Bringing together visual analytics and probabilistic programming languages

Jonas Aaron Gütter
Friedrich Schiller Universität Jena
register number 152127
Supervisor: Dipl. Inf. Philipp Lucas
First reviewer: Prof. Dr. Joachim Giesen
Second reviewer: Dr. Sören Laue

Friday 14th April, 2023

Abstract

A probabilistic programming language (PPL) provides methods to represent a probabilistic model by using the full power of a general purpose programming language. Thereby it is possible to specify complex models with a comparatively low amount of code. With Uber, Microsoft and DARPA focusing research efforts towards this area, PPLs are likely to play an important role in science and industry in the near future. However, in most cases, models built by PPLs lack appropriate ways to be properly visualized, although visualization is an important first step in detecting errors and assessing the overall fitness of a model. The thesis at hand aims to improve this situation by providing an interface between a popular PPL named PyMC3 and the software Lumen which provides several visualization methods for statistical models. The thesis shows how arbitrary models built in PyMC3 can be visualized with Lumen using the interface. It becomes clear that even for very simple cases, visualization can contribute an important part in understanding and validating the model since Bayesian models often behave unexpectedly. Lumen can therefore act as a useful tool for model checking.

Affidavit

I hereby confirm that my thesis entitled "Bringing together visual analysis and probabilistic programming languages" is the result of my own work. I did not receive any help or support from commercial consultants. All sources and/or materials applied are listed and specified in the thesis.
Furthermore, I confirm that this thesis has not yet been submitted as part of another examination process neither in identical nor in similar form.

Place, Date	Signature

Contents

1 Introduction

A probabilistic programming language (PPL) provides methods to represent a probabilistic model by using the full power of a general purpose programming language. Thereby it is possible to specify complex models with a comparatively low amount of code. With Uber [1], Microsoft [2] and DARPA [3] focusing research efforts towards this area, PPLs are likely to play an important role in science and industry in the near future. However, in most cases, models built by PPLs lack appropriate ways to be properly visualized, although visualization is an important first step in detecting errors and assessing the overall fitness of a model. This could be resolved by the software Lumen which provides several visualization methods for statistical models. Previous to this thesis, PPLs were not yet supported by Lumen. The goal of the master thesis at hand is to change that by implementing an interface between Lumen and PyMC3, a PPL wrapped in a python package, so that exploring PPL models by visual analytics becomes possible. The report starts with explaining the basic concepts of Bayesian statistics, along with the most popular sampling methods used to draw inference from Bayesian models. Following that, the idea of probabilistic programming is introduced, as well as the PyMC3 package as a concrete application and the Lumen software as a means of visualizing the resulting models. Having covered the theoretical aspects, the focus is then laid on practical implementations: The same Bayesian model is built once with and once without PyMC3 and the results are compared to illustrate the functionality of Bayesian modeling and of PyMC3 in particular. After that, the core task of the thesis, the implementation of the interface, is described. For this purpose, operations like model fitting, model marginalization and model conditioning have to be implemented using PyMC3. Also a kernel density estimator is used to generate a probability density function out of samples. Finally, a number of examples are shown to illustrate the usefulness of the visualization, and the thesis is concluded.

2 Concepts of Bayesian Statistics

In this chapter, the theoretical concepts of building statistical models using a Bayesian approach are explained, as well as the methods to draw inference from a model by sampling. The former requires an understanding of basic calculation rules for working with conditional probabilities. These rules are further explained in Appendix A.

2.1 General principle

Bayesian data analysis can be used for parameter estimation and prediction of new data points. Before doing Bayesian data analysis one has to come up with a model whose parameters can then be estimated by Bayesian methods and that can then be used for prediction. In addition to specifying the model, it is necessary to specify prior distributions on the model parameters. This is a major difference to the classical frequentist approach where model parameters are just single points. In Bayesian data analysis model parameters are treated as random variables that follow a probability distribution. So the first step is usually to define these so-called prior distributions. Finding a good prior can be a complex task. A prior should represent the knowledge that the researcher has about the parameter previous to seeing any data.

Parameter estimation Once the model as well as the priors are specified, parameter estimation can be performed by applying the Bayes rule to calculate posterior distributions on the parameters, that is, probability distributions on the parameters given the observed data. The Bayes rule allows to invert the order of a conditional probability and goes as follows:

$$p(A|B) = p(B|A) * p(A)/p(B) \tag{1}$$

Transferring that to a model with data X and parameters θ, according to [4], we get:

$$p(\theta|X) = p(X|\theta) * p(\theta)/p(X) \tag{2}$$

Here, $p(\theta|X)$ is the posterior distribution of the parameters. Getting this distribution is equivalent to getting the parameter estimates in the classical frequentist approach. $p(X|\theta)$ is the distribution of the data given the parameters. This is what is specified in the model structure that has to be generated before doing any Bayesian analysis. $p(\theta)$ is the distribution of the parameters without seeing any data. It is the prior distribution of the parameters that has already been set up in the first step. $p(X)$ is the distribution over the data without seeing any parameters. It can be calculated by integrating over the joint distribution of the data and the parameters, but that integral can become difficult to solve with increasing model complexity [4]. However, since the data is known in advance and will not change during the estimation process, it can be treated as a constant and therefore ignored for many applications.

Prediction Prediction means giving a statement about how future data might look that was generated by the same mechanism on which the model was learned. To predict new data points it is necessary to get the posterior distribution of the data, that is, the probability distribution over the data variables x given the observed data X: $P(x|X)$. New data points can then be generated by sampling from this distribution. It can be calculated by integrating over the joint posterior distribution [4]:

$$P(x|X) = \int p(x, \theta|X)d\theta \tag{3}$$

This requires the joint posterior distribution to be known: the probability distribution over the data as well as the parameters given the observed data $p(x, \theta|X)$. The joint posterior can be computed as follows:

$$\begin{aligned} p(x, \theta|X) &= p(x|\theta, X) * p(\theta|X) \\ &= p(x|\theta) * p(\theta|X) \end{aligned} \tag{4}$$

The simplification in the second step is possible since the future data x depends on the observed data X only indirectly via the parameters θ: If θ is known, X has no influence on x anymore. Hence, we can write $p(x|\theta, X) = p(x|\theta)$.

2.2 Inference through sampling

As stated in Equation 2, using the Bayes formula, we get the posterior distribution the following way:

$$p(\theta|X) = p(X|\theta) * p(\theta)/p(X) \tag{5}$$

We get both the likelihood $p(X|\theta)$ and the prior $p(\theta)$ from the model assumptions. The marginalized probability over the data, $p(X)$, however, is more complicated. To compute it analytically we would have to solve the following integral:

$$\begin{aligned} p(X) &= \int p(\theta, X)d\theta \\ &= \int p(X|\theta) * p(\theta)d\theta \end{aligned} \tag{6}$$

Calculating this integral can be very difficult or impossible since the integral can be multidimensional [4]. So it is not always possible to get the posterior distribution of the parameters analytically. Therefore, methods to sample from the posterior distribution of the parameters have been developed. These methods are called Markov Chain Monte Carlo methods (MCMC) or Markov chain simulations. A Markov chain is a sequence of events where the probability for each event only depends on the preceding event [5]. MCMC methods all have in common that parameter values are drawn repeatedly as some form of such a Markov chain: In each iteration, the parameter values θ^t depend only on the values of the preceding iteration θ^{t-1}. So in each step t, values are drawn from a transition distribution $T_t(\theta^t|\theta^{t-1})$. This distribution has to be constructed in a way that it converges to the posterior distribution $p(\theta|X)$. Once it has converged, the parameter draws can be used as samples. Some of the more popular MCMC methods are explained below.

Gibbs Sampling Gibbs Sampling assumes that we don't know the joint posterior distribution $p(\theta = \theta_0, ..., \theta_n)$, but we do know the conditional distribution $p(\theta_i|\theta_0, ..., \theta_{i-1}, \theta_{i+1}, ..., \theta n)$ for each quantity θ_i. At first, arbitrary starting points are set for θ, then each θ_i is drawn from its conditional distribution given the latest values of the other quantities. This is repeated until convergence is achieved [6].

Metropolis The metropolis algorithm can be seen as a generalization of the Gibbs sampler [4] and was originally developed to calculate the behavior of interacting molecules without having to compute the multidimensional integrals previously needed for that computation[7]. The algorithm goes as follows :

1. An arbitrary starting location is chosen for each of the molecules.
2. For each molecule, a new possible location is proposed. How this new location is calculated is itself a complex task and can be crucial for the performance of the algorithm. For the sake of simplicity we assume here that the proposed location is just randomly chosen in the near vicinity of the old location.
3. The change of energy in the system introduced by the position change is calculated.
4. If the proposed location has a lower energy level than the current one, the proposed location is accepted. If it has a higher energy level, it is accepted with a probability anti proportional to the energy increase, that is, the higher the energy increase, the lower the chance of accepting the proposed position. After that, the process is repeated up from step 2.

This algorithm is equivalent to sampling from a probability distribution. The molecules represent variables of the probability distribution, their location represents a specific value of the variable and the negative energy of a configuration of molecules represents the probability density of a point in the distribution. The movement of the molecules through space is the Markov chain here. After the algorithm has converged, the distribution of the chosen points resembles the true probability distribution. It is important to note that for this algorithm to work it is necessary that the probability density, apart from a constant factor, can be computed for any given point of the probability distribution since it is needed for evaluating the acceptance of the proposal [4].

Hamiltonian Monte-Carlo The base approach of Hamiltonian Monte Carlo (HMC) is that we assume a particle in a multidimensional space where each dimension represents a variable of the target distribution and thus the location θ of the particle represents a specific point on the target distribution. As in the metropolis algorithm, a starting location is arbitrarily chosen. The difference to the metropolis algorithm now lies in the generation of the new locations. An auxiliary variable r and the parameters L and ϵ are introduced for this purpose. r represents the momentum of the particle, L is the number of steps to get to one sample and ϵ is the step size for one of these steps. For generating one sample point, a change of locations is proposed L times and is accepted or rejected according to the metropolis algorithm. In each of the L steps, θ and r are updated by using the so-called leapfrog integrator, which is shown in Equation 7.

$$\begin{aligned} r &\leftarrow r + (\epsilon/2)\, \nabla_\theta \,(log\, p(\theta)) \\ \theta &\leftarrow \theta + \epsilon r \\ r &\leftarrow r + (\epsilon/2)\, \nabla_\theta \,(log\, p(\theta)) \end{aligned} \tag{7}$$

As one can see, the update process requires gradient information on the log probability density of the target distribution. So this method cannot be applied if gradient information is unavailable. The generated locations here show less random walk behavior and therefore better performance than the algorithms above if the parameters are set properly [8].

NUTS The No-U-Turn-Sampler (NUTS) is a self-tuning variant of Hamiltonian Monte Carlo where the number of steps per sample L does not need to be specified at a fixed value anymore. Instead, it is attempted to stop the process once the maximum distance between initial and proposed point is reached. This maximum is reached when the inner product between r and the distance between initial and proposed point becomes negative. The next iteration would then only decrease the distance [4]. Once the stop is reached, a sample is drawn from all the points that were visited [8].

2.3 Evaluating models

Since a model almost never catches all aspects of reality it is necessary, after a Bayesian model has been found, to carry out model checks. Traditionally, model checking has been performed by calculating test quantities or test statistics. A test quantity is usually a scalar summary of data and parameters. A test statistic is also a summary of data, but is conditioned on a fixed set of parameters. It can be used in Bayesian as well as in non-Bayesian contexts. Those summaries are then used to compute a tail-area-probability, also called p-value. In the case of a test statistic, the p-value gives the probability that, given a parameter θ, a test statistic T for replicated data X^{rep} is greater or equal the test statistic for the observed data X, as shown in Equation 8:

$$p_C = Pr(T(X^{rep}) >= T(X)|\theta) \tag{8}$$

[4]
The p-value for a test quantity, on the other hand, is not conditional on a fixed θ since in Bayesian statistics parameters are drawn from a distribution the same way as outcome variables. Instead, the observed data X is the fixed quantity for the Bayesian p-value, as shown in Equation 9

$$p_B = Pr(T(X^{rep}, \theta) >= T(X, \theta)|X) \tag{9}$$

[4]
If a test quantity has a p-value close to 0 or 1, it means that the test quantity for the simulated data is nearly always larger (when it is close to 1) or smaller (close to 0) than the test quantity for the observed data and therefore the aspect that is analyzed by the test quantity is not well captured in the model.
Gelman [9] argues that visualization can be seen as an alternative form of model checking. He even claims that "when more complex models are being used, graphical checks are more necessary than ever to detect areas of model misfit" [9]. In graphical model checking, the visualization plays the role of the test quantity. Even though test quantities are usually scalars, one can also compute them as vectors so that they can be displayed graphically [9]. When we take this one step further, in my opinion, the data and the parameters can be seen as a vector summary of themselves, so by displaying them, as it is done in this thesis later, we are performing a regular model check according to the proceedings described above.

3 Probabilistic Programming Languages

In this section, the idea of Probabilistic Programming is explained. It is defined what a Probabilistic Programming Language(PPL) is and a number of PPLs are introduced

3.1 Probabilistic Programming

According to Pfeffer [10], probabilistic programming is the process of creating a probabilistic reasoning system with the means of a programming language. A probabilistic reasoning system is, in turn, a structure that uses knowledge and logic to calculate a probability. Here, knowledge can be interpreted as data about certain quantities and logic can be interpreted as the knowledge about how these quantities interact and influence each other. Pfeffer [10] uses the example shown in Figure 1 to illustrate the functionality of a probabilistic reasoning system. There, we want to calculate the probability of scoring a goal by a corner kick during a game of soccer. The probabilistic reasoning system includes general knowledge about the situation, for example, that overall 9% of corner kicks result in a goal. That is what is called the korner-kick model in the example. Additional data about the circumstances of this particular situation are given to the system as evidence. In the example, these circumstances are a tall center forward, an inexperienced goalie and strong winds. The system now uses this data together with its logic of how each circumstance affects the outcome to calculate probabilities for a given event, here, the scoring of a goal. The part of computing these probabilities is called inference algorithm in the example. From a data science perspective, this means that a Probabilistic Programming Language(PPL) provides ways of describing a probability model and drawing inference from it which, in the ideal case, are more flexible, more efficient and better to understand than traditional approaches [3]. Currently, a number of PPLs is publicly available, with the most popular being mentioned below:

Stan is an open-source program written in C++ that is designed for Bayesian inference on user-specified models. As described in [11], it consists of the following components:

- A modeling language that allows users to specify custom models
- An inference engine which uses Hamilton Monte Carlo methods for sampling to get an approximation for the posterior distribution
- An L-BFGS optimizer for finding local optima
- Procedures for automatic differentiation which are able to compute gradients required by the sampler and the optimizer
- Routines to monitor the convergence of parallel chains and compute inferences and effective sample sizes

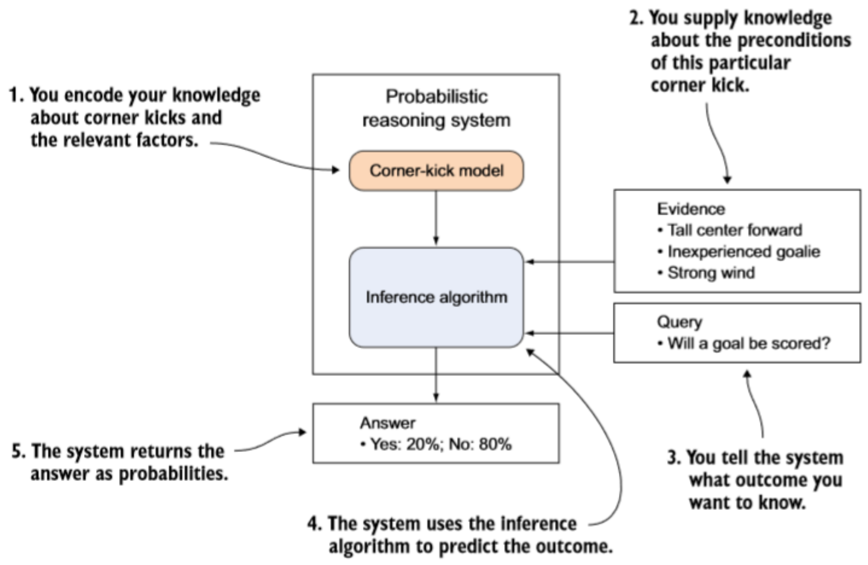

1. You encode your knowledge about corner kicks and the relevant factors.
Probabilistic reasoning system
Corner-kick model
Inference algorithm
2. You supply knowledge about the preconditions of this particular corner kick.
Evidence
• Tall center forward
• Inexperienced goalie
• Strong wind
Query
• Will a goal be scored?
3. You tell the system what outcome you want to know.
Answer
• Yes: 20%; No: 80%
5. The system returns the answer as probabilities.
4. The system uses the inference algorithm to predict the outcome.

Figure 1: General workflow example of a probabilistic reasoning system

- Wrappers for Python, R, Julia and other languages It also provides basic plotting methods for the posterior distributions of parameters

Edward is a python library that allows working with graphical models, neural networks, implicit generative models and Bayesian programs. It supports most of the abovementioned sampling methods and relies on TensorFlow [12]

Pyro is a PPL built in Python that focuses on deep learning and artificial intelligence. It uses the deep learning platform PyTorch as a backend [1].

Figaro is available as a Scala library. Models in Figaro are treated as objects and a focus is laid on the possibility that more specialized models are derived from base models so that the former can inherit properties from the latter. Figaro uses the Metropolis-Hastings algorithm for inference, a generalized version of the Metropolis algorithm described in Subsection 2.2 [13].

PyMC3 is the PPL used in this thesis. It is described in detail in the following Subsection 3.2.

3.2 PyMC3

PyMC3 is an open-source probabilistic programming framework for Python. The following explanations are taken from Salvatier et al. [14]. Bayesian models in PyMC3 are described by encoding the prior, the sampling and the posterior distributions through three types of random variables: Stochastic, deterministic and observed stochastic ones. Stochastic random variables have values which are in part determined randomly, according to a chosen distribution. Commonly used probability distributions like Normal, Binomial etc. are available for this. Deterministic random variables, on the other hand, are not drawn from a distribution but are calculated by fixed rules from other variables, for example by taking the sum of two variables. Lastly, there are the observed stochastic random variables which are similar to stochastic random variables except that they are treated as observed variables and therefore get assigned observed data when being initialized. This kind of random variable is used to represent distributions of data whereas stochastic variables are used to represent distributions of parameters.

PyMC3 mainly uses sampling techniques to draw inference on posterior distributions. It focuses especially on the No-U-Turn Sampler, a Markov Chain Monte Carlo algorithm that relies on gradient information to generate samples from posterior distributions, as explained in Subsection 2.2. PyMC3 also provides basic methods for plotting posterior distributions.

The code piece in Listing 1 shows a simple example of a Bayesian model taken from Salvatier et al. [14]. There, the data X1, X2 and Y is used to fit a regression model. First, prior distributions for the model parameters are set up as stochastic random variables, then the regression model itself is specified by a deterministic random variable and lastly the sampling distribution is described by an observed stochastic random variable to which the observed outcome Y is given as a parameter. Finally, the posterior distribution is simulated by drawing 500 samples from it.

```python
import pymc3 as pm
basic_model = pm.Model()
with basic_model:
    # Describe prior distributions of model parameters.
    # Stochastic variables
    alpha = pm.Normal('alpha', mu=0, sd=10)
    beta = pm.Normal('beta', mu=0, sd=10, shape=2)
    sigma = pm.HalfNormal('sigma', sd=1)
    # Specify model for the output parameter.
    # Deterministic variable
    mu = alpha + beta[0]*X1 + beta[1]*X2
    # Likelihood of the observations.
    # Observed stochastic variable
    Y_obs = pm.Normal('Y_obs', mu=mu, sd=sigma, observed=Y)

# Model fitting by using sampling strategies
with basic_model:
    # Draw 500 posterior samples
    trace = pm.sample(500)
    pm.summary(trace)
```

Listing 1: Example code of a simple Bayesian model using PyMC3

4 Lumen

In this section, the software Lumen is introduced and its abilities and uses are elucidated.

Lumen is an interactive web-frontend designed for graphically displaying of models and data [15]. It uses modelbase as backend, a Python package that provides means for fitting models to data and manipulate models by marginalization and conditionalization [16]. Figure 2 shows an exemplary view of the frontend. After a model is loaded, all random variables of the model are shown in the 'Schema'-panel on the left. A user then has to select the variables that should be displayed by dragging the according names into the 'Specification'-panel in the middle. There, a choice has to be made on which channel the information of a variable should be displayed. The available channels are:

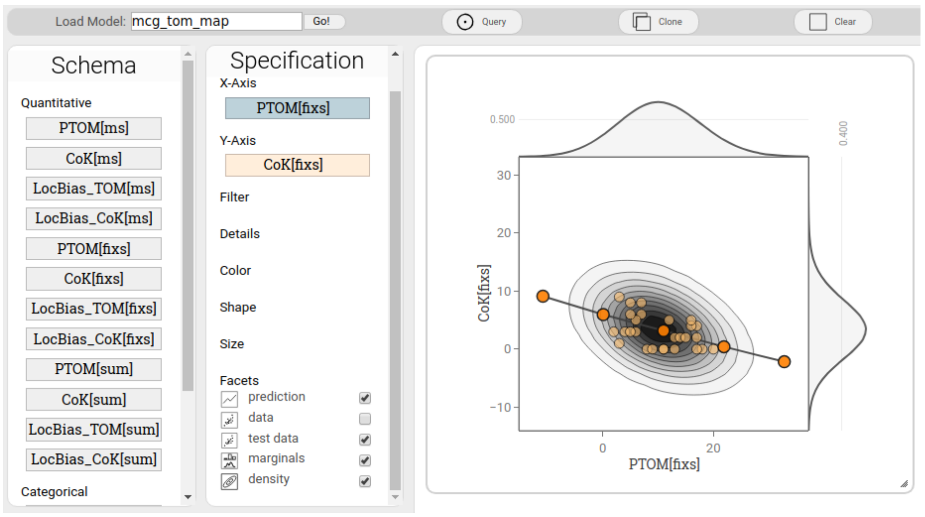
- X-Axis
- Y-Axis
- Color
- Shape
- Size

Once one or more variables are moved there, the according visualization is shown in the right panel. Additionally, a filter can be set to only consider a specified interval within the range of a variable. The user can also select which elements of the model should be displayed. The following elements can be chosen:

- prediction
- data
- test data
- marginals
- density

For the prediction, it is possible to specify any of the selected variables as dependent or independent. It is also possible to specify additional independent variables for the prediction by dragging them into the 'Details' field.

Visualizing a model as described above can require marginalizing and conditioning it multiple times. The necessary operations are performed by the modelbase backend [16]. How modelbase was complemented to enable a visualization of models built with PyMC3 is described in Section 6.


Load Model: mcg_tom_map

X-Axis
PTOM[fixs]

Y-Axis
CoK[fixs]

Facets:
- prediction ✓
- data ☐
- test data ✓
- marginals ✓
- density ✓

Figure 2: View on the lumen frontend

5 Example: Building a probabilistic model

The practical part of the thesis starts here. To check if PyMC3 is applied correctly, a Bayesian model is fitted first without the use of a PPL. Then the same model is fitted by using PyMC3. The results of both approaches are compared.

Assume we want to fit a model with the two dimensions x and mu. One of them, x, we can observe, the other one, mu, we can't observe. Furthermore, we set the following priors:

$$\begin{aligned} mu &\sim N(0,1) \\ x &\sim N(mu,1) \end{aligned} \tag{10}$$

Assume again that we observe some data X of the observable dimension x. We now want to compute the joint posterior distribution of both dimensions mu and x, conditioned on the observed data X.
The data X is generated as follows:

```python
# Generate data
numpy.random.seed(1)
size = 100
mu = numpy.random.normal(0,1,size=size)
sigma = 1
X = numpy.random.normal(mu,sigma,size=size)
```

Listing 2: Data generation for the example model

5.1 Creating a simple Bayesian model without PyMC3

The joint posterior distribution is computed as shown in Equation 11.

$$\begin{aligned} & P(x,mu|X) \\ =& P(x|mu,X) * P(mu|X) \\ =& P(x|mu) * P(mu|X) \\ =& P(x|mu) * P(X|mu) * P(mu)/P(X) \\ =& P(x|mu) * P(X|mu) * P(mu)/\int P(X,mu)\ d\,mu \\ =& P(x|mu) * P(X|mu) * P(mu)/\int P(X|mu) * P(mu)\ d\,mu \end{aligned} \tag{11}$$

Each element of the last line of Equation 11 can be implemented in Python relatively easy:
The likelihood for any new data point given mu, $P(x|mu)$, is according to our model assumptions the density of x in a normal distribution with mean mu. It can be be implemented as shown in Listing 3.

```python
from scipy.stats import norm
def likelihood_x(x,mu):
    density = norm.pdf(x, loc=mu, scale=1)
    return density
```

Listing 3: Implementation of the likelihood for one data point

The likelihood of the observed data X given mu, $P(X|mu)$, is the product of the likelihoods for each data point in X. It can be implemented as shown in Listing 4.

```python
def likelihood_X(X,mu):
    res = 1
    for point in X:
        res *= likelihood_x(point,mu)
    return res
```

Listing 4: Implementation of the likelihood for the observed data

The prior probability for any mu, $P(mu)$, is just the density of the standard normal distribution at the point mu, as shown in Listing 5:

```python
def prior_mu(mu):
    density = norm.pdf(mu, loc=0, scale=1)
    return density
```

Listing 5: Implementation of the prior for mu

The last part of the equation, the integral of the product between likelihood and prior, is computationally more intensive but nevertheless easy to implement as well. Listing 6 shows the implementation.

```python
def likelihood_times_prior_mu(X,mu):
    return likelihood_X(X,mu) * prior_mu(mu)

def prior_X(X):
    res = integrate.quad(
        lambda mu: likelihood_times_prior_mu(X,mu),
        a=-np.inf,
        b=np.inf
    )[0]
    return res
```

Listing 6: Implementation of the prior for X

At last, we only have to multiply all those pieces to get to a posterior joint density for any combination of X and mu. Listing 7 shows the according implementation.

```python
def joint_posterior(x,mu,X):
    res = likelihood_x(x,mu) * likelihood_X(X,mu) *
        prior_mu(mu) / prior_X(X)
    return res
```

Listing 7: Implementation of the joint posterior distribution

The joint posterior distribution can now be visualized as it is done in Figure 3. The upper left figures show histograms of the data points of x and mu. The upper right figures show the marginalized posterior distributions for both variables. The bottom figure shows the density of the joint posterior distribution as a contour plot, along with the data points. We see that mu is very narrowly contained around zero whereas x has a much broader distribution. A relationship between mu and x is not visible. This is astonishing at first: Since we know that x is dependent on mu, I would expect this dependency to show up in the visualization. In other words, I would expect the probability density for a high mu and a high x to be relatively similar to the density of a mu near zero and a high x, and to be much higher than the density of a high mu and a low x. Let's look again at the formula:

$$P(x, mu|Y) = P(x|mu) * P(X|mu) * P(mu) / \int P(X|mu) * P(mu) \; d \; mu \quad (12)$$

To evaluate the above mentioned expectations, it is not necessary to keep the normalizing constant since it does not affect the order of the values. So we keep:

$$P(x, mu|Y) = P(x|mu) * P(X|mu) * P(mu) \quad (13)$$

When we consider only the first and the last term of from the right of this formula, it behaves exactly as the expectation above states:

$$P(x = 1|mu = 1) * P(mu = 1) = P(x = 1|mu = 1) * P(mu = 0) \quad (14)$$

and

$$P(x = 1|mu = 1) * P(mu = 1) > P(x = -1|mu = 1) * P(mu = 1) \quad (15)$$

What was messing with the expectations is the term in the middle which gives the likelihood of the observed data. For a value of mu that is far from the true mean, that term quickly becomes extremely small since all the data points speak against it. That's why the distribution is so strongly centered to the middle: All the outer values of mu are getting assigned extremely low density values by this middle term. The dependence between mu and x is in fact still there in the joint posterior distribution, it's just overlapped by the centering effect of the data. To understand this example better, I make changes in the model and/or in the data generation and look how those changes affect the posterior distributions. Firstly, the model is learned without any data points. It delivers the distribution that is shown in Figure 4. The according histograms are of course empty since no data was generated that could be displayed there. The

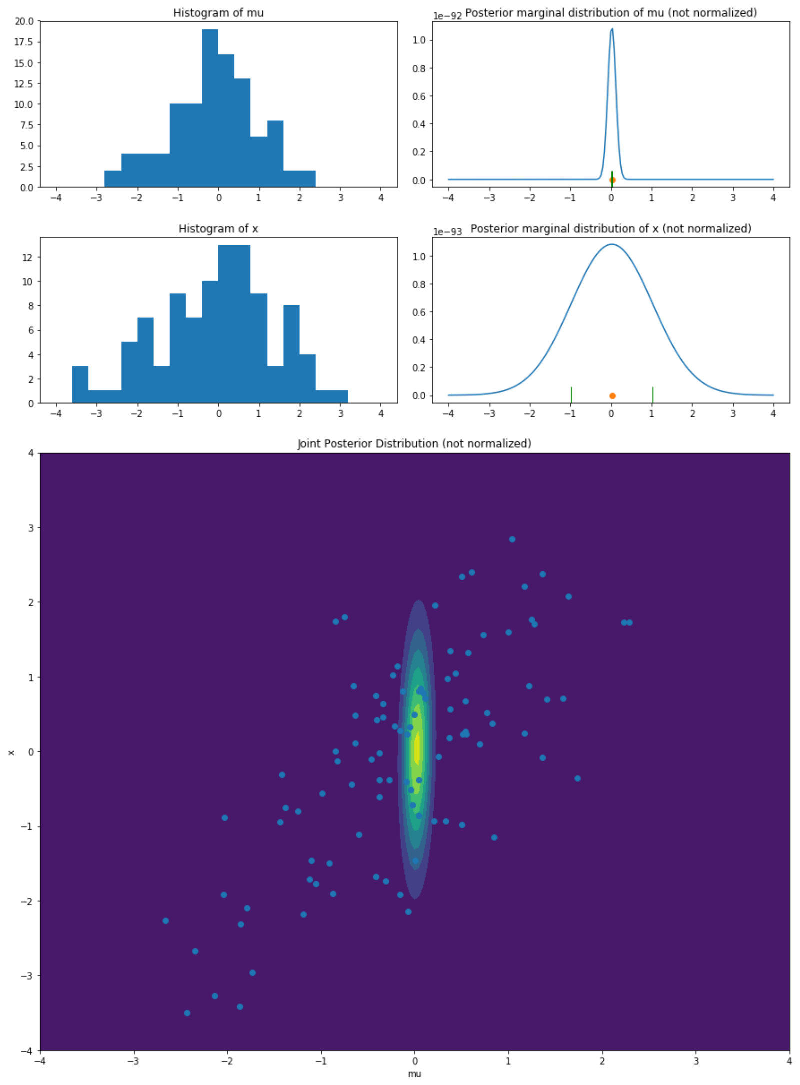


Figure 3: Posterior distributions of the example model

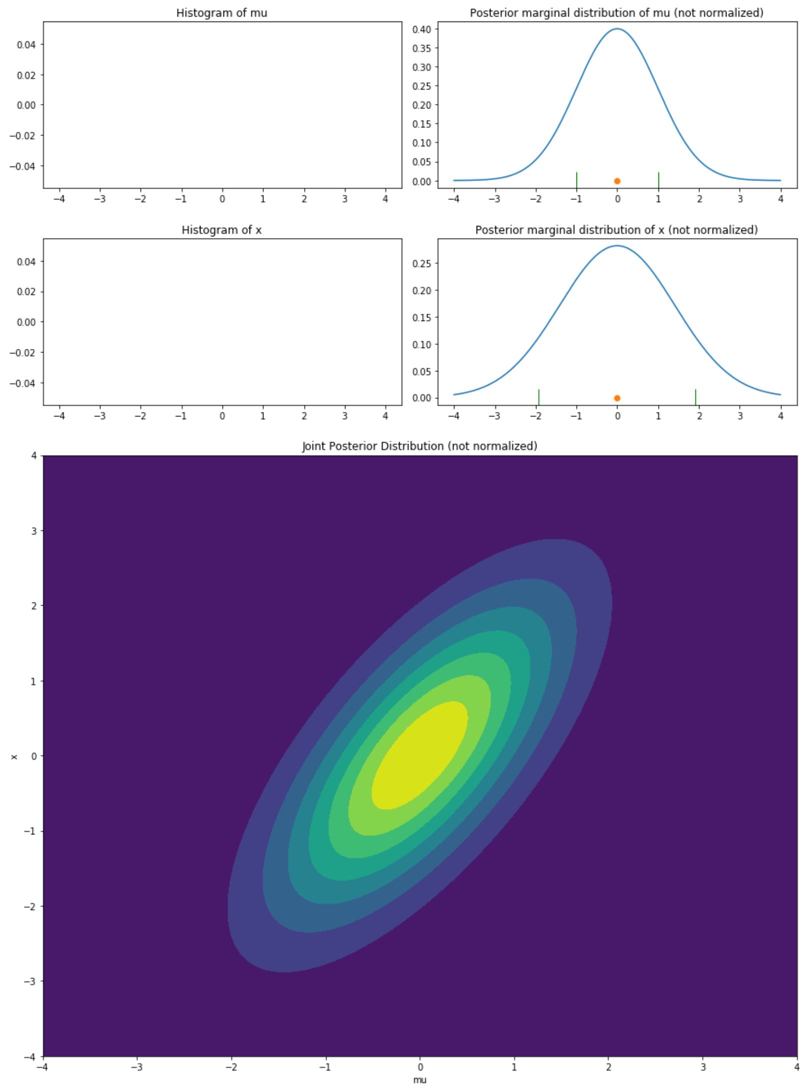


Figure 4: Posterior probability densities of the example model without any data points

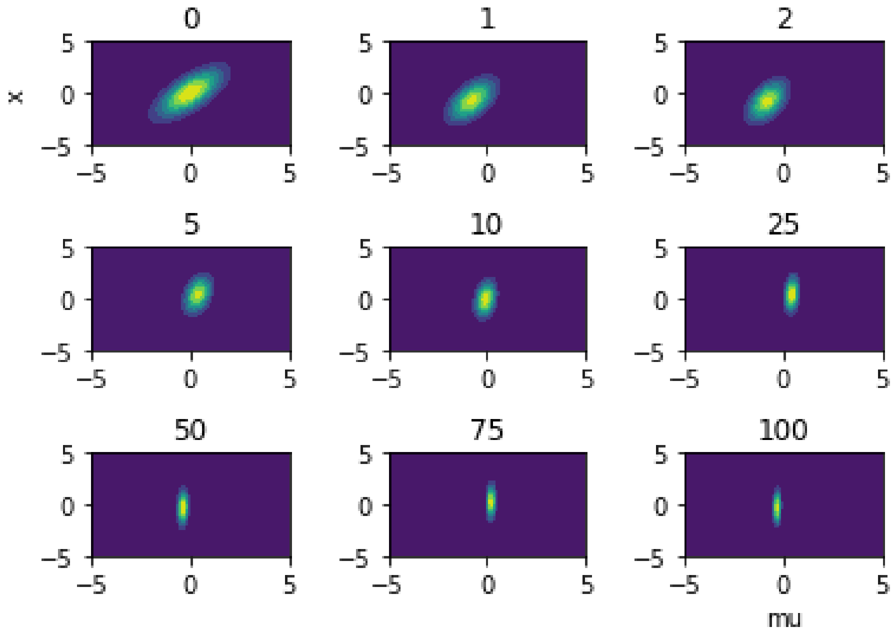


Figure 5: Posterior distributions for different numbers of data points. The number on top of each plot shows how many data points were used for fitting the model.

dependency between the variables is in Figure 4 clearly visible since the data points do not influence the plot anymore. The more data points we include, the more the joint posterior distribution is compressed to the center, as shown in Figure 5. There, the density is depicted for different numbers of data points. It is clearly visible how the dependency between the variables diminishes as the number of data points increases.
Figure 6 shows a comparison between a posterior distribution where the parameter for the standard deviation in the data generation and in the prior is set to 1, and a posterior distribution where this parameter is set to 5. Astonishingly, the standard deviation of the posterior is nearly the same in both cases whereas the mean of the variables in the latter distribution is at around -1.5.

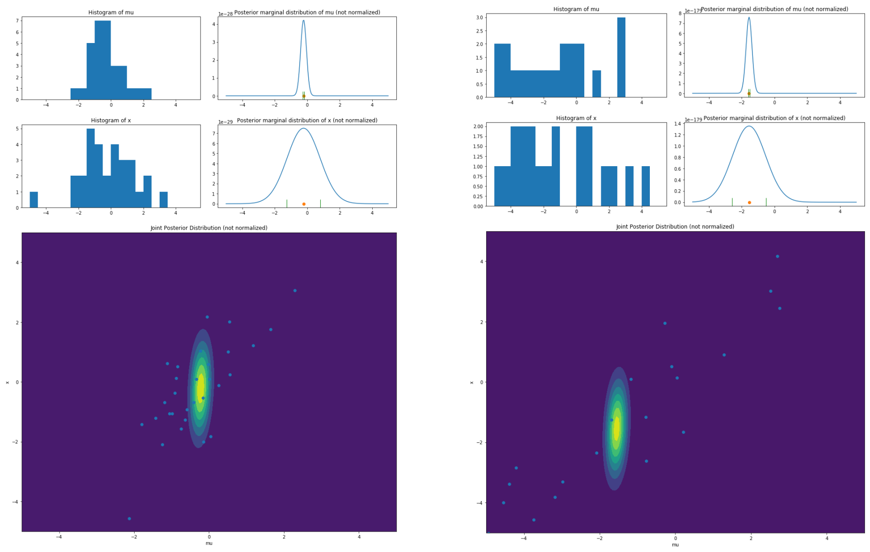


Figure 6: Posterior probability densities of the example model. For the plots on the left, the standard deviation in the data generation and in the prior was set to 1. For the plots on the right, it was set to 5.

We now change the data generating mechanism to a non-hierarchic process shown in Equation 16, so that only a single variable is drawn from a normal distribution with mean 0 and the combined standard deviation from the previous two variables.

$$x \sim N(0, 2) \tag{16}$$

As one can see in Figure 7, the distribution looks very similar. The histogram for the variable mu is not shown there, since mu is no longer part of the data generating mechanism. In the model, mu is still considered and a posterior distribution for it is learned. The similarity between Figure 3 and Figure 7 is not surprising since the data generated from the according mechanisms in Equation 10 and Equation 16 looks very similar, even if the actual data generating mechanisms are different.

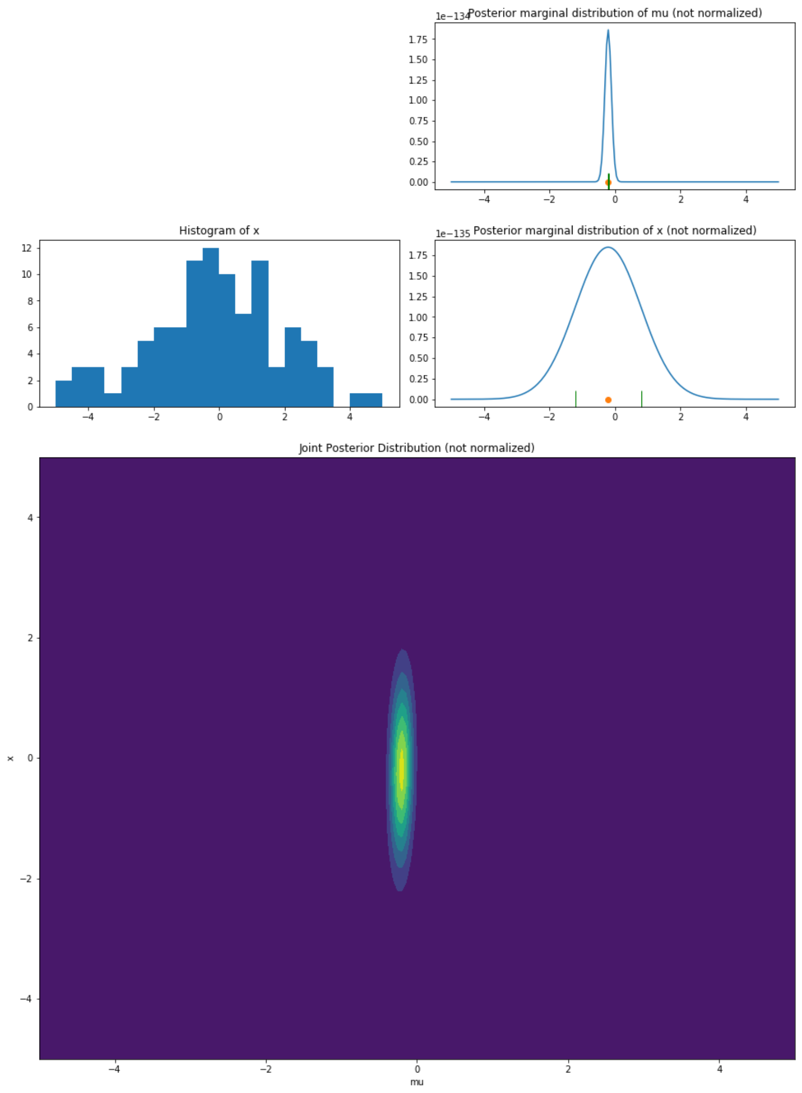


Figure 7: Posterior probability densities of the example model with altered data generation

5.2 PyMC3 example of a simple Bayesian model

The model from Subsection 5.1 is now created by using PyMC3 with the code shown in Listing 8.

```
import numpy as np
import pandas as pd
import pymc3 as pm
import matplotlib.pyplot as plt

# Generate data
np.random.seed(2)
size = 100
mu = np.random.normal(0,1,size=size)
sigma = 1
X = np.random.normal(mu,sigma,size=size)

# Specify model
basic_model = pm.Model()
with basic_model:
    sigma = 1
    mu = pm.Normal('mu',mu=0,sd=sigma)
    X = pm.Normal('X',mu=mu,sd=sigma,observed=X)

# Draw samples from posterior
nr_of_samples = 100
with basic_model:
    trace = pm.sample(nr_of_samples,chains=1)
    samples_mu = trace['mu']
    samples_X = np.random.normal(
        samples_mu,1,size=nr_of_samples)
```

Listing 8: Code used to specify the example model in PyMC3

First, the same data is generated as in Subsection 5.1. Then, *mu* and X are specified, *mu* as a stochastic random variable and X as an observed stochastic random variable. Finally, samples from the posterior distribution are drawn using the sample-method. Figure 8 shows these samples in comparison to the posterior distribution from Figure 3. The red crosses depict the samples, the background distribution is the same as in Figure 3. It is clearly visible that the samples from the PyMC3 implementation match the distribution from the manual implementation, so it is assumed that the PyMC3 captures the model correctly.

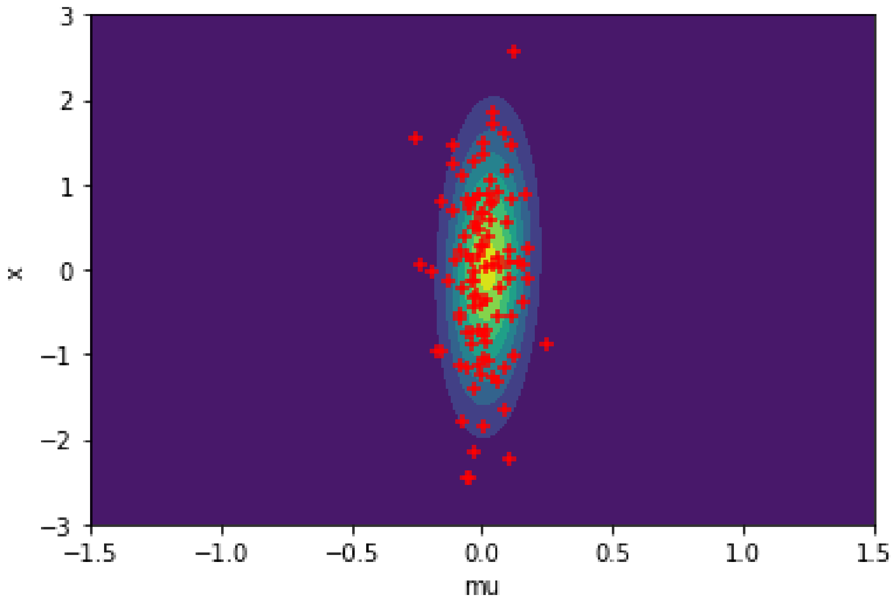


Figure 8: Comparison between the posterior samples drawn by PyMC3 and the posterior distribution calculated without PyMC3.

6 Integration of PyMC3 in Lumen

This section describes how the main goal of the thesis is achieved: The implementation of an interface that enables Lumen to display models built in PyMC3.

Visualizing a model as described in Section 4 can require marginalizing and conditioning multiple times. The necessary operations are performed by the modelbase backend in which each model type is represented by a class. This class is required to provide certain methods that allow to manipulate the model in a way that proper data can be generated for the Lumen frontend. The most important operations that are embedded in these methods are:

- fitting a model
- marginalization
- conditioning
- computing the probability density for a given data point

Visualizing models built with PyMC3 in Lumen requires the abovementioned operations to be carried out for those models. Expanding Lumen by implementing these operations for arbitrary models poses the main task of the thesis at hand. Below is explained how this was done for each of these operations.

Fitting a model in Bayesian statistics means finding the joint posterior distribution. In the context of Probabilistic Programming it means drawing samples from all random variables so that the joint posterior distribution can be approximated. PyMC3 provides methods for sampling from both observed and unobserved random variables which only have to be applied here. Figure 9 shows the implementation of the process. There, the variable $self.model\_structure$ holds the full model that was created before in PyMC3. As an additional feature, since in Bayesian data analysis there is no test data, the samples that were generated during the fitting process are assigned to the variable that originally was supposed to hold the test data. This way, the sample points can later be visualized in Lumen, albeit to some extent incorrectly labeled as test data.

```
import pymc3 as pm
def _fit(self):
    with self.model_structure:
        # Draw samples
        nr_of_samples = 500
        trace = pm.sample(nr_of_samples,chains=1,cores=1)
    for varname in trace.varnames:
        self.samples[varname] = trace[varname]
        ppc = pm.sample_ppc(trace)
    for varname in self.model_structure.observed_RVs:
        # each sample has 100 draws in the ppc,
        # so take only the first one for each sample
        self.samples[str(varname)] =
            [samples[0]
                for samples in np.asarray(ppc[str(varname)])
            ]
    # Change order of sample columns
    # so that it matches order of fields
    self.samples = self.samples[self.names]
    self.test_data = self.samples
    return ()
```

Listing 9: Implementation of fitting a model

Marginalizing a model normally means that one has to integrate over the variables that should be removed. In the case of Probabilistic Programming where we have only samples instead of analytical functions, it is far easier: The samples of the variables one wants to marginalize out are just removed and that's it. The corresponding code is shown in Listing 10.

```
def _marginalizeout(self, keep, remove):
    # Remove all variables in remove
    for varname in remove:
        if varname in list(self.samples.columns):
            self.samples = self.samples.drop(varname,axis=1)
    return ()
```

Listing 10: Implementation of marginalizing a model

Conditioning a model follows the same principle as marginalizing: Samples that do not fall inside the interval dictated by the condition are just removed. This operation is implemented as shown in Listing 11.

```
def _conditionout(self, keep, remove):
    names = remove
    fields =
        self.fields if names is None else self.byname(names)
    # Konditioniere auf die Domaene der Variablen in remove
    for field in fields:
        # filter out values smaller than domain minimum
        filter = self.samples.loc[:,str(field['name'])] >
            field['domain'].value()[0]
        self.samples.where(filter, inplace = True)
        # filter out values bigger than domain maximum
        filter = self.samples.loc[:,str(field['name'])] <
            field['domain'].value()[1]
        self.samples.where(filter, inplace = True)
    self.samples.dropna(inplace=True)
    return ()
```

Listing 11: Implementation of conditioning a model

Getting a probability density in the context of Probabilistic Programming is not as straightforward as in classical statistics since we do not have probability density function. This function instead can be approximated by the samples drawn from the posterior. In the implementation at hand, the chosen approximation method is the kernel density estimation using Gaussian kernels, provided by the scikit-learn package [17]. In this method, each data point is assigned a Gaussian curve centered on the point. The density for a given point is then determined by summing up the densities of all the Gaussians. It is implemented as shown in Listing 12.

```
def _density(self, x):
    X = self.samples.values
    kde = KernelDensity(
        kernel='gaussian', bandwidth=0.1
    ).fit(X)
    x = np.reshape(x,(1,len(x)))
    logdensity = kde.score_samples(x)[0]
    return np.exp(logdensity).item()
```

Listing 12: Implementation of computing the probability density for a given point

Treating independent variables Independent variables are variables that are part of a model but for which no probability distribution is learned. Displaying independent variables currently is not supported in Lumen. That means that a model is allowed to include such variables, but the independent variables themselves cannot be visualized.

7 Example Cases

In this section, example models are created in PyMC3 to show how the finished interface performs in practice.

7.1 First simple example

As a basic example to implement in PyMC3 we choose the model from Section 5. The expectation here is that the resulting Lumen plot should look much like the visualizations in Figure 3. The model contains the random variables X and mu, the data for it was generated by the following distributions:

$$\begin{aligned} X &\sim N(\mu, 1), \\ \mu &\sim N(0, 1), \end{aligned} \tag{17}$$

The priors for X and mu match exactly these distributions. The corresponding model is described in PyMC3 as shown in Listing 13.

```
basic_model = pm.Model()
with basic_model:
    mu = pm.Normal('mu', mu=0, sd=1)
    X = pm.Normal('X', mu=mu, sd=1, observed=data['X'])
```

Listing 13: PyMC3 model of use case 1

Visualizing this model in Lumen results in the plot shown in Figure 9. Considering the different ranges in the mu-axis, the plot looks quite similar to the visualizations in Figure 3, meeting the expectations.

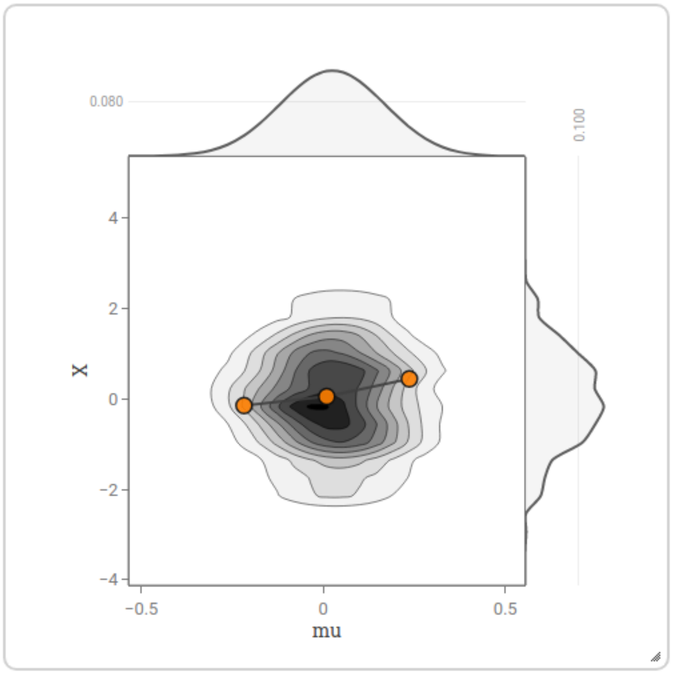


Figure 9: Lumen visualization of the example model from Section 5

7.2 Linear Regression

As another basic use case, a linear regression model used by Salvatier et al to illustrate the PyMC3 functionality [14] is visualized. The model is explained by Salvatier et al as follows:
"We are interested in predicting outcomes Y as normally-distributed observations with an expected value mu that is a linear function of two predictor variables, X_1 and X_2:

$$\begin{aligned} Y &\sim N(\mu, \sigma^2), \\ \mu &= \alpha + \beta_1 X_1 + \beta_2 X_2 \end{aligned} \tag{18}$$

where α is the intercept, and β_i is the coefficient for covariate X_i, while σ represents the observation or measurement error."
The following priors are assigned to the random variables:

$$\begin{aligned} \alpha &\sim N(0, 10), \\ \beta_1 &\sim N(0, 10), \\ \beta_2 &\sim N(0, 10), \\ \sigma &\sim |N(0, 1)| \end{aligned} \tag{19}$$

The code creating this model is shown in Listing 14. It differs from the implementation of Salvatier et al in that the former independent variables X_1 and X_2 are now treated as random variables. This change was made since we want to display these variables in Lumen, and Lumen currently is not able to display independent variables.

```
np.random.seed(123)
alpha, sigma = 1, 1
beta_0 = 1
beta_1 = 2.5
size = 100
X1 = np.random.randn(size)
X2 = np.random.randn(size) * 0.2
Y = alpha + beta_0 * X1 + beta_1 * X2 +
    np.random.randn(size) * sigma
data = pd.DataFrame({'X1': X1, 'X2': X2, 'Y': Y})

basic_model = pm.Model()
with basic_model:
     # Priors for unknown model parameters
     alpha = pm.Normal('alpha', mu=0, sd=10)
     beta_0 = pm.Normal('beta_0', mu=0, sd=10)
     beta_1 = pm.Normal('beta_1', mu=0, sd=10)
     sigma = pm.HalfNormal('sigma', sd=1)
     # Expected value of outcome
     mu = alpha + beta_0 * data['X1'] + beta_1 * data['X2']
     # Likelihood (sampling distribution) of observations
     Y = pm.Normal('Y', mu=mu, sd=sigma, observed=data['Y'])
     X1 = pm.Normal('X1',
         mu=data['X1'],
         sd=sigma,
         observed=data['X1']
         )
     X2 = pm.Normal('X2',
         mu=data['X2'],
         sd=sigma,
         observed=data['X2']
         )
```

Listing 14: PyMC3 model of linear regression example

Plotting the two variables X_1 and X_2 against each other and comparing this to the posterior probability density for those two variables results in the plot shown in Figure 10. The Lumen visualization here is able to clearly identify a mismatch between model and data: The spread of the posterior distribution is too high in X_2 direction and too low in X_1 direction. It might be that the change of the variables X_1 and X_2 from independent to random variables introduced this discrepancy.

7.3 Eight schools model

The problem shown below is often used in the literature as an example to illustrate Bayesian modeling [4][18][19]. It is about eight schools where the impact of a coaching program on test results was estimated. For each school, a separate estimate and a standard error was calculated. These estimates are

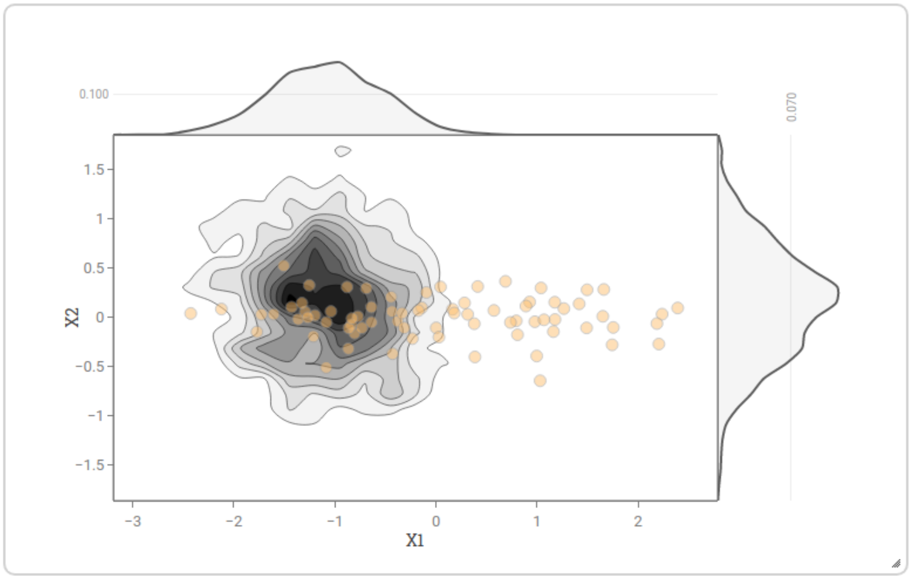
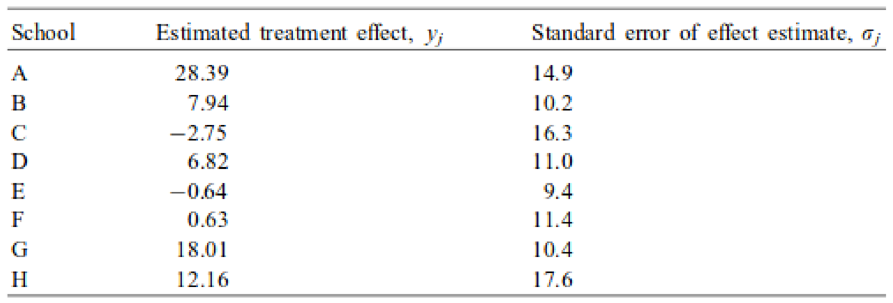


Figure 10: Lumen visualization of two observed variables from the linear regression model

shown in Table 1 and serve as data for our modeling problem.
Sinharay et al now fit a hierarchical model for this process and apply a number of methods to check in which ways their models deviates from the data [18]. Below, one of their methods is recapitulated and it is analyzed if the conclusions they draw can also be deduced from a visualization of the model in Lumen. Sinharay et al use a hierarchical normal-normal model shown in Equation 20 to capture the mechanism: An estimated treatment effect y_i is there seen as an independent draw from a normal distribution that is centered around the real treatment effect θ_i. However, it is assumed that each school potentially has a different treatment effect which, in turn, is independently drawn for every school from the same normal distribution. The mean μ and the standard deviation τ of this normal distribution are hyperparameters with uniform priors.

$$\begin{aligned} y_i &\sim N(\theta_i, \sigma_i), \\ \theta_i &\sim N(\mu, \tau), \\ \mu &\sim 1, \\ \tau &\sim 1, \\ i &= 1, ..., 8 \end{aligned} \tag{20}$$

Simulations from posterior distributions are obtained for τ, μ and θ by finding the conditional distributions $p(\tau|y)$, $p(\mu|\tau, y)$ and $p(\theta|\tau, \mu, y)$ and then sampling from these distributions, where y denotes the observed estimated treatment effects. For each draw from those distributions, eight data points y^{rep} are gen-



School	Estimated treatment effect, y_j	Standard error of effect estimate, σ_j
A	28.39	14.9
B	7.94	10.2
C	−2.75	16.3
D	6.82	11.0
E	−0.64	9.4
F	0.63	11.4
G	18.01	10.4
H	12.16	17.6

Table 1: Observed effects and standard errors of coaching programs on test scores in 8 schools

erated and a number of test statistics was calculated on them: the largest of the eight observed outcomes, the smallest of the eight observed outcomes,the average, and the sample standard deviation. Those test statistics were then displayed graphically and compared to the according test statistic of the observed effects y, resulting in the plots shown in Figure 11
To visualize the model in Lumen, the model is first generated in PyMC3 using the code shown in Listing 15. Then, the plots generated by Sinharay et al shown in Figure 11 are replicated for the PyMC3 model to check if it was correctly carried over to PyMC3. The results are shown in Figure 12. The plots of both models look very similar, leading to the conclusion that both implementations correctly describe the same model.

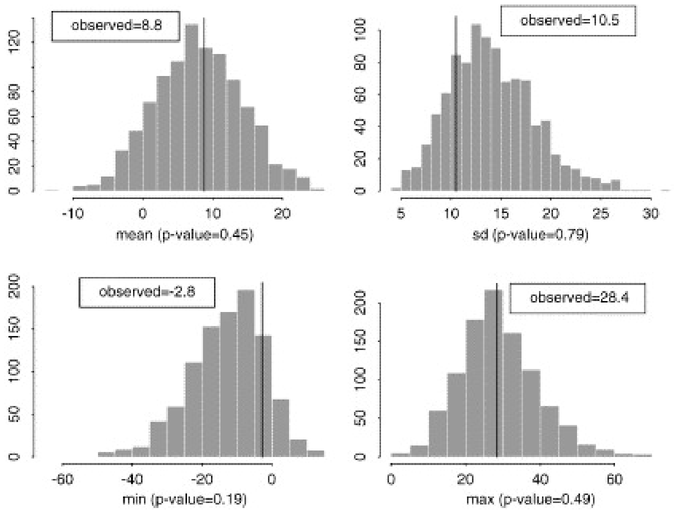


Figure 11: Histograms of the test statistics of the simulated values generated by Sinharay et al

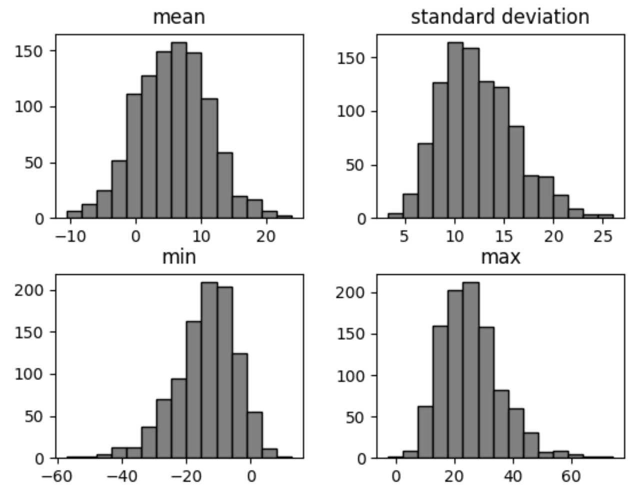


Figure 12: Histograms of the test statistics of the simulated values generated by the PyMC3 model

```
scores = [28.39,7.94,-2.75,6.82,-0.64,0.63,18.01,12.16]
standard_errors = [14.9,10.2,16.3,11.0,9.4,11.4,10.4,17.6]
data = pd.DataFrame({'test_scores': scores,
    'standard_errors': standard_errors})

with pm.Model() as normal_normal_model:
    tau = pm.Uniform('tau',lower=0,upper=10)
    mu = pm.Uniform('mu',lower=0,upper=10)
    theta_1 = pm.Normal('theta_1', mu=mu, sd=tau)
    theta_2 = pm.Normal('theta_2', mu=mu, sd=tau)
    theta_3 = pm.Normal('theta_3', mu=mu, sd=tau)
    theta_4 = pm.Normal('theta_4', mu=mu, sd=tau)
    theta_5 = pm.Normal('theta_5', mu=mu, sd=tau)
    theta_6 = pm.Normal('theta_6', mu=mu, sd=tau)
    theta_7 = pm.Normal('theta_7', mu=mu, sd=tau)
    theta_8 = pm.Normal('theta_8', mu=mu, sd=tau)
    theta = [theta_1,theta_2,theta_3,theta_4,
        theta_5,theta_6,theta_7,theta_8]

    test_scores = pm.Normal('test_scores',mu=theta,
        sd=data['standard_errors'],
        observed=data['test_scores'])
```
Listing 15: PyMC3 model of the eight schools example

Generating the same plots in Lumen is not possible since the calculation and display of summary statistics is not supported there. Instead, the observed estimates can be shown and compared to their posterior distribution, as it is done in Figure 13. It looks not very informative, though. One problem is that the data consists of very few points. Another problem is that the visualization is fragmented into multiple peaks, making it hard to judge the overall fit to the data.

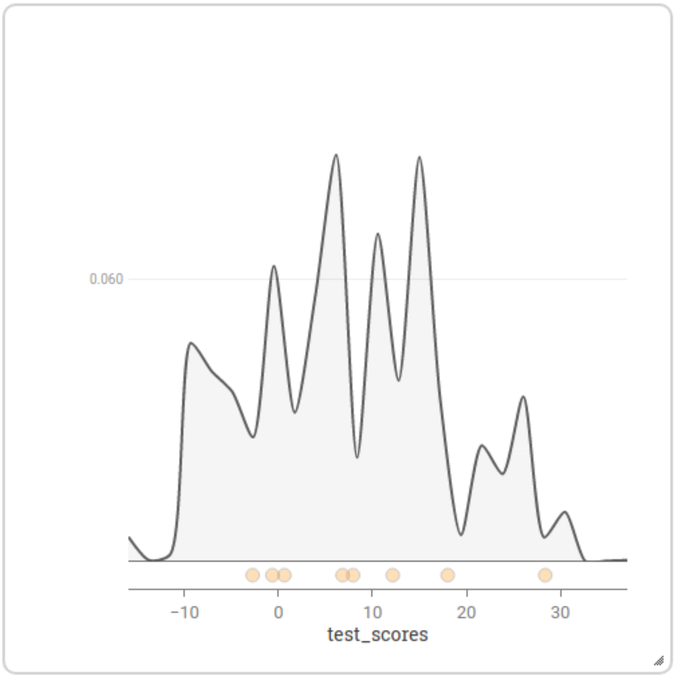


Figure 13: Visualization of the coaching effects for the eight schools model in Lumen

8 Discussion

Each of the abovementioned example models could successfully be visualized in Lumen via the interface created during the thesis at hand. In the linear regression example in Subsection 7.2, the Lumen visualization helped detecting the misfit between model and data. However, models with a big number of variables were not tested during this work. For such complex models, a visualization could likely be hindered by the circumstance that the number of samples needed to cover the whole parameter space probably increases quickly with the number of parameters. This holds especially for conditioning a model: The probability that even one sample point falls within such an interval becomes very small for increasing number of parameters.

Apart from that, the potential benefits that Lumen could bring for users of probabilistic models still have to be pointed out more specifically: Validating models is a possibility here, but this is typically done by displaying test statistics of simulated data. As of now, Lumen instead only displays the probability density of the simulated data itself, and does not aggregate it to test statistics. In the eight schools example in Subsection 7.3, it did not perform so well in showing the similarities or discrepancies between model and data, compared to the classical method of displaying test statistics. Hence, expanding Lumen so that it can display distributions of test statistics might be a reasonable step for future work.

On the other hand, it is also important to learn how probabilistic models are currently used and validated in practice. Other ways of validating probabilistic models might be in use, that are not mentioned so much in literature as the abovementioned one.

9 Conclusion and outlook

In the thesis at hand an interface was implemented to link the visualization software Lumen to the probabilistic programming language PyMC3. To visualize a model in Lumen, it has to be possible to marginalize and condition the model as well as estimating a probability density for a given point. PyMC3 draws inference by generating samples from a posterior distribution, so the necessary operations to perform the abovementioned tasks also work with these samples: Marginalizing and conditioning is done by removing columns or rows, respectively, from the samples. Estimating a probability density is done by approximating the samples through a sum of Gaussians and then take the density from there.
In the course of the thesis a simplified Bayesian model was created and visualized. Even this most simplified model did not behave as intuitively expected. Since it is very difficult to correctly assess the behavior of a Bayesian model by just looking at the code, especially for complex models, but even, as shown here, for the most simple ones, validating a model by visualizing it becomes extremely important. The interface implemented here makes it possible to check PyMC3 models in Lumen which can help spotting discrepancies between the model and the expectations of the researcher.
Several fields remain where future work could bring rewarding results:

- In terms of validating models, it could be useful to enable Lumen to aggregate observed and simulated data and display those aggregations. This is a widespread method for validating Bayesian models and would therefore pose a valuable addition to the existing abilities of the program.

- Being able to display independent variables would also be beneficial. Although they have no posterior distribution, independent variables are often important parts of a model and visualizing them can give valuable insights for judging the model fitness.

- It is likely that models used in practice are much more complex than the ones showed in the thesis at hand. It is also likely that the performance of Lumen might have difficulties in dealing with increasing model complexity, due to a lack of samples. Enabling the program to stay functional and performant when dealing with more complex models is therefore crucial for its broader application.

Lumen is already providing useful features for performing visual analytics. By realizing these improvements it can develop its full potential as a powerful tool for validating and exploring probabilistic models.

List of Figures

List of Tables

Listings

References

[1] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman, "Pyro: Deep Universal Probabilistic Programming," *Journal of Machine Learning Research*, 2018.

[2] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani, "Probabilistic programming," in *Proceedings of the on Future of Software Engineering*, pp. 167–181, ACM, 2014.

[3] L. Hardesty, "Probabilistic programming does in 50 lines of code what used to take thousands," Apr. 2015. [Online; accessed 23-August-2018].

[4] D. B. D. A. V. John B. Carlin, Hal S. Stern and D. B. R. A. Gelman, *Bayesian Data Analysis, 3Rd Edn.* T&F/Crc Press, 2014.

[5] "Definition of markov chain in us english." [Online; accessed 22-March-2019].

[6] H. F. Martz and R. A. Waller, "Bayesian methods," in *Methods in Experimental Physics*, pp. 403–432, Elsevier, 1994.

[7] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[8] M. D. Hoffman and A. Gelman, "The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo.," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1593–1623, 2014.

[9] A. Gelman, "Exploratory data analysis for complex models," *Journal of Computational and Graphical Statistics*, vol. 13, no. 4, pp. 755–779, 2004.

[10] A. Pfeffer, *Practical Probabilistic Programming.* Manning Publications, 2016.

[11] A. Gelman, D. Lee, and J. Guo, "Stan: A probabilistic programming language for bayesian inference and optimization," *Journal of Educational and Behavioral Statistics*, vol. 40, no. 5, pp. 530–543, 2015.

[12] D. Tran, A. Kucukelbir, A. B. Dieng, M. Rudolph, D. Liang, and D. M. Blei, "Edward: A library for probabilistic modeling, inference, and criticism," *arXiv preprint arXiv:1610.09787*, 2016.

[13] A. Pfeffer, "Figaro: An object-oriented probabilistic programming language," *Charles River Analytics Technical Report*, vol. 137, p. 96, 2009.

[14] J. Salvatier, T. V. Wiecki, and C. Fonnesbeck, "Probabilistic programming in python using PyMC3," *PeerJ Computer Science*, vol. 2, p. e55, apr 2016.

[15] P. Lucas, "lumen." `https://github.com/lumen-org/lumen`, 2016.

[16] P. Lucas, "modelbase." `https://github.com/lumen-org/modelbase`, 2016.

[17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[18] S. Sinharay and H. S. Stern, "Posterior predictive model checking in hierarchical models," *Journal of Statistical Planning and Inference*, vol. 111, no. 1-2, pp. 209–221, 2003.

[19] D. B. Rubin, "Estimation in parallel randomized experiments," *Journal of Educational Statistics*, vol. 6, no. 4, pp. 377–401, 1981.

Appendices

A Rules of Probabilistic Inference

When working with Bayesian models it will be necessary to transform conditional, marginal and joint distributions into one another. There are three rules of probabilistic inference which achieve this: The chain rule, the total probability rule and the Bayes' rule. The following explanations are taken from Pfeffer [10].

Chain rule The chain rule is used to calculate a joint probability distribution of several variables from local conditional probability distributions of these variables:

$$P(X_1, X_2, ...X_n) = P(X_1)P(X_2|X_1)P(X_3|X_1, X_2)...P(X_n|X_1, X_2, ...X_{n-1})) \tag{21}$$

Total probability rule The total probability rule calculates the probability distribution over a subset of variables, also called a marginal distribution, by summing out all the other variables, that is by summing the probability distributions for each combination of values of these variables:

$$P(\boldsymbol{X}|\boldsymbol{Z}) = \sum_{\boldsymbol{y}} P(\boldsymbol{X}, \boldsymbol{Y} = \boldsymbol{y}|\boldsymbol{Z}) \tag{22}$$

Bayes' rule Bayes' rule calculates the probability of a cause, given an effect, by using the prior probability of the cause and the probability of the effect given the cause.

$$P(X|Y) = (P(Y|X) * P(X))/P(Y) \tag{23}$$