Bayesian Logistic Regression:

Theoretical foundations and application to real use cases employing specific probabilistic programming (PyMC, PyStan and Tensorflow Probability)

Bayesian Statistics and Probabilistic Programming Course Project

Núria Camí, Ana de Garay, Claudia Herron

November 13, 2022

Contents

1 Introduction

1.1 Motivation and goals

This project consists of doing an exhaustive review of the Logistic Regression function from a Bayesian perspective. In summary, apart from giving a complete explanation of the main concepts related with this algorithm, our idea is to perform several Bayesian experiments with three different programming languages (PyMC, PyStan and Tensorflow Probability) in order to end up comparing them. Therefore, our main goals have been:

- Study the Logistic Regression and the Logistic Function from a theoretical point of view and introduce its main properties.
- Present an appropriate Bayesian background related to the mentioned topic.
- Perform some first experiments on synthetic data with PyMC (Binary univariate and bivariate classification, and Softmax regression)
- Repeat the experiment of bivariate classification on real data with PyMC, PyStan and Tensorflow Probability
- Compare PyMC, PyStan and Tensorflow Probability regarding the convergence, autocorrelation, effective sample size, execution time and performance.

The code can be found in the following repository BSPP (https://github.com/claudia-hm/BSPP) under the folder *Project*.

2 Background

2.1 Logistic regression

The **logistic regression** is a powerful supervised Machine Learning algorithm used to assign observations to a discrete set of classes. Despite its name containing the term of *regression*, it is used to solve classification problems rather than regression ones. In fact, the logistic regression model is no longer than an extension of the linear regression but for classification problems [4].

The **logistic function** (also known as **sigmoid function**), defined below, can only take values between 0 and 1. This is the key property of this function from the classification perspective.

$$\text{Logistic Function: } logistic(z) = \frac{1}{1+e^{-z}} \qquad \forall z \in \mathbb{R},\ logistic(z) \in [0,1]$$

For simplicity, many times it is referred to z as the standard linear model $\alpha+\beta x$. However, we can actually use more than one independent variable. For example, we could consider expressions like $z = \alpha+\beta_0 x_0+\beta_1 x_1$, or even $z = \alpha + \beta x_0 x_1$. This is known as **multiple logistic regression**.

Another key concept worth mentioning is the **decision boundary**, which is the value of x_i for which $logistic(z) = \theta$, where θ is a fixed number between 0 and 1. For example, by considering $\theta = 0.5$ in the uni-dimensional case, if we solve $0.5 = \alpha + \beta \cdot x_i$, we obtain:

$$x_i = -\frac{\alpha}{\beta}.$$

In this case, we are giving the same 'importance' to missclassify a sample of class 0 as from class 1. However, there is not any need of having a symmetrical misclassification and it can be easily changed by modifying the value of θ.

Consequently, the decision boundary for the two-dimensional case with $\theta = 0.5$ would be computed as:

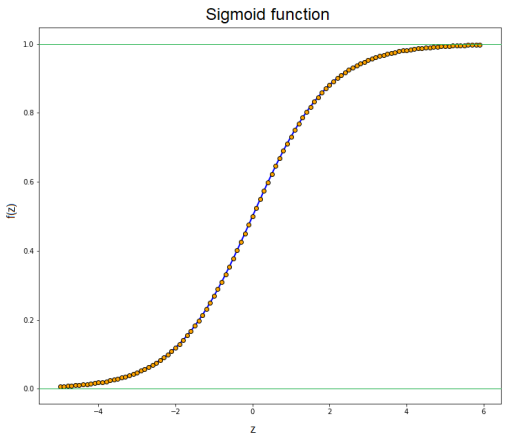

Figure 1: Sigmoid function.

$$x_1 = -\frac{\alpha}{\beta_1} + \left(-\frac{\beta_0}{\beta_1} x_0\right)$$

Note that the decision boundary for the uni-dimensional case was a scalar (a point) and now we got a line. With a point we can perfectly split the uni-dimensional data into two groups, and the same happens with a line in two-dimensional data, with a plane in three-dimensional data, and so on.

Another important function that we will use to extend logistic regression to the multiclass problem is the **softmax function**, defined as follows:

$$softmax_i(\mu) = \frac{exp(\mu_i)}{\sum exp(\mu_k)}$$

Two key properties of this function are: the output is always positive and the sum of all the components of the softmax adds up to 1, so we can interpret them as probabilities. In fact, the softmax function reduces to the logistic for two classes.

2.2 Generalized linear models: from regression to classification

The logistic regression model is an example of a broad class of models known as generalized linear models (GLM), which are extensions of the linear model. Particularly, logistic regression extends linear regression to classify instances of data instead of doing a regression.

A simplest example to explain how we can extend the linear model is using the coin-flipping problem. Usually, a **Bernoulli distribution** is used as likelihood, which expects a parameter θ having values limited to the [0,1] range. A Beta distribution for θ could be already valid for this approach. However, imagine that we have a linear model with parameters α and β ($\alpha + \beta x$). This model will potentially be defined in the real line $\mathbb{R}$, so, can we manage to use it for our parameter? Well, by applying the logistic function, we actually can:

$$y \sim Bern(\theta), \quad \text{where} \quad \theta = logistic(\alpha + \beta x) \in [0, 1]$$

This is how we have effectively transformed a linear regression model into a classification model. In fact, what we have indirectly done is considering a known function f (in our case, the logistic function) that do some kind of transformation to the linear model, like this:

$$\mu = f(\alpha + \beta x)$$

In GLM vocabulary, this function f is known as the **inverse link function**.

Finally, we can expand the idea of logistic regression to multiple classes thanks to the softmax function. In this scenario, we replace the Bernoulli distribution by the **categorical distribution**. This distribution describes the possible outcomes of a random variable that can take on one of K possible values, with the probability of each category separately specified [8]. Equivalently to the relationship between Bernoulli distribution and Binomial distribution, the categorical distribution is a special case of the multinomial distribution [4].

2.3 Frequentist Logistic Regression

In the frequentist view of Logistic regression, data is sampled from a fixed distribution defined by some *true* but unknown parameters. The main goal is to obtain point estimates of the parameters, rather than the full posterior distribution. To do so, frequentist techniques apply Maximum likelihood estimation (MLE) to maximize the log-likelihood, or minimize the negative log likelihood (loss function) [1]. Within this setting, we consider the classical Machine Learning scenario, where we can find 3 key elements: a class of mathematical models, a cost function describing the desired properties of the solution and a learning algorithm to minimize the cost.

Firstly, we need a mathematical model that describes the probability of classifying instances of data. Let's consider the binary univariate case, where our dataset $\mathcal{D}$, is a collection of real values, $x_i \in \mathbb{R}$, and labels $y \in \{0, 1\}$ such that $\mathcal{D} = (x_i, y_i)$. The logistic function is a perfect fit, as it can be seen as a mapping function from the space of real numbers to the interval [0,1]. Therefore, we can model the probability of obtaining a positive label (we could alternatively consider the negative label), given the data, with the sigmoid function:

$$p(y = 1) = \frac{1}{1 + e^{-\alpha x + \beta}}$$

Secondly, we define our loss or cost function where we specify the penalties for misclassifying instances. The cost function measures the discrepancies between the true label in our data, y_i, and the prediction for the same instance, $\hat{y_i}$. In classification, it is common to use the **cross-entropy** loss or **log-loss**, which has the following mathematical formulation:

$$cost(y_i, \hat{y_i}) = -y_i \log \hat{y_i} - (1 - y_i) \log 1 - \hat{y_i}$$

If we want to compute the total cost of our dataset, we can simply add up all the individual costs per instance. As a result, our loss function $\mathcal{L}$ for our dataset $\mathcal{D} = (x_i, y_i)$ is:

$$\mathcal{L}(\mathcal{D}) = \sum_i -y_i \log \hat{y_i} - (1 - y_i) \log 1 - \hat{y_i}$$

As we are considering the frequentist approach, we are going to assume that there exist some *true* parameters α an β for our logistic function that minimize the loss. We are in front of an optimization problem without closed form solution, so to solve it we can use an iterative algorithm such as **gradient descent** to find the parameters $\hat{\alpha}$ and $\hat{\beta}$ that best fit our data according to our loss function. To do so, we need to compute the derivative of the loss function with respect to our parameters [6].

Let's define

$$z = \alpha x + \beta$$

$$\hat{y} = \sigma(z)$$

Then, using the chain rule

$$\frac{\partial \mathcal{L}}{\partial \alpha} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial \alpha}$$

... and computing the partial derivatives...

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}}(-y \log \hat{y} - (1 - y) \log 1 - \hat{y}) = -y \frac{1}{\hat{y}} - (-1) \frac{1 - y}{1 - \hat{y}}$$

$$\frac{\partial \hat{y}}{\partial z} = \hat{y}(1 - \hat{y})$$

$$\frac{\partial z}{\partial \alpha} = x$$

... we obtain the derivative with respect to α.

$$\frac{\partial \mathcal{L}}{\partial \alpha} = (-y\frac{1}{\hat{y}} - (-1)\frac{1-y}{1-\hat{y}}) \cdot \hat{y}(1-\hat{y}) \cdot x = (\hat{y} - y) \cdot x$$

We do the same for β and we obtain:

$$\frac{\partial \mathcal{L}}{\partial \beta} = (-y\frac{1}{\hat{y}} - (-1)\frac{1-y}{1-\hat{y}}) \cdot \hat{y}(1-\hat{y}) \cdot 1 = \hat{y} - y$$

By iteratively updating the estimates of α and β we will arrive at an approximation of the global minimum of our loss function. This is the solution of the classification problem using Logistic Regression via the frequentist approach.

2.4 Bayesian Logistic Regression

As shown on section 2.2, logistic regression is an extension of linear regression. From a Bayesian perspective, a linear regression model can be expressed as,

$$y \sim N(\mu = \alpha + \beta x, \sigma = \epsilon)$$

.i.e., a normal distribution (does not need to be a normal necessarily, we employ it here for the sake of simplicity), with mean $\alpha + \beta x$ and standard deviation ϵ. From this probabilistic approach, given that values α, β and ϵ are, in principle, unknown for us, we can set prior distributions to each of them. When prior distributions have no population basis, they can be difficult to construct, and therefore should play a minimal role in the posterior distribution. Such distributions are sometimes called 'reference prior distributions', and the prior density is described as vague, flat, diffuse or non-informative. The rationale for using noninformative prior distributions is often said to be 'to let the data speak for themselves,' so that inferences are unaffected by information external to the current data. A related idea is the weakly informative prior distribution, which contains some information—enough to 'regularize' the posterior distribution, that is, to keep it roughly within reasonable bounds—but without attempting to fully capture one's scientific knowledge about the underlying parameter (ref:BDA3).

A Bayesian model as the above defined would have a posterior distribution such as the following:

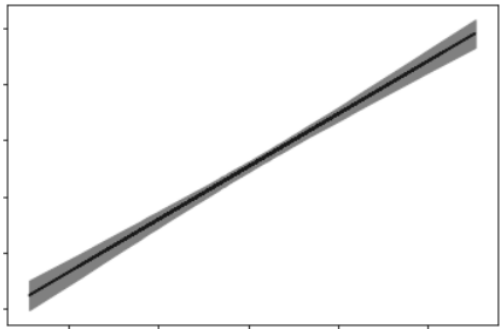

Figure 2: Posterior linear regression with 95% HDP

Where we can see the average line that fits the data together with the average mean values of α and β. The posterior uncertainty is reflected with a semitransparent band that illustrates the 95% Highest Posterior Density (HPD) interval of μ. A $100(1-\gamma)$ HPD region for μ is a subset $\mathcal{C}$ such that

$$\mathcal{C} = \{\mu : p(\mu|x) \geq k\}$$

where k is the largest number such that

$$\int_{\mathcal{C}} p(\mu|x)d\mu = 1 - \gamma$$

(ref:https://people.stat.sc.edu/hitchcock/stat535slidesday3.pdf)

Hence, applying the sigmoid function to this model lead us to the logistic regression, that can be defined as follows:

$$\theta = logistic(\alpha + \beta X)$$

$$y \sim Bern(\theta)$$

where the main difference with the simple linear regression is the use of a Bernoulli distribution instead of a Gaussian distribution (or Student's t-distribution) and the use of the logistic function that allows us to generate a parameter in the range [0, 1], suitable for feeding the Bernoulli distribution. Observe that variable θ here is deterministic, in the sense that it is fully determined by its arguments, even if the arguments are stochastic. Notice that logistic regression is actually a regression since we are regressing the probability that a data point belongs to class 1, given a feature. It should be noticed that we are observing only a dichotomous variable and inferring the continuous probability.

2.5 Probabilistic programming: PyMC, PyStan and Tensorflow probability

The main idea behind Bayesian models is very simple: we have some initial, a priori, knowledge and after observing some evidence (data), we combine all the information with Bayes' formula to obtain posterior knowledge. However, many times there is no simple analytical expression for the posterior. Thanks to the arrival of the computational era and the development of probabilistic programming languages, it is possible to compute the posterior for nearly all Bayesian models. These languages allow the user to separate the model creation and the inference with just a few lines of code.

In this project, we decided to work with Python, and three of the state-of-the-art probabilistic programming libraries: PyMC, PyStan and Tensorflow probability. PyMC allows the user to build Bayesian models with a simple and intuitive syntax and fit them using Markov chain Monte Carlo (MCMC) methods. This library includes different sampling algorithms such as Hamiltonian Montecarlo, No-U-Turn Sampler (NUTS) or Variational inference. PyStan offers an interface to Stan, a platform for statistical modeling and high-performance statistical computation. The default sampling algorithm is NUTS. The third library is TensorFlow Probability (TFP). The main advantage of this package is that it allows to combine probabilistic models and deep learning on high performance hardware, such as GPUs and TPUs. From these libraries, we selected PyMC to develop the core of the project as it seem to us the most user-friendly. Then, we built the final model also with PyStan and TFP and benchmark different performance metrics.

3 Methodology

In this section we will develop a series of classification experiments taking advantage of Bayesian logistic regression. We will perform such experiments both with synthetic and real data. We will do so with the support of PyMC, where the computationally demanding parts are written using NumPy and Theano [3]. Theano is a Python library originally developed for deep learning that allows us to define, optimize, and evaluate mathematical expressions involving multidimensional arrays efficiently. The main reason PyMC uses Theano is because some of the sampling methods, like NUTS, need gradients to be computed and Theano can deal efficiently with automatic differentiation. Also, Theano compiles Python code to C code, and hence PyMC is really fast [4]. The last version of PyMC at the moment of writing this report is 4.0.1, which is the one we have used to perform our experiments.

3.1 Synthetic data

3.1.1 Binary univariate classification

We start by the most simple case, a two-class classification , i.e., $y \in \{0, 1\}$ with just one feature, i.e., $x \in \mathbb{R}$. Therefore in this case, the standard linear model to which we will apply the logistic function will be

$$\alpha + \beta x, \quad \alpha, \beta, x \in \mathbb{R}$$

We simulated two clusters of size 50 each following a normal distribution, the first one centered on 2.5 and the second one centered on 4, with a standard deviation of 1 for both of them.

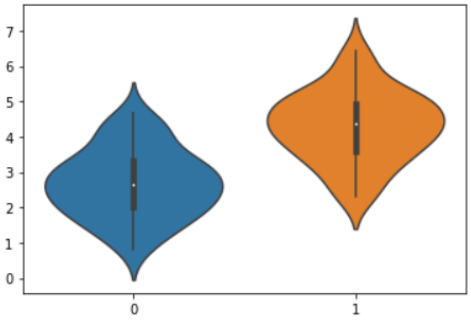

Figure 3: Binary univariate classification: Violin plot of the synthetic data

Once we have the data, we need to specify the model, that is, our prior distribution and likelihood. As the likelihood, we will choose a Bernoulli ($p = \theta$) distribution, and for the priors we select $N(\mu = 0, \sigma = 10)$ for both α and β. Now, we will give a detailed description of how to define a probabilistic model with PyMC, including the code, shown on Figure 4 and the step by step explanation. The model is defined with PyMC as follows:

- `line 1`: creates a container for our model, using the `with` statement to indicate that everything inside the with block points to the same model, `model_0`.
- `lines 2-4`: specifies the priors for α and β, and μ is defined as the line $\alpha + x_0\beta$, being x_0 our previously generated data points.
- `lines 5-6`: θ is deterministically defined (fully determined by its arguments) as the sigma function applied to μ, and the classification decision boundary is simply defined as $-\frac{\alpha}{\beta}$ (i.e., $x : logistic(\alpha + x\beta) = 0.5$).
- `line 7`: specifies the likelihood following the same syntax as for the prior, except that we pass the data using the observed argument. The data can be a Python list, a NumPy array or a Pandas DataFrame. Here we pass y_0 as the observed data, i.e., the corresponding labels in $\{0, 1\}$ for data points in x_0.
- `line 8`: calls `find_MAP`; this function calls optimization routines provided by SciPy library to return the Maximum a Posteriori (MAP).
- `line 9`: provides the sampling method; in this case we use `NUTS()`, a sampler for continuous variables based on Hamiltonian mechanics (ref: https://docs.pymc.io/en/latest/api/generated/pymc.NUTS.html).
- `line 10`: performs the inference with `sample()` which takes as arguments the number of samples we want to draw (that we set as 10000 in this case), the sampling method and the starting point. We can also enter the number of chains to produce. In this case, we will produce the default number, i.e., 4 for the computer executing this code (Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz).

```python
with pm.Model() as model_0:
    alpha = pm.Normal('alpha', mu=0, sigma=10)
    beta = pm.Normal('beta', mu=0, sigma=10, shape=K)
    mu = alpha + pm.math.dot(x_0, beta)
    theta = pm.Deterministic('theta', 1 / (1 + pm.math.exp(-mu)))
    bd = pm.Deterministic('bd', -alpha/beta)
    yl = pm.Bernoulli('yl', p=theta, observed=y_0)
    start = pm.find_MAP()
    step = pm.NUTS()
    trace_0 = pm.sample(8000, step, start)
```

Figure 4: Python code defining binary univariate logistic regression model

We therefore produced 4 sampling chains that will determine our posterior distribution, according to the defined priors and likelihood. PyMC allows us to run a model several times in parallel and thus get a parallel chain for the same parameter. As stated above, this is specified with the argument `chains` in the `sample` function. Since each chain is independent of the others and each chain should be a good sample, they should look similar to each other. Besides checking for convergence, these parallel chains can be used also for inference; instead of discarding the extra chains, we can combine them to increase the sample size.

In order to check the results, we employ the `plot_trace` method from `arviz` library. We will get two plots for each unobserved parameter, as we can see on Figure 5. On the left, we get a plot with the kernel density estimation (KDE), a technique that allows us to create a smooth curve given a set of data. This can be useful if you want to visualize just the shape of data, as a kind of continuous replacement for the discrete histogram [2]. The 4 lines that can be observed correspond to each of the chains of our sampling. On the right, we get the individual sampled values at each step during the sampling, that should look like white noise.

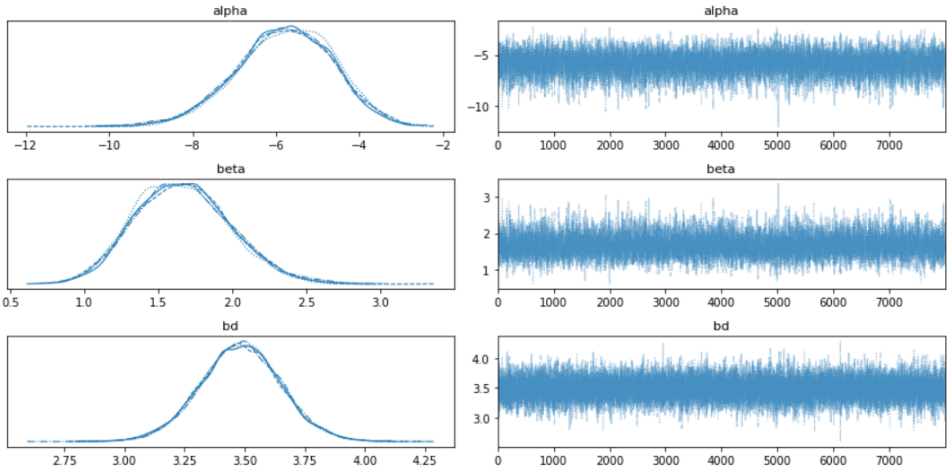

Figure 5: Binary univariate classification: Posterior plots

Finally, we plot the data together with the fitted logistic curve, as shown on Figure 6 . The fitted sigmoid function is the blue line and the decision boundary is the red line, with their corresponding 95% HDIs. From this plot it is easy to interpret the classification for each data point; points at the right side of the decision boundary will be classified as class 1, and those at the left side will be classified as class 0. Black dots are positioned along the y-axis according to their real class.

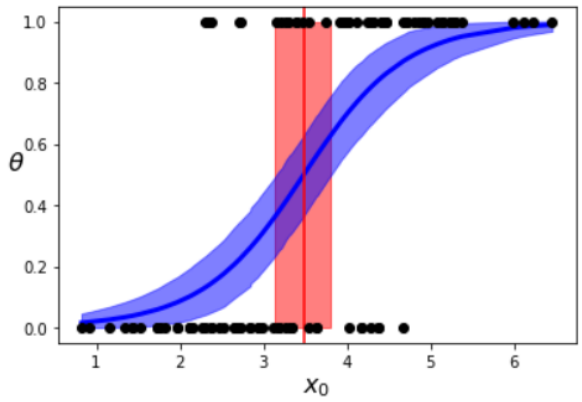

Figure 6: Binary univariate classification: Sigmoid function and decision boundary

3.1.2 Binary bivariate classification

Now let us go a step forward and include multiple features instead of just one, i.e., we will consider $y \in \{0, 1\}$ and $x \in \mathbb{R}^N$, with $N > 1$. For the sake of simplicity and to be able to produce clearer plots, we chose $N = 2$. Therefore we will consider $x = (x_0, x_1)$ and $\beta = (\beta_0, \beta_1)$. Therefore in this case, the standard linear model to which we will apply the logistic function will be

$$\alpha + \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix} \begin{pmatrix} x_0 & x_1 \end{pmatrix}, \quad \alpha \in \mathbb{R}, \quad \beta, x \in \mathbb{R}^2$$

Again, we will simulate two clusters of size 50 each following a normal distribution, the first one centered on [2.5, 2.5] and the second one centered on [4, 4], with a standard deviation of [1, 1] for both of them. The plot is shown on Figure 7. The model definition is quite similar to the one on the previous experiment, and further details can be checked on notebook "Logistic regression with pymc - Synthetic Data.ipynb". The obtained posterior plots are shown on Figure 8.

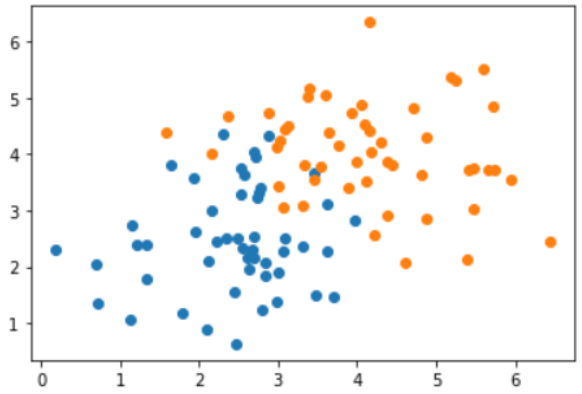

Figure 7: Binary bivariate classification: Data

Notice that now in the case of the β plots, we are getting 4 lines for each of β_0 and β_1, and for the decision boundary now we get 100 sets of curves, each for a data point. Finally we can plot the decision boundary and its HDI interval, displayed on Figure 9.

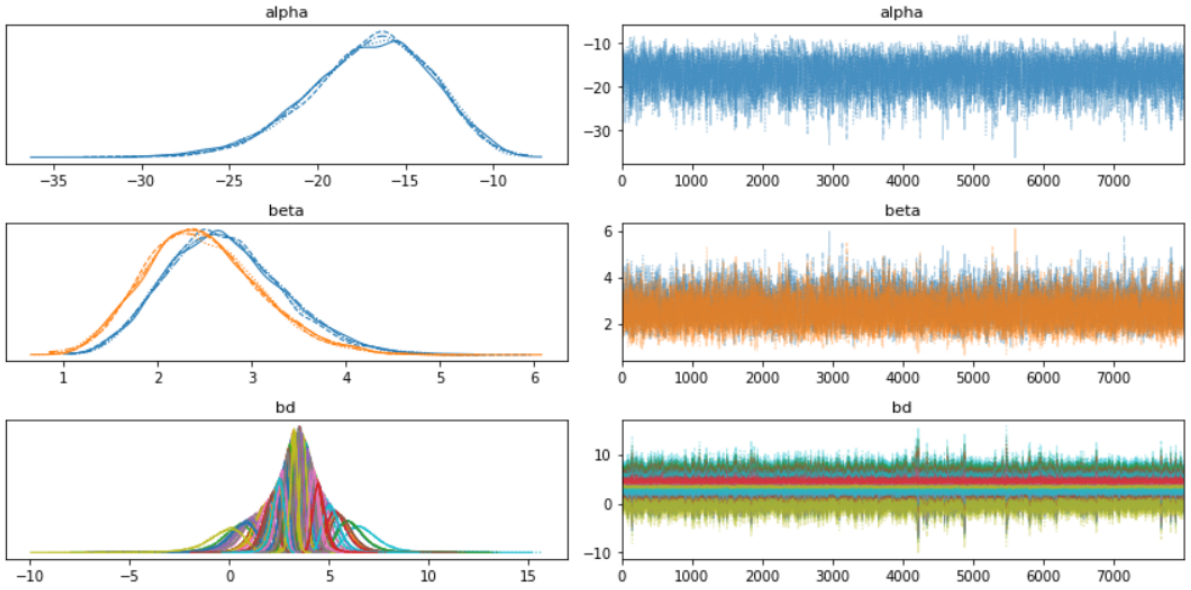

Figure 8: Binary bivariate classification: Posterior plots

3.1.3 Softmax regression or multinomial logistic regression

Finally, let us perform a non-binary classification. We will consider three different classes, i.e., $y \in \{0, 1, 2\}$ and also two different features. Therefore in this case, the standard linear model to which we will apply the logistic function will be

$$\begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \end{pmatrix} + \begin{pmatrix} \beta_0^0 \\ \beta_1^0 \\ \beta_2^0 \end{pmatrix} x_0 + \begin{pmatrix} \beta_0^1 \\ \beta_1^1 \\ \beta_2^1 \end{pmatrix} x_1, \quad \alpha, \beta^0, \beta^1 \in \mathbb{R}^3$$

We can plot a projection in $\mathbb{R}^2$ of our data, as shown on Figure 11

Some changes are needed to define our model. Besides the corresponding changes in the shape parameter when defining the priors of α and β, we have employed the softmax function as a generalization of the logistic function to multiple dimensions. Again, further details can be checked on notebook "Logistic regression with pymc - Synthetic Data.ipynb". Finally, we observe the results of the sampling on Figure . Notice that now we have 3 different sets of lines for α (as we are in a 3-class scenario), and 3 different sets of lines for each β^0 and β^1. We will not plot the decision boundary, given that this multi-dimensional case would be more intrincated and unintuitive than the previous ones.

3.2 Real use case

Now we will apply logistic regression to a real dataset. The chosen dataset is the Palmer Penguins dataset, very well suited for exploration & visualization. Data was collected and made available by Dr. Kristen Gorman and the Palmer Station, Antarctica LTER, a member of the Long Term Ecological Research Network. It contains data for 344 penguins. There are 3 different species of penguins in this dataset, collected from 3 islands in the Palmer Archipelago, Antarctica [5]. We perform a quick Exploratory Data Analysis that can be checked on notebook "Logistic regression with pymc - Real data.ipynb". The species variable will be our dependent variable, and we will consider four different independent variables: `bill_length_mm`, `bill_depth_mm`, `flipper_length_mm`, `body_mass_g`. We can take a glance at the relations in our selected subset in Figure 12.

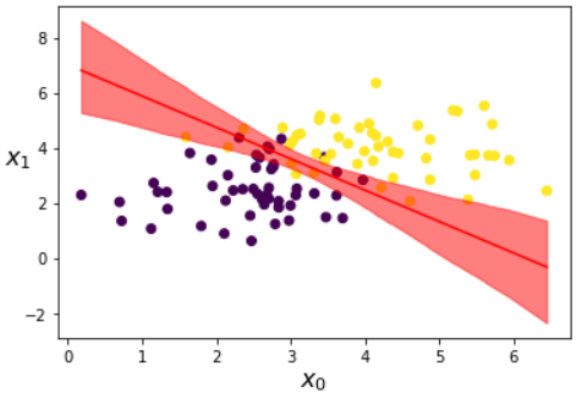

Figure 9: Binary bivariate classification: Decision boundary

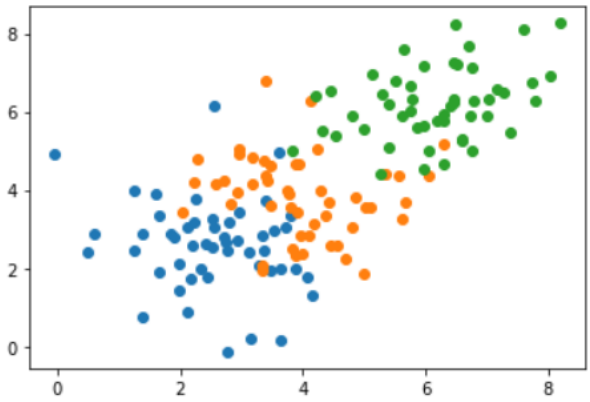

Figure 10: Multinomial logistic regression: Data projection

3.2.1 Binary univariate classification

Again, we start by the most simple case, a two-class classification. We will just consider species Adelie and Chinstrap, i.e., $y \in \{0, 1\}$. Here we will employ just independent variable, `flipper_length_mm`, i.e., $x \in \mathbb{R}$. As we can appreciate on Figure 12, the two classes are not separable when considering the aforementioned feature. Now that we have the data, we need to specify the model, that is, our prior distribution and likelihood. As the likelihood, we will choose a Bernoulli ($p = \theta$) distribution, and for the priors we select $\alpha \sim N(\mu = 0, \sigma = 10)$ and $\beta \sim N(\mu = 0, \sigma = 1)$. We obtain both the posterior plots for α, β and the decision boundary (Figure 13), and the data together with the fitted logistic curve (Figure 14).

As we saw above, due to the non-separability of the clusters, the linear decision boundary is not able to classify samples without incurring in missclassifications.

3.2.2 Binary bivariate classification

Now let us go a step forward and include multiple features instead of just one, i.e., $x \in \mathbb{R}^N$, with $N > 1$. For the sake of simplicity and to be able to produce clearer plots, we chose $N = 2$. We chose independent variables `bill_length_mm` and `flipper_length_mm`. Again define our model as in the homonym experiment with the synthetic dataset, and with priors $\alpha \sim N(\mu = 0, \sigma = 10)$ and $\beta \sim N(\mu = 0, \sigma = 1)$. We obtain both the posterior plots for α, β and the decision boundary (Figure 15), and the data separated by the decision boundary (Figure 16).

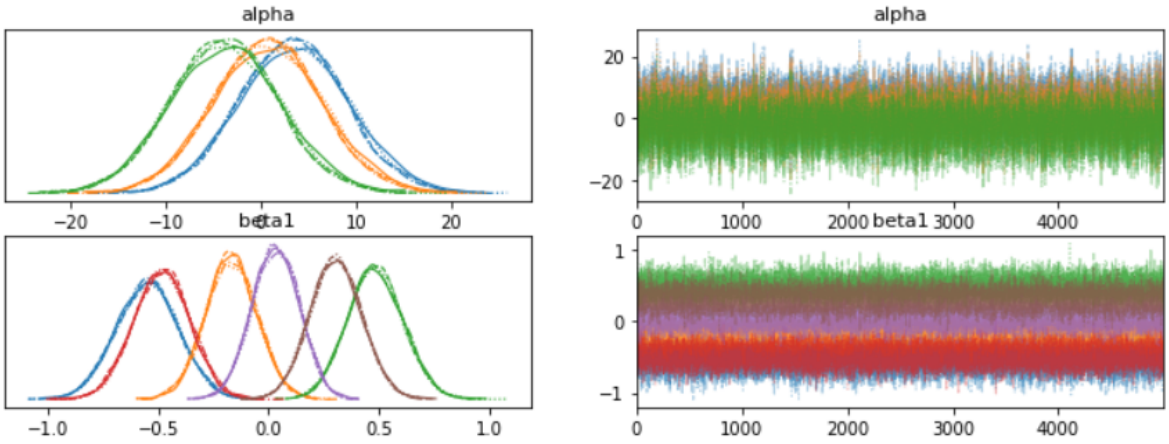

Figure 11: Multinomial logistic regression: Posterior plots

Notice that in the case of the β plots shown in Figure 15, we are getting a set of 4 lines (as we have selected 4 chains) for each of β_0 and β_1, and for the decision boundary now we get 100 sets of curves, each for a data point. Again, as we can see in Figure 16, we are in front of a non-separable case and therefore our linear classifier is not able to display a great performance.

3.2.3 Softmax regression or multinomial logistic regression

Finally, for the dependent variable we use the three classes (Adelie, Gentoo and Chinstrap), and the four considered dependent variables. We are also standardizing the data, since this will help the sampler to run more efficiently. Therefore in this case, the standard linear model to which we will apply the logistic function will be

$$\begin{pmatrix}\alpha_0\\ \alpha_1\\ \alpha_2\end{pmatrix} + \begin{pmatrix}\beta_0^0\\ \beta_1^0\\ \beta_2^0\end{pmatrix} x_0 + \begin{pmatrix}\beta_0^1\\ \beta_1^1\\ \beta_2^1\end{pmatrix} x_1 + \begin{pmatrix}\beta_0^2\\ \beta_1^2\\ \beta_2^2\end{pmatrix} x_2 + \begin{pmatrix}\beta_0^3\\ \beta_1^3\\ \beta_2^3\end{pmatrix} x_3, \quad \alpha, \beta^0, \beta^1, \beta^2, \beta^3 \in \mathbb{R}^3$$

We proceed to define our model as we did with the synthetic dataset. We chose as priors $\alpha \sim N(\mu = 0, \sigma = 10)$ and $\beta \sim N(\mu = 0, \sigma = 2)$. Finally we can plot our sampling results for α and β posteriors. It can be noticed in Figure 17 that now in the case of the β plots, we are getting 12 lines for each of $\beta_i^j \quad i \in \{0, 1, 2\}, j \in \{0, 1, 2, 3\}$. Again, We will not plot the decision boundary, given that this multi-dimensional case would be more intrincated and unintuitive than the previous ones.

4 Experiments

In this section we will reproduce the binary multivariate classification scenario that we saw with the real dataset, but now we will employ PyStan for benchmarking. In particular, the chosen subset to perform the experiments would be the one with penguins from Adelie and Chinstrap species, considering all the independent variables (a total of four) introduced above. We will focus on benchmarking some of the aspects of the models obtained with these three different probabilistic programming tools. Specifically, we will compare:

- Convergence
- Autocorrelation

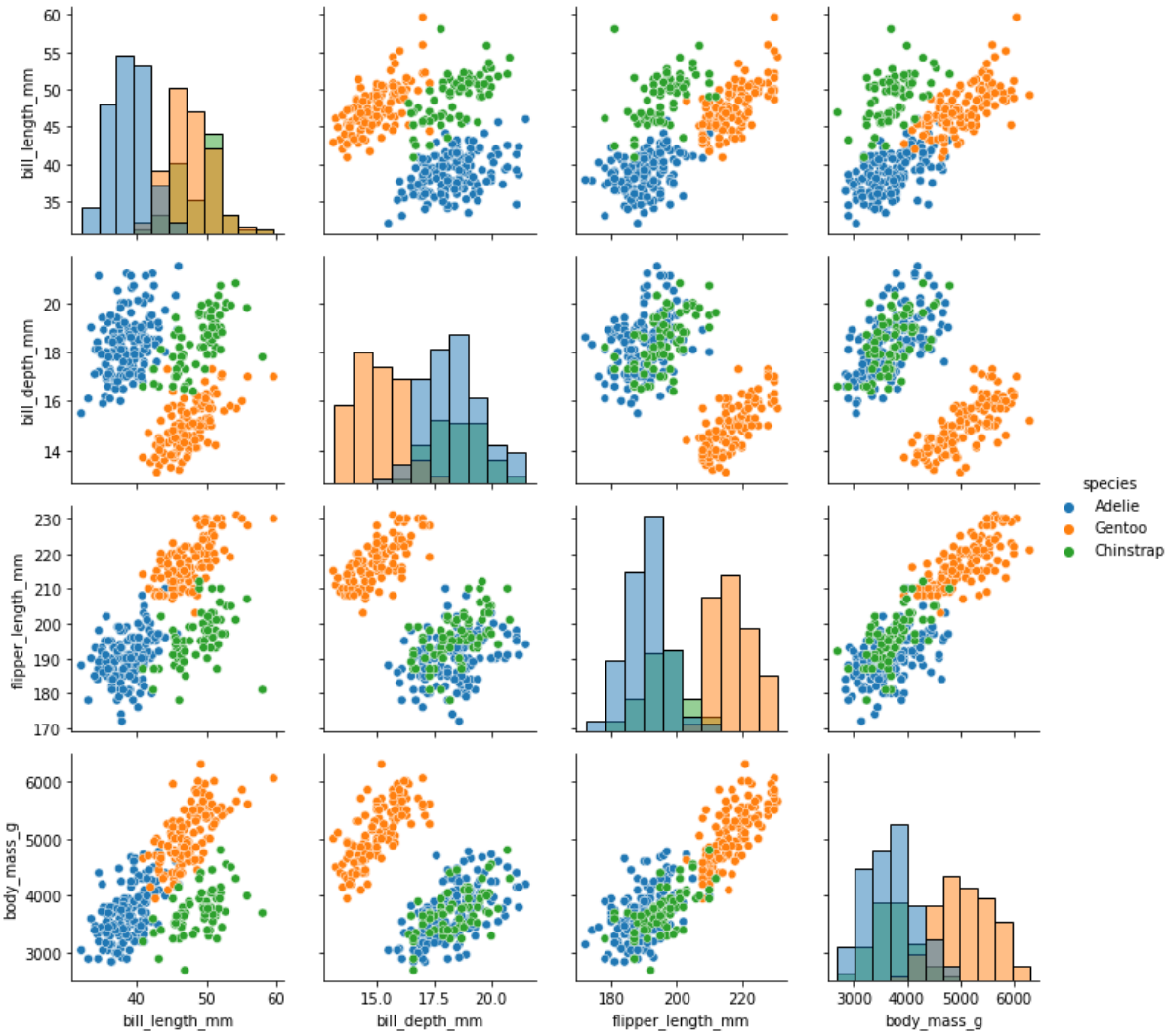

Figure 12: Scatter plot matrix of Penguins dataset

- Effective sample size
- Execution time
- Performance

For the sampling, we selected 2 chains and 2000 samples per chain.

Moreover, we include a subsection with some of the installation and technical issues we have encountered when dealing with the mentioned libraries.

4.1 Installation and technical issues

In this project we had a considerable amount of technical issues related to the usage of the above mentioned libraries. In the case of PyMC, the main issue was that we started our project with version 3.x.x of the package. The 6th June, a new version 4.0.0 was launched, and it included changes in the syntax that we had to resolve. Moreover, most of the code examples were in the previous version so we had to adapt all the code that we found online.

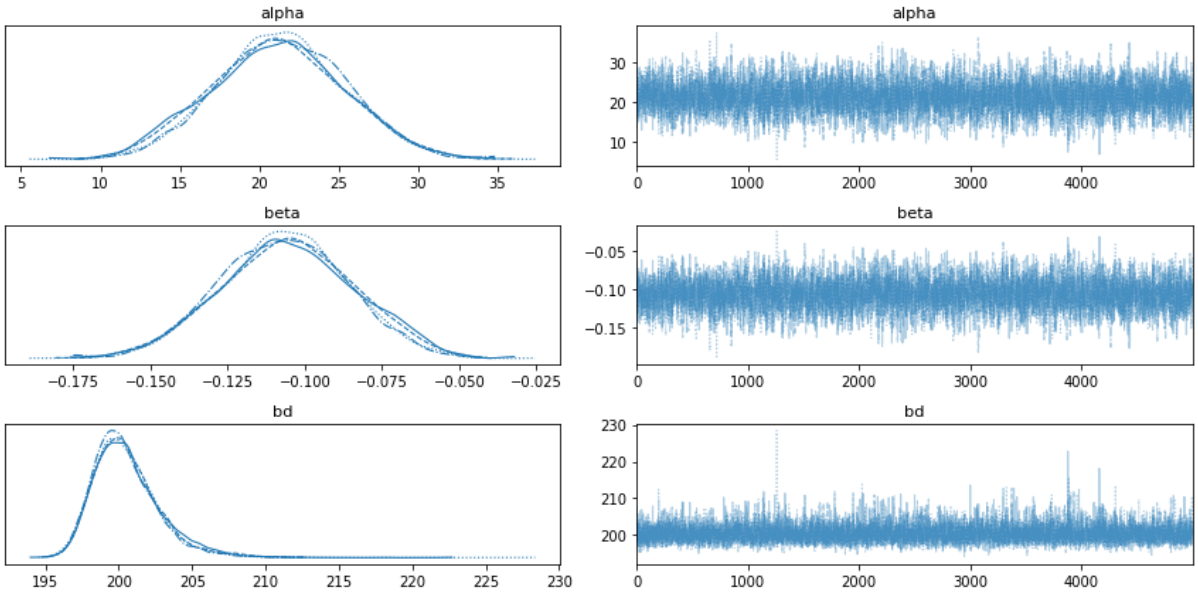

Figure 13: Binary univariate classification: Posterior plots

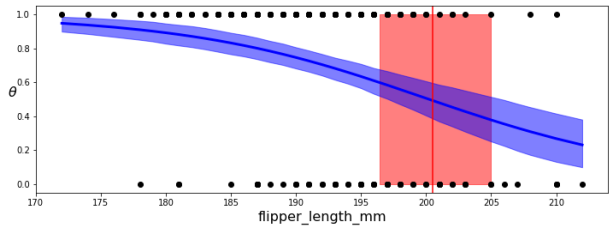

Figure 14: Binary univariate classification: Sigmoid curve and decision boundary

In addition, also with PyMC, we encountered some convergence warnings that we were finally able to avoid. However, it is worth mentioning that when defining the model, it is easy to find warnings related to NUTS acceptance value if it is not close to 0.8 (default value). Given that this kind of warning was still present even though we explored a wide set of different sampling parameters, we tried to reach PyMC developers in order to solve the issue. Fortunately, it had been answered before here:

```
I think you can argue that this warning is not necessary, of all the warnings the sampler
can show at the moment this is clearly the least worrying.  (Just to be clear:  only if the
actual acceptance rate is higher than the target).  There really isn't a good reason why an
acceptance rate of 0.8 is always the value you want.  High acceptance rates mean we spend a
lot of time solving the hamiltonian very precisely, and at some point this just isn't worth
the effort.  It is quite common to increase the target acceptance rate to something like 0.9
or even 0.99 for more complicated posterior geometries.  So this warning means that we are
solving the ode even more accurately than we said we wanted, which isn't a problem in itself.
However, this also means something didn't go quite as planned, because during tuning we usually
expect to adapt the step size such that the acceptance rate matches the target.  This usually
just means we've spent a bit of time needlessly, but it could also indicate that maybe other
parts of the adaptation (the mass matrix) didn't quite work out, and that might actually be
a problem.  So to summarize:  don't worry too much about this one, but to be safe, maybe increase
the number of tuning steps a bit.
```

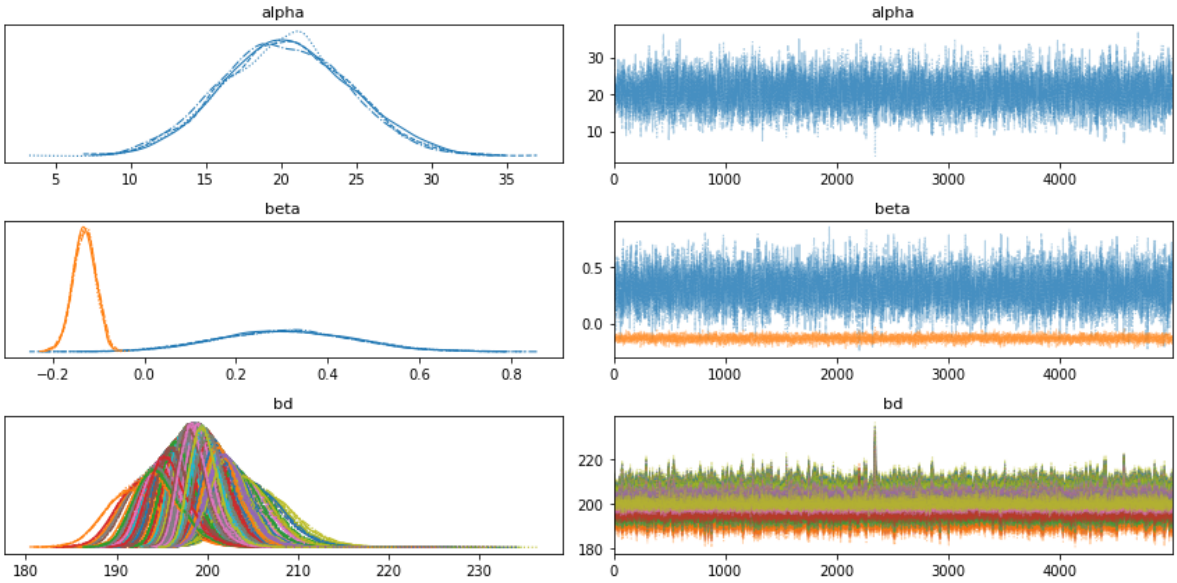

Figure 15: Binary bivariate classification: Posterior plots

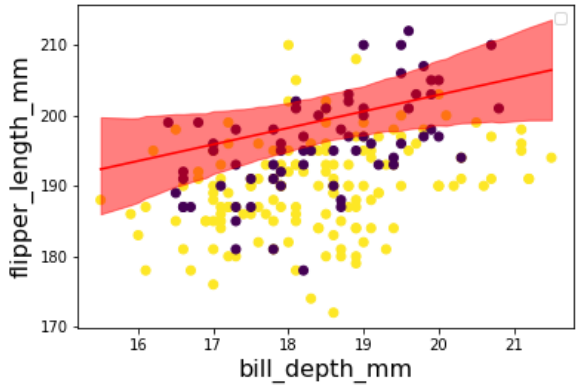

Figure 16: Binary bivariate classification: decision boundary

Therefore, we were confident preserving a configuration that showed an acceptance value higher than 0.8. However, as mentioned above (it can be checked on notebook "Logistic regression with pymc - Real data.ipynb"), we were finally able to avoid the warning.

Regarding the PyStan library, its installation was cumbersome and we decided to create virtual environments. First, we started using v2 of this package, as most of the information that we found online referred to this version. However, the execution times were irrationally big. Then, we decided to update our version to 3.4.1. Again, the syntax was different from version to version, so we had to adapt our code. We found that many people online criticized PyStan as difficult-to-use (for example see figure 18). The main problem of this library is that it gives errors with very little information (probably because it is an interface to Stan) and the documentation and code examples for the new version are scarce and austere. In fact, we did not manage to sample from the posterior distribution of the softmax regression model, (see notebook "Logistic regression with pystan - penguins.ipynb") as we obtained an error (`RuntimeError: Initialization failed.`) that could have many different sources.

Finally, the first obstacle we encountered with the TensorFlow probability library, was that most of the example notebooks we consulted used functions from TensorFlow version 1. As we were running our code

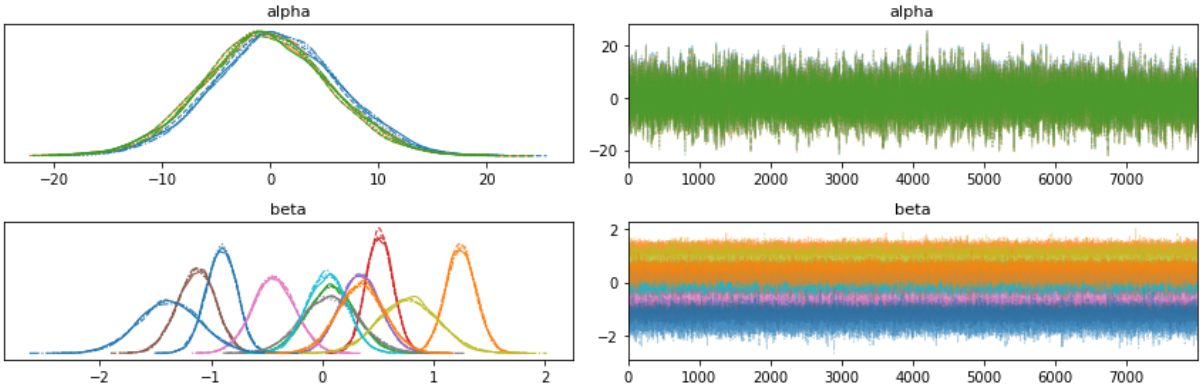

Figure 17: Multinomial logistic regression: posterior plots

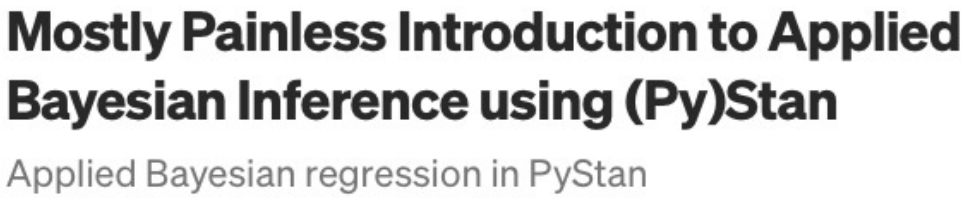

Figure 18: Example of PyStan critisizm

in v2 on Colab, the optimal solution we found was setting the instruction `compat.v1` between the function call. For example, instead of using `tf.get_variable`, calling it as `tf.compat.v1.get_variable`.

Apart from this, when trying to perform the different comparisons between the three programming languages, it was impossible for us to execute the three notebooks on the same device due to some package installation incompatibilities. That impeded us from doing a proper comparison between the three languages. For simplicity, as already indicated in the Methodology section, we only performed those comparisons between PyMC and PyStan. In any case, we have kept the experiments we have done with Tensorflow Probability in the work repository.

4.2 Sampling diagnostics

In this subsection we analyze the samplings performed with PyStan and PyMC over the real dataset with two classes and four features.

4.2.1 Convergence

A quantitative way to check for convergence is by using the Gelman-Rubin test. The GR diagnostic framework relies on m parallel MCMC chains, each run for n steps with starting points determined by a distribution that is over-dispersed relative to the target distribution. The GR statistic (denoted $\hat{R}$) is the square root of the ratio of two estimators for the target variance. In finite samples, the numerator overestimates this variance and the denominator underestimates it. Each estimator converges to the target variance, meaning that $\hat{R}$ converges to 1 as n increases. When $\hat{R}$ is sufficiently close to 1, the GR diagnostic declares convergence [7]. We have used `rhat` method from Arviz in order to do so and we have seen that for all our models we arrive to convergence.

4.2.2 Autocorrelation

An ideal sample will lack autocorrelation, that is, a value at one sampled point should be independent of the values at other sampled points. Therefore, we plot the correlogram of the samples obtained with the three different libraries. We have used `plot_autocorr` method from Arviz in order to do so. We omit the plots in this report because we have one per variable. The results are quite good, as these plots show low autocorrelation (see notebook *Experiments - PyMC, PyStan.ipynb)*

4.2.3 Effective sample size

A correlated sample provides less information than a sample of the same size without autocorrelation. Hence, given a sample of a certain size with a certain degree of autocorrelation we could try to estimate what will be the size of the sample with the same information without autocorrelation. We have used `ess` method from Arviz in order to do so.

	alpha	**beta0**	**beta1**	**beta2**	**beta3**
PyStan	3646.64	3269.49	3808.75	3777.26	3023.93
PyMC	3138.06	3193.95	3030.51	3215.40	2553.13

4.2.4 Execution time

We have measured the time taken for the sampling process with our two considered tools. The machine used is a MacBook Air 2015 with Processor 1.6GHz Dual-Core Intel Core i5 and 4 Gb or memory. For PyMC, the sampling took 51.09 seconds and for PyStan (without model building) 2.78 seconds. For the tensorflow probability model, we run it on Google Colab and it took approximately 10 minutes to run.

4.2.5 Performance

We will measure the accuracy for train and test (unseen) data in the three scenarios. We will feed our sampling process with just the train data. In order to calculate the accuracy, we will take mean values of the concatenation of sampled chains for α, β_0, β_1 and β_2, and simply calculate

$$p(y = 1) = \frac{1}{1 + e^{-\alpha + x_0\beta_0 + x_1\beta_1 + x_2\beta_2 + x_3\beta_3}}$$

for each observation (x_0, x_1, x_2, x_3, y), be it train or test.

We obtained the same accuracy in the two bayesian models: 73%. Also, we tested what was the default accuracy for the classical ML library `sklearn`, and we obtained the same.

5 Conclusions

In this project, we have analyzed the Bayesian model for Logistic regression and implemented it in three probabilistic programming Python libraries. After diagnosing the posterior sampling for a the Palmer Penguin dataset, we concluded that our three models were able to infere good results. In terms of performance, we obtained a quite good figure (73%) and in terms of execution time, PyStan was the fastest. However, from our point of view, PyMC is the most user-friendly library.

References

[1] Siwei Causevic. *Frequentist vs. bayesian approaches in Machine Learning*. Aug. 2020. URL: `https://towardsdatascience.com/frequentist-vs-bayesian-approaches-in-machine-learning-86ece21e820e`.

[2] By: Matthew Conlen. *Kernel Density Estimation*. URL: `https://mathisonian.github.io/kde/`.

[3] *Home*. URL: https://www.pymc.io/welcome.html.

[4] Osvaldo A. Martin, Ravin Kumar, and Junpeng Lao. *Bayesian Modeling and Computation in Python*. Boca Raton, Dec. 2021. ISBN: 978-0-367-89436-8.

[5] *Palmerpenguins R Data Package*. URL: https://allisonhorst.github.io/palmerpenguins/.

[6] Saishruthi Swaminathan. *Logistic regression - detailed overview*. Jan. 2019. URL: https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc.

[7] Dootika Vats and Christina Knudson. *Revisiting the Gelman-Rubin Diagnostic*. 2018. DOI: 10.48550/ARXIV.1812.09384. URL: https://arxiv.org/abs/1812.09384.

[8] Wikipedia. *Categorical distribution — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Categorical%20distribution&oldid=1027330818. [Online; accessed 27-June-2022]. 2022.