

# toy data generation
x_train = Normal(loc=0, scale=1, dim=d).sample(N)
y_train = Bernoulli(probs=np.random.rand(K)).sample(N)
data = {x.name: x_train, y.name: y_train}


# compile and fit the model with training data
m.compile()
m.fit(data)

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

7.4 Mixture of Gaussians

```
import edward as ed
import inferpy as inf
import numpy as np
import tensorflow as tf



K, d, N, T = 3, 4, 1000, 5000


# toy data generation
x_train = np.vstack([inf.models.Normal(loc=0, scale=1, dim=d).sample(300),
```

(continues on next page)

(continued from previous page)

```
                    inf.models.Normal(loc=10, scale=1, dim=d).sample(700)])

######## Inferpy ##########


# model definition
with inf.ProbModel() as m:

    # prior distributions
    with inf.replicate(size=K):
        mu = inf.models.Normal(loc=0, scale=1, dim=d)
        sigma = inf.models.InverseGamma(concentration=1, rate=1, dim=d,)
    p = inf.models.Dirichlet(np.ones(K)/K)

    # define the generative model
    with inf.replicate(size=N):
        z = inf.models.Categorical(probs = p)
        x = inf.models.Normal(mu[z], sigma[z],observed=True, dim=d)

# compile and fit the model with training data
data = {x: x_train}
m.compile(infMethod="MCMC")
m.fit(data)

# print the posterior
print(m.posterior(mu))
```

7.5 Linear Factor Model (PCA)

```
import edward as ed
import inferpy as inf

K, d, N = 5, 10, 200

# model definition
with inf.ProbModel() as m:
    #define the weights
    with inf.replicate(size=K):
        w = inf.models.Normal(0, 1, dim=d)

    # define the generative model
    with inf.replicate(size=N):
        z = inf.models.Normal(0, 1, dim=K)
        x = inf.models.Normal(inf.matmul(z,w),
                              1.0, observed=True, dim=d)

# toy data generation
x_train = inf.models.Normal(loc=0, scale=1., dim=d).sample(N)
data = {x.name: x_train}
```

(continues on next page)

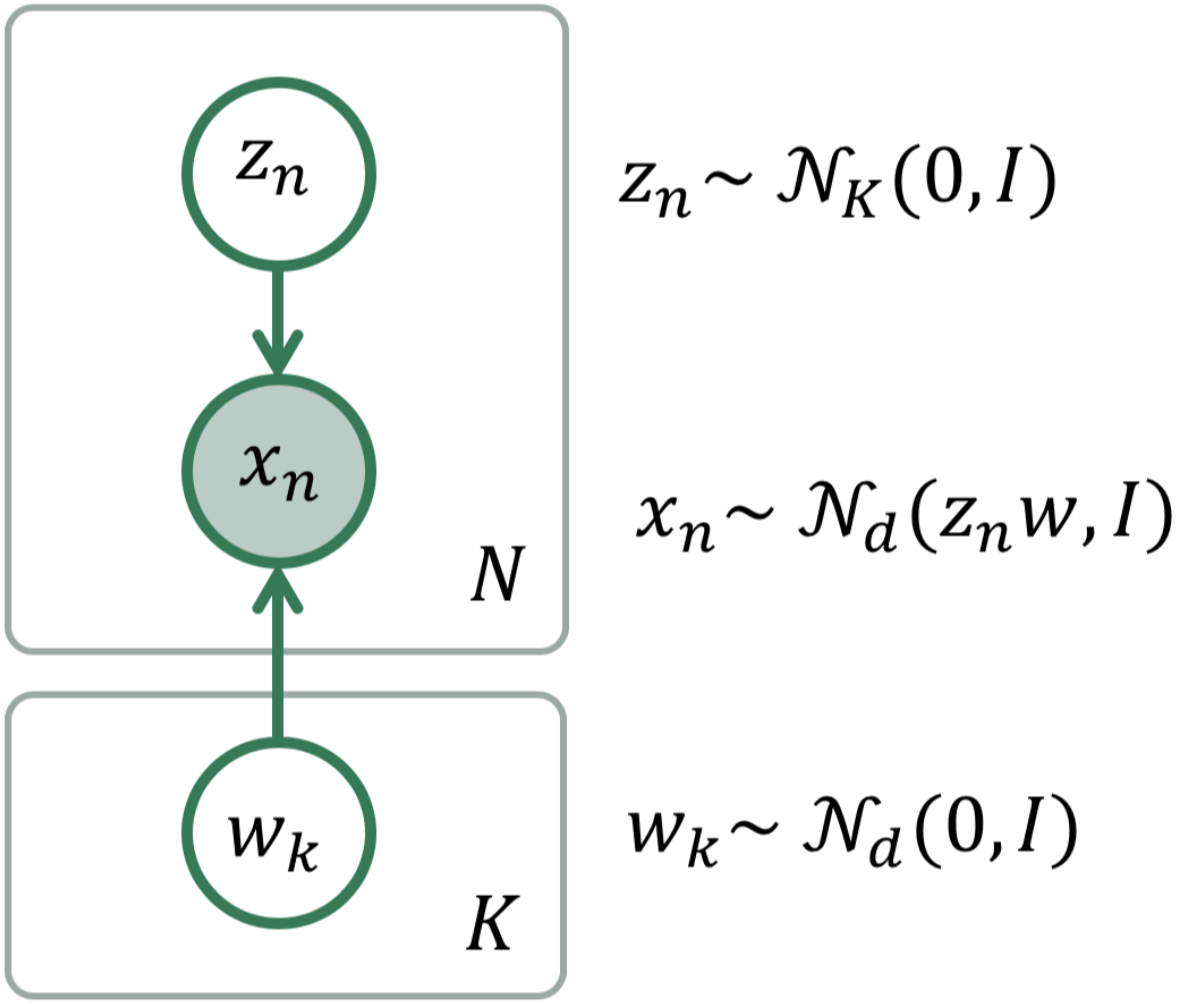


Fig. 1: Linear Factor Model

(continued from previous page)

```
# compile and fit the model with training data
m.compile()
m.fit(data)

#extract the hidden representation from a set of observations
hidden_encoding = m.posterior(z)


inf.dot(z, w)
```

7.6 PCA with ARD Prior (PCA)

```
import edward as ed
import inferpy as inf
from inferpy.models import Normal, InverseGamma

K, d, N = 5, 10, 200

# model definition
with inf.ProbModel() as m:
    #define the weights
    with inf.replicate(size=K):
        w = Normal(0, 1, dim=d)

    sigma = InverseGamma(1.0,1.0)

    # define the generative model
    with inf.replicate(size=N):
        z = Normal(0, 1, dim=K)
        x = Normal(inf.matmul(z,w),
                   sigma, observed=True, dim=d)

# toy data generation
x_train = Normal(loc=0, scale=1., dim=d).sample(N)
data = {x.name: x_train}


# compile and fit the model with training data
m.compile()
m.fit(data)

#extract the hidden representation from a set of observations
hidden_encoding = m.posterior(z)
```

7.7 Matrix Factorization

```
import inferpy as inf
from inferpy.models import Normal

N=200
M=50
K=5

# Shape [M,K]
with inf.replicate(size=K):
    gamma = Normal(0,1, dim=M)

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

# x_mn has shape [N,K] x [K,M] = [N,M]

with inf.replicate(size=N):
    x = Normal(inf.matmul(w,gamma), 1, observed = True)

m = inf.ProbModel([w,gamma,x])

data = m.sample(size=N)

log_prob = m.log_prob(data)

m.compile(infMethod = 'KLqp')

m.fit(data)

print(m.posterior([w,gamma]))
```

CHAPTER 8

Inferpy vs Edward

8.1 Bayesian Linear Regression

```
import edward as ed
import inferpy as inf
import numpy as np
import tensorflow as tf

d, N =  5, 20000



# toy data generation
x_train = inf.models.Normal(loc=10, scale=5, dim=d).sample(N)
y_train = np.matmul(x_train, np.array([10,10,0.1,0.5,2]).reshape((d,1))) \
          + inf.models.Normal(loc=0, scale=5, dim=1).sample(N)



### InferPy #######

# model definition
with inf.ProbModel() as m:

    # define the weights
    w0 = inf.models.Normal(0,1)
    w = inf.models.Normal(0, 1, dim=d)

    # define the generative model
    with inf.replicate(size=N):
        x = inf.models.Normal(0, 1, observed=True, dim=d)
        y = inf.models.Normal(w0 + inf.dot(x,w), 1.0, observed=True)
```

(continues on next page)

(continued from previous page)

```
# compile and fit the model with training data
m.compile()
data = {x: x_train, y: y_train}
m.fit(data)

# print the posterior distributions
print(m.posterior([w, w0]))



### Edward #####

# define the weights
w0 = ed.models.Normal(loc=tf.zeros(1), scale=tf.ones(1))
w = ed.models.Normal(loc=tf.zeros(d), scale=tf.ones(d))

# define the generative model
x = ed.models.Normal(loc=tf.zeros([N,d]), scale=tf.ones([N,d]))
y = ed.models.Normal(loc=ed.dot(x, w) + w0, scale=tf.ones(N))

# compile and fit the model with training data
qw = ed.models.Normal(loc=tf.Variable(tf.random_normal([d])),
                      scale=tf.nn.softplus(tf.Variable(tf.random_normal([d]))))
qw0 = ed.models.Normal(loc=tf.Variable(tf.random_normal([1])),
                       scale=tf.nn.softplus(tf.Variable(tf.random_normal([1]))))

inference = ed.KLqp({w: qw, w0: qw0}, data={x: x_train, y: y_train.reshape(N)})
inference.run()

# print the posterior distributions
print([qw.loc.eval(), qw0.loc.eval()])
```

8.2 Gaussian Mixture

```
import edward as ed
import inferpy as inf
import numpy as np
import tensorflow as tf



K, d, N, T = 3, 4, 1000, 5000


# toy data generation
x_train = np.vstack([inf.models.Normal(loc=0, scale=1, dim=d).sample(300),
                     inf.models.Normal(loc=10, scale=1, dim=d).sample(700)])

######## Inferpy ##########


# model definition
with inf.ProbModel() as m:

```

(continues on next page)

(continued from previous page)

```
    # prior distributions
    with inf.replicate(size=K):
        mu = inf.models.Normal(loc=0, scale=1,
                               dim=d)
        sigma = inf.models.InverseGamma(
            concentration=1, rate=1, dim=d,)
    p = inf.models.Dirichlet(np.ones(K)/K)

    # define the generative model
    with inf.replicate(size=N):
        z = inf.models.Categorical(probs = p)
        x = inf.models.Normal(mu[z], sigma[z],
                              observed=True,
                              dim=d)
# compile and fit the model with training data
data = {x: x_train}
m.compile(infMethod="MCMC")
m.fit(data)

# print the posterior
print(m.posterior(mu))



######## Edward ##########



# model definition

# prior distributions
p = ed.models.Dirichlet(concentration=tf.ones(K)/K)
mu = ed.models.Normal(0.0, 1.0, sample_shape=[K, d])
sigma = ed.models.InverseGamma(concentration=1.0,
                               rate=1.0,
                               sample_shape=[K, d])
# define the generative model
z = ed.models.Categorical(logits=tf.log(p) -
                                 tf.log(1.0 - p),
                          sample_shape=N)
x = ed.models.Normal(loc=tf.gather(mu, z),
                     scale=tf.gather(sigma, z))

# compile and fit the model with training data
qp = ed.models.Empirical(params=tf.get_variable(
    "qp/params",
    [T, K],
    initializer=tf.constant_initializer(1.0 / K)))
qmu = ed.models.Empirical(
    params=
    tf.get_variable("qmu/params",
                    [T, K, d],
                    initializer=
                    tf.zeros_initializer()))
qsigma = ed.models.Empirical(
    params=
    tf.get_variable("qsigma/params",
```

(continues on next page)

(continued from previous page)

```
                    [T, K, d],
                    initializer=
                    tf.ones_initializer()))
qz = ed.models.Empirical(
    params=
    tf.get_variable("qz/params",
                    [T, N],
                    initializer=
                    tf.zeros_initializer(),
                    dtype=tf.int32))

gp = ed.models.Dirichlet(concentration=tf.ones(K))
gmu = ed.models.Normal(loc=tf.ones([K,d]),
             scale=tf.ones([K,d]))
gsigma = ed.models.InverseGamma(concentration=
                                tf.ones([K,d]),
                      rate=tf.ones([K,d]))
gz = ed.models.Categorical(logits=tf.zeros([N, K]))

inference = ed.MetropolisHastings(
    latent_vars={p: qp, mu: qmu,
                 sigma: qsigma, z: qz},
    proposal_vars={p: gp, mu: gmu,
                   sigma: gsigma, z: gz},
    data={x: x_train})

inference.run()

# print the posterior
print(qmu.params.eval())
```

CHAPTER 9

inferpy package

9.1 Subpackages

9.1.1 inferpy.inferences package

Module contents

9.1.2 inferpy.models package

Submodules

inferpy.models.normal module

inferpy.models.random_variable module

Module contents

9.1.3 inferpy.util package

Submodules

inferpy.util.error module

inferpy.util.ops module

inferpy.util.runtime module

inferpy.util.wrappers module

Module contents

9.2 Submodules

9.3 inferpy.prob_model module