An introduction to data analysis using the PyMC3 probabilistic programming framework: A case study with Gaussian Mixture Modeling

Shi Xian Liew, Mohsen Afrasiabi, and Joseph L. Austerweil

August 28, 2019

Author Note
Correspondence concerning this article should be addressed to: Shi Xian Liew, 1202 West Johnson Street, Madison, WI 53706. E-mail: shixianliew@gmail.com

Abstract

Recent developments in modern probabilistic programming have offered users many practical tools of Bayesian data analysis. However, the adoption of such techniques by the general psychology community is still fairly limited. This tutorial aims to provide non-technicians with an accessible guide to `PyMC3`, a robust probabilistic programming language that allows for straightforward Bayesian data analysis. We focus on a series of increasingly complex Gaussian mixture models – building up from fitting basic univariate models to more complex multivariate models fit to real-world data. We also explore how `PyMC3` can be configured to obtain significant increases in computational speed by taking advantage of a machine's GPU, in addition to the conditions under which such acceleration can be expected. All example analyses are detailed with step-by-step instructions and corresponding Python code.

Keywords: probabilistic programming; Bayesian data analysis; Markov chain Monte Carlo; computational modeling; Gaussian mixture modeling

1 Introduction

Over the last decade, there has been a shift in the norms of what counts as rigorous research in psychological science. Although there are many reasons for the shift in norms, two main reasons include (1) failures to replicate pivotal studies (Open Science Collaboration, 2015) and (2) the publication of a particular article with a far-fetched conclusion, that extrasensory perception is real, which passed the methodological rigor at the time (Bem, 2011; see also Wagenmakers, Wetzels, Borsboom, & Van Der Maas, 2011). There have been a number of changes that have happened that are beyond the scope of this article, including large-scale multi-lab replications of previous and new studies (e.g. Ebersole et al., 2016; Open Science Collaboration, 2015; Frank et al., 2017), pre-registration of research studies (Nozek, Ebersole, DeHaven, & Mellor, 2018), and the collection of larger sample sizes whenever possible. In this article, we focus on aiding psychologists with conducting appropriate, sophisticated statistical analyses and how to implement these models using a modern tool: probabilistic programming with `PyMC3`.

One of the main culprits cited by researchers reforming psychological methodology is the widespread use of simple statistical models within the traditional frequentist null hypothesis testing framework (Wagenmakers et al., 2018). This traditional framework is blamed because of the many methods experimentalists can use to "cheat" (intentionally or unintentionally), such as "p-hacking" where participants are run until a hypothesis test is significant, "the file drawer effect" where studies without a significant test are not published and stay in a file drawer (or now a computer folder), skewing the reported results in published articles, and failure to correct for multiple comparisons (Nieuwenhuis, Forstmann, & Wagenmakers, 2011). Bayesian data analysis is largely robust to these issues (Lee & Wagenmakers, 2005; Wagenmakers, Lee, Lodewyckx, & Iverson, 2008). Combined with the advent of easy-to-use and free software to conduct standard analyses in the Bayesian hypothesis testing framework (e.g., JASP; JASP Team, 2018), Bayesian methods are increasingly being used by psychologists.

Although Bayesian data analysis has improved the methodology used in psychology, it is not a panacea to all statistical problems that can arise when analyzing complex research data. One part of Bayesian data analysis is that a model for the null hypothesis must always be explicitly specified. The usefulness of the analysis depends on the appropriateness of the null model for the researchers' question(s) and the properties of their data set. Some researchers view this as a weakness because ideally, one should want to analyze data with as few prior assumptions needing to be made as possible (a strength of frequentist nonparametric tests, albeit at the cost of lower statistical power). Other researchers view this as a strength (including the authors of this article) because null hypothesis testing never really avoided this issue – it was only hidden using asymptotic results based on assumptions about how data are generated, which are not realistic for data collected in psychological experiments. Further, proponents of this viewpoint would rather the null model be explicitly and thoughtfully defined rather than implicitly assumed, which can lead to possible misinterpretation of the analyses.

A major issue that arises in Bayesian data analysis is how to pick the appropriate probabilistic model for the data collected in a research study. For simple experimental designs, probabilistic models with a few parameters can be used, which allow for Bayesian versions of many traditional hypothesis tests (e.g., Bayesian ANOVAs). Heuristics can be used, such as prior sensitivity analyses, to ensure that the data do not violate the model assumptions too much. However, if the model is misspecified, then the result of these analyses can be difficult to interpret and the values of heuristics may be misleading. For example, just as it is problematic to perform an ANOVA to analyze percentage data, using a Bayesian ANOVA would also be problematic. Although frequentist nonparametric tests help with these problems, they also suffer some of the same problems discussed above as parametric frequentist tests while decreasing the statistical power. Bayesian nonparametric tests are possible to use (Ferguson, 1973), but are at the forefront of research in statistics, quite nuanced (e.g., the Dirichlet Process Gaussian Mixture Model is consistent as an

estimator of a probability density, but not as an estimator of the number of latent clusters; Ghosal, Ghosh, & Ramamoorthi, 1999; Miller & Harrison, 2013), and not understood well enough at this point in time to serve as the foundation for analyzing psychology experiments. Ideally, researchers should have access to a set of Bayesian data analysis tools between simple parametric and fully nonparametric models that allow the investigation of hypotheses in more complex data. Further, the tools should make it easy for researchers to define models that reflect any structure inherent in their experimental data.

Probabilistic programming languages (PPL) empower researchers to define arbitrarily complex, semi-parametric models for their data sets while still being straightforward to conduct statistical analyses. For conducting Bayesian data analyses, three PPLs, BUGS (Lunn, Spiegelhalter, Thomas, & Best, 2009), JAGS (Plummer, 2017), and STAN (Carpenter et al., 2017), have gained prominence within psychology (Kruschke, 2014; Lee & Wagenmakers, 2014).[1] One of the main advantages of these languages is the ability to estimate distributions that are otherwise mathematically intractable. This is primarily possible through the use of Markov chain Monte Carlo (MCMC) methods – a family of techniques that allow for the the sampling of these distributions over multiple iterations. Within a Bayesian framework, this is especially useful for the sampling of the posterior distributions of a model.

This tutorial functions primarily as a guide on the practice, rather than theory, of performing Bayesian data analysis using a PPL language known as `PyMC3`, which is not often used within the psychology community. Written as a Python package, this PPL is versatile while being highly accessible to researchers without an extensive background in computational statistics. Relevant theory will be presented to the extent that it aids and guides the practice itself, and interested readers are strongly encouraged to follow the rich resources exploring these theories in much greater detail. This tutorial should also serve as a concise introduction to Bayesian parameter estimation and model comparison using

[1] As of the writing of this manuscript, companion code for Lee and Wagenmakers (2014) ported to `PyMC3` is available at the following GitHub page: `https://github.com/pymc-devs/resources/tree/master/BCM`

MCMC. Readers are at the very least assumed to have a basic understanding of probability and Bayesian inference (e.g., what conditional probabilities are) and some familiarity with the Python programming language (e.g., what `numpy` arrays are and how they generally work).

Unlike BUGS, JAGS, or STAN, `PyMC3` is built on top of `Theano` (Theano Development Team, 2016), a powerful optimization library that allows for efficient computation of complex tensor operations. For this tutorial, readers do not need to know any `Theano` as `PyMC3` interfaces with it for them. A major benefit of being built on `Theano` is that `Theano` can perform optimization computation on either a Central Processing Unit (CPU) or Graphics Processing Unit (GPU). As we demonstrate in Section 6, GPU-based optimization enables model estimation for models with many parameters given very large data sets. In fact, it was used to create many of the deep learning methods over the last decade.

Supplementing this tutorial is a `README.md` file which contains our preferred installation methods as well as troubleshooting guides to some of the issues we have encountered. This file will be actively maintained on the tutorial's GitHub repository (`https://github.com/AusterweilLab/pymc3tut_dist`) as `PyMC3` undergoes its own development and questions arise from readers.

The `PyMC3` project documentation website (https://docs.pymc.io) contains excellent documentation for the language (see also Salvatier, Wiecki, & Fonnesbeck, 2016), and also provides specific details on installing `PyMC3`. The website clearly details the main advantages of `PyMC3`, including its ease of use, accessibility, large library of built-in distributions, and customizability with convenience. The documentation for a large portion of its more commonly used applications are quite thorough. We refer the reader to their documentation if they are struggling with a commonly used model that we do not cover in this tutorial. Without repeating too much of the material that is already online, this tutorial condenses some of the core `PyMC3` functionality that are most relevant to

psychologists without assuming a strong mathematical background. As a case study, we will focus on using `PyMC3` to conduct Bayesian data analysis with a series of increasingly complex Gaussian mixture models.

Most of the code described in here are compiled as separate Python scripts provided with the tutorial. Code blocks drawn directly from these files will appear within frames with numbers indicating their lines on the corresponding files. Occasionally we will encourage the reader to fiddle with some statements or introduce other code blocks that do not form part of the main script – these code blocks will appear without numbers on the left.

The tutorial will begin with the construction, optimization, and analysis of a univariate Gaussian mixture model. Progressive sections will then build on the preceding material by exploring `PyMC3`'s capabilities in performing sampler diagnostics, predictive checks, and model comparison. More specifically, in Section 3 we fit a univariate Gaussian mixture model to an appropriate simulated data set. Section 4 will then explore how the language easily allows for the generalization from univariate to multivariate models, ultimately leading up to a complex hierarchical model that is fit to real-world data in Section 5. Section 6 contains an introduction to the difference between computing on a CPU and a GPU using `PyMC3` and how a researcher should decide which to use for their own work.

We leave installation details to the `README.md` file, as the software packages are continuously changing. This tutorial will assume that you are working with Python 3.7 with standard libraries `numpy`, `scipy`, and `matplotlib`. You will also need to install `Theano` and `PyMC3`. Computing on the GPU requires a `NVIDIA` graphics card (it relies on the proprietary CUDA general purpose GPU coding framework) and may require different installation than the standard ones provided by `pip`.

2 Why use a Probabilistic Programming Language?

From a computational perspective, Bayesian inference can be notoriously difficult. Historically, Bayesian data analyses were limited to either cases limited to a parameterized family of prior distributions for each data type (conjugate families). For example, closed form solutions for updating the probability of the mean given observed data, the data variance, and the parameters for a Gaussian prior on the mean exist and are straightforward to calculate, even by hand. The advent of MCMC techniques and more powerful computational processing enabled more complex Bayesian models to be used (Gelfand, Hills, Racine-Poon, & Smith, 1990). Initially, researchers would often write model-specific code to perform Bayesian inference. If the same abstract process is being used to conduct Bayesian inference for each model (e.g., MCMC), creating model-specific code is inefficient and more likely to produce errors. Although there is an initial learning curve for learning how to use the language, PPLs allow researchers to focus their efforts on specifying the appropriate probability model and statistics to compute for their data and research question(s).

As indicated earlier, other PPLs already exist (e.g., `BUGS`, `JAGS`, and `STAN`). There are different strengths and weaknesses to all of them (e.g., discrete distributions are more difficult to use in STAN, whereas JAGS is often less efficient for continuous distributions). In this article, we illustrate how to use one particular PPL, `PyMC3`. There are three main reasons for selecting `PyMC3` for the tutorial over other PPLs: (1) a large suite of statistical data analysis and model comparison tools are built into the language, (2) it can be "run" on a Graphics Processing Unit (GPU) which has distinct benefits and weaknesses (as we illustrate in Section 6), and (3) unlike `BUGS`, `JAGS`, or `STAN`, there are currently no resources for learning `PyMC3` that were written with an audience of psychologists in mind.

3 Univariate Gaussian Mixture Modeling

In this section, we demonstrate the core functionality of `PyMC3` by specifying a simple univariate Gaussian mixture model and fitting the model to simulated data. In its simplest form, a Gaussian mixture model assumes that the data are generated by a mixture of one or more Gaussian distributions. Within a Bayesian framework, we make the additional assumption that the parameters specifying these Gaussian distributions are themselves generated by some prior distribution as well as on the mixture weights. Although we will have more specific examples later, we will start with each data point encoding an arbitrary one-dimensional continuous value. These may be the nose sizes of different animals from an animal shelter that has cats, dogs, and ferrets; student test scores; or movie reviews from different critics. The goal here is typically to infer some statistic of interest (e.g., the mean, or standard deviation) for each group in a data set, where each data point is a one-dimensional value and there is no class information (e.g., the nose sizes of each animal without knowing whether the animal is a dog, cat, or ferret).

This section of the tutorial directly corresponds to the provided `gmm.py` script available from the accompanying GitHub repository. We guide the reader through each block of code and provide the key output at each step.

We begin by importing the necessary Python packages and generating our simulated data. The following code block below specifies three normal distributions with different means and standard deviations and takes 1000 random samples from these distributions at the specified proportions. Running this section should result in the generation of data presented in Figure 1.

```
1 import pymc3 as pm
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 # Seed the rng for exact reproducible results
6 seed = 68492
7 np.random.seed(seed)
```

```
8
9  ## Generate data with K components.
10 # Means, standard deviations, proportions
11 mus = [ 0,  6,-5]
12 sigmas = [ 1, 1.5, 3]
13 ps  = [.2, .5,.3]
14 # Total amount of data
15 N = 1000
16 # Stack data into a single array
17 y = np.hstack([np.random.normal(mus[0], sigmas[0], int(ps[0]*N)),
18                np.random.normal(mus[1], sigmas[1], int(ps[1]*N)),
19                np.random.normal(mus[2], sigmas[2], int(ps[2]*N))])
20
21 ## Plot the data as a histogram
22 plt.hist(y, bins=20, alpha=0.5)
23 # Add axes labels
24 plt.xlabel('Simulated values')
25 plt.ylabel('Frequencies')
26 plt.show()
```

PyMC3 comes in when we specify our model and run the sampling process. Before presenting the model in code, it is useful to specify the model formally using generative process notation. We define the mixture model as follows,[2]

$$\begin{aligned}
\boldsymbol{p} &\sim \text{Dir}(\boldsymbol{\alpha}) \\
z_i|p_k &\sim \text{Cat}(\boldsymbol{p}) \\
\mu_k &\overset{iid}{\sim} N(0, 10) \\
\sigma_k &\overset{iid}{\sim} \text{HalfCauchy}(1) \\
y_i|z_i = k &\sim N(\mu_k, \sigma_k)
\end{aligned}$$

Here our model assumes each observation y is sampled from one of the K number of components. Each component is a normal distribution with a mean μ_k with a normally distributed prior and standard deviation σ_k with a half-Cauchy prior.[3] The individual

[2]Whenever possible, we try to maintain the following mathematical notation standards: scalar variables as lowercase italicized letters, dimension sizes as uppercase italicized letters, vectors as lowercase bold letters, and matrices as uppercase bold letters.

[3]A discussion on the choice of priors is outside the scope of this article. However, we direct the reader

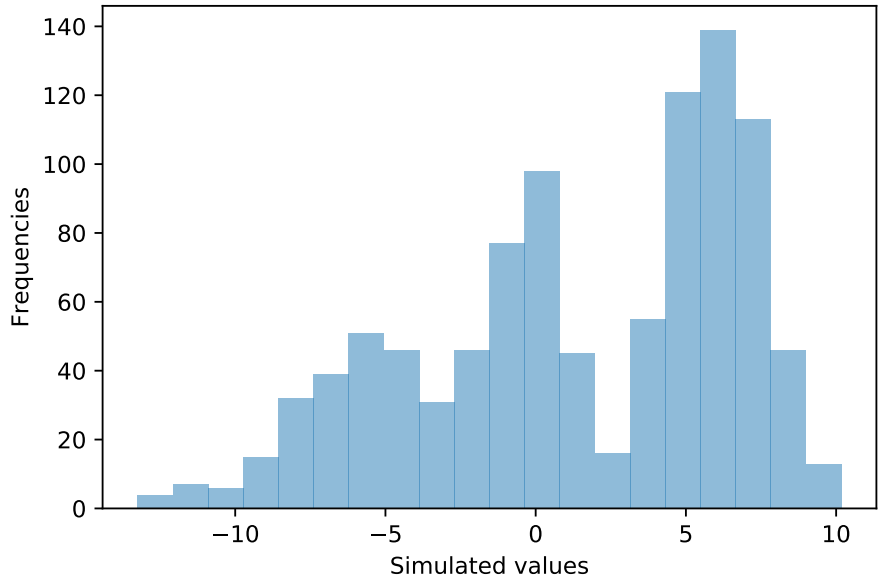

Figure 1: Simulated data sampled from the mixture of three normal distributions.

group assignments z_i are samples from a categorical distribution with probabilities $\boldsymbol{p} = p_1 ... p_K$, over which we use the Dirichlet distribution (parameterized by a concentration parameter vector $\boldsymbol{\alpha} = \alpha_1 ... \alpha_K$) as a prior. Note that since we have no prior expectations about the propensity of each cluster component, we set each element of $\boldsymbol{\alpha}$ to the same value of 1. This is equivalent to a uniform distribution over all 3-dimensional discrete probability distributions.

This code block completely specifies our model with $K = 3$ components.

```
28 ## Build model
29 gmm = pm.Model()
30 # Specify number of groups
31 K = 3
32
33 with gmm:
34     # Prior over z
35     p = pm.Dirichlet('p', a=np.array([1.]*K))
36
```

to Gelman (2006) for specific information on why the half-Cauchy is chosen as the prior over the standard deviation.

```
37      # z is the component that the data point is being sampled from.
38      # Since we have N data points, z should be a vector with N elements.
39      z = pm.Categorical('z', p=p, shape=N)
40
41      # Prior over the component means and standard deviations
42      mu = pm.Normal('mu', mu=0., sd=10., shape=K)
43      sigma = pm.HalfCauchy('sigma', beta=1., shape=K)
44
45      # Specify the likelihood
46      Y_obs = pm.Normal('Y_obs', mu=mu[z], sd=sigma[z], observed=y)
```

In PyMC3, model specification requires the variables to be declared within the context of the language's Model object, that is, within a `with gmm:` statement where gmm is a Model object (see lines 29 and 33 in our example). The declaration of the model parameters within this context allows all relevant variables to be associated with the appropriate model. Trying to create a random variable object without specifying the model explicitly outside the context of a PyMC3 Model will result in an error.

Indented statements inside the `with` statement (lines 34 to 46) show the specification of each model parameter. A statement such as `mean = pm.Normal('mean', mu=0, sd=1, shape=1)` tells PyMC3 that you are defining a random variable named mean that has a shape of 1 (i.e., this random variable is a scalar), follows a normal distribution with a mean of 0 and standard deviation of 1, and assigning it to a Python variable called mean. Every random variable that is declared within a model's context requires a string passed as the first argument, which is used as the name of the random variable within the model. Although it may be confusing at first, this is distinct from the Python variable to which it is assigned. In practice it is most useful to have the name of the random variable match the name of the Python variable to which it is assigned (as is done in our example), although this is not a hard rule. The shape argument is an optional argument for most of the built-in distributions that come with PyMC3, although notable exceptions include some multivariate distributions (discussed more in the next section). Since our current model involves the estimation of K means and variances (different μ_k and σ_k values), our mu and

`sigma` variables need to be defined with a shape of K (lines 42 and 43).

To complete the model, the likelihood function for the data must be defined. This is performed in line 46, where the specification of the `observed` argument explicitly flags the random variable `Y_obs` as one that is not a latent variable that can vary across model simulations (i.e., it will not be estimated), but instead is given by the data. The use of `z` to index `mu` and `sigma` is useful here because `z` can only take on integer values, allowing us to perform indexing on the `mu` and `sigma` variables (despite their values being unknown and estimated for each simulation by `PyMC3`).

To start the sampling process, we use the `pm.sample()` function. This function should be placed within the context of our model, and can be run without arguments. In practice, it may be useful to specify at least the number of iterations or draws that we want the sampler to run with `draws` (default is 500 draws), the number of chains with `nchains` (default is 4), and the sampling algorithm with `step`. Without a value passed to `step`, `PyMC3` will try to automatically assign an appropriate algorithm depending on the type of variable, with No U-Turn Sampling (NUTS; Hoffman & Gelman, 2014) used for continuous variables and Gibbs sampling used for discrete variables (`ElemwiseCategorical` in `PyMC3`). After sampling, the resulting set of MCMC chains are returned as a single `PyMC3` `trace` object, which contains the sampled values for every iteration in every chain and is used for subsequent analyses. In `gmm.py`, we explicitly define the sampling algorithms and run the sampler with the following lines:

```
48 ## Run sampler
49 with gmm:
50     # Specify the sampling algorithms to use
51     step1 = pm.NUTS(vars=[p, mu, sigma])
52     step2 = pm.ElemwiseCategorical(vars=[z])
53 
54     # Start the sampler!
55     trace = pm.sample(draws=2000,
56                       nchains=4,
57                       step=[step1, step2],
58                       random_seed=seed)
```

If `nchains` is more than 1, `PyMC3` will attempt to run chains in parallel on every available core, which can provide faster sampling times for computing devices (CPU or GPU) with multiple cores. Running the above code block should yield an output similar to below:

```
Multiprocess sampling (4 chains in 4 jobs)
CompoundStep
>CategoricalGibbsMetropolis: [z]
>NUTS: [sigma, mu, p]
100%|____________| 10000/10000 [00:48<00:00, 204.83draws/s]
The gelman-rubin statistic is larger than 1.4 for some parameters. The sampler did
  not converge.
```

We will discuss the failure to converge message in Section 3.1.2.

Note that although we set the sampler to take 2000 draws, the total number displayed above is 2500. The extra 500 draws are the first 500 tuning steps that the sampling algorithm takes to allow the Markov chain to sample from a reasonable space in the distribution. By default, these initial draws are discarded and not included in the `trace` object that is returned. Within `pm.sample()` you can set the argument `discard_tuned_samples = False` to preserve these draws, and also specify a different number of tuning draws with the `tune` argument (e.g. `tune = 1000`). Note that we provide a seed to the random number generator. This is so we get the same sequence of random numbers and can reproduce the results. Using a different seed or leaving it unspecified should still provide very similar quantitative results, assuming everything is working properly.

3.1 Diagnostics

3.1.1 Visual inspection of trace plots

`PyMC3`'s accessibility lies not only in its ease of model specification, but also its library of built-in functions to diagnose and analyse the resulting set of MCMC samples. One of the

primary checks of sampling adequacy (and indeed whether our model is well specified), is a visual analysis of the trace plots and corresponding posterior estimates. This can be performed by running the following code block (note that it no longer falls inside the model context):

```
60 # Plot results
61 pm.traceplot(trace,
62              varnames=['mu','p','sigma'], # Specify which variables to plot
63              lines={'mu':mus,'p':ps,'sigma':sigmas}) # Plots straight lines - useful
       for simulations
64 plt.show()
```

The `pm.traceplot()` function requires passing the `trace` object as the first argument. Commonly used optional arguments include `varnames` to specify a list of the model's random variables that you want to include in the plot, as well as `lines` to overlay convenient lines on the plots (e.g., the true parameter values from the data simulation). Values for the `lines` argument should be a Python dictionary of `variable_name: location_of_line` (key-value pairs). Note that the variable names specified for this argument should be the names that were defined within the model, and not the names of the Python variables the distributions were assigned to. For instance, if we had changed line 42 to `mu = pm.Normal('means', mu=0., sd=10., shape=K)`, then to plot the appropriate traces we would need to change the values of `varnames` from `['mu','p','sigma']` to `['means','p','sigma']`.

The results are presented in Figure 2, with the left column of plots showing the estimated posterior distributions for each specified variables, and the right column of plots showing the paths of the corresponding chains (these plots are commonly referred to as *trace* plots).

To ascertain if the sampler was making appropriate draws from the posterior distributions, we would expect to see in the trace plots a series of chains moving around a localized region in the parameter space. Since our random variables here were defined as

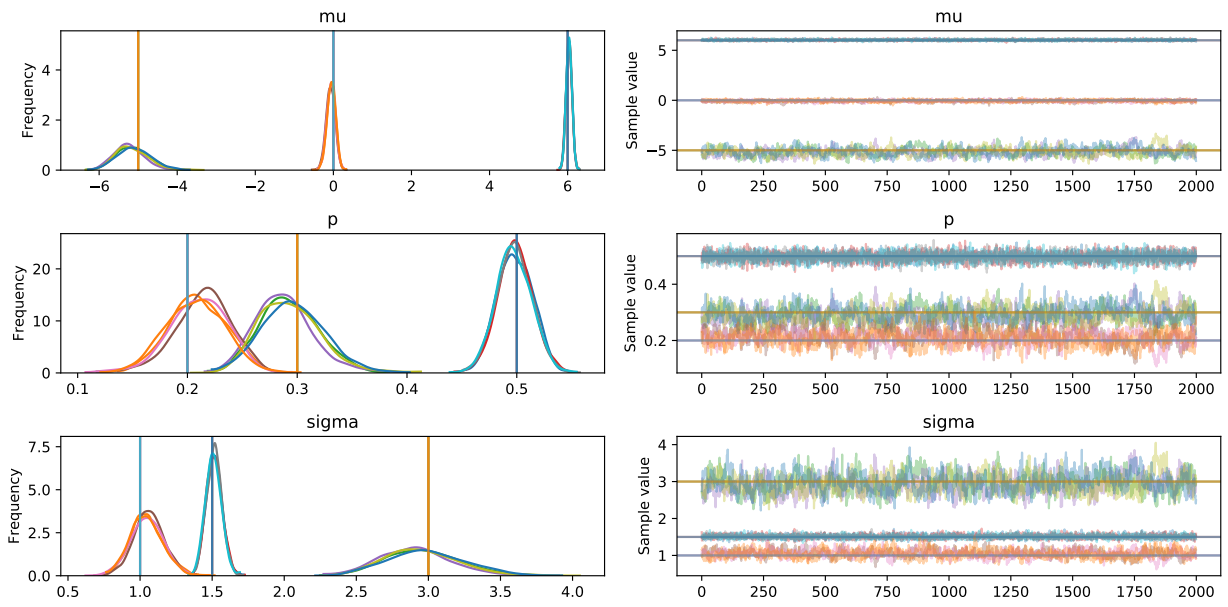

Figure 2: Summary plots for the mu, sigma, and p parameters of the `gmm` model. Estimated posterior distributions are presented on the left column of plots, with corresponding trace plots for each parameter shown in the right column of plots. Straight lines indicate the true parameter values used to generate the data.

vectors of three elements, we should see in Figure 2 exactly three regions where the chains maintain a generally flat trajectory. An inspection of the trace plots in Figure 2 indicates good convergence. In addition, the observation that the chains and posterior plots seem to follow the straight lines (i.e., the true parameter values) is also a good sign that the sampler is working correctly and that the model is not misspecified. Unfortunately, data typically do not include the parameter values estimated by the model (if it did, why use the sampler at all?). Thus, other convergence evaluations are typically used in conjunction with visual inspection of the trace plot.

3.1.2 The Gelman-Rubin Convergence Criterion

The visual confirmation of convergence appears to contradict the failure to converge warning that was printed after running the sampler. This warning is based on the Gelman-Rubin criterion (Gelman & Rubin, 1992), which is (roughly) the ratio of the

pooled variance of values across all chains to the average variance of each chain. Intuitively, this should be equal to 1 if all the chains converge because at that point, they are all sampling from the same underlying distribution and thus should share the same variance. If they are stuck in their own different local regions of space, then each chain's variance will be much smaller than the variance of the values pooled across chains. It provides a useful quantitative (albeit imperfect; see Robert & Castella, 2006 for more detailed information about MCMC convergence) guide for convergence quality, where values close to 1 indicate good convergence. The precise Gelman-Rubin criterion values can be printed using the statement

```
print(pm.gelman_rubin(trace,varnames=['mu','sigma','p']))
```

This returns the following output

```
{'sigma': array([3.39960389, 5.13348064, 5.63645693]),
 'p': array([7.59986218, 5.14964063, 2.00216592]),
 'mu':  array([36.83303876, 18.30074953, 10.19890733])}
```

In this output, we see that none of the results for the Gelman-Rubin statistic values are close to 1. The reason for this discrepancy is known as the *label switching* problem for mixture models (Jasra, Holmes, & Stephens, 2005; Richardson & Green, 1997), and it is due to the non-identifiability of the component indices. That is, the sampler does not know which index label to assign to each component, and so across chains, the sampler is likely to assign different labels to the same cluster component. Recall that we have specified each parameter here as a 3-element vector. The first chain can end up drawing samples of the component mean μ_k such that first, second, and third elements of the vector correspond directly to the first, second, and third component means respectively. However, the second chain is just as likely to have the second, first, and third indices (or any other permutation) correspond to the first, second, and third component means, respectively. Formally, the label indices in the model are exchangeable, meaning that any permutation is assigned the

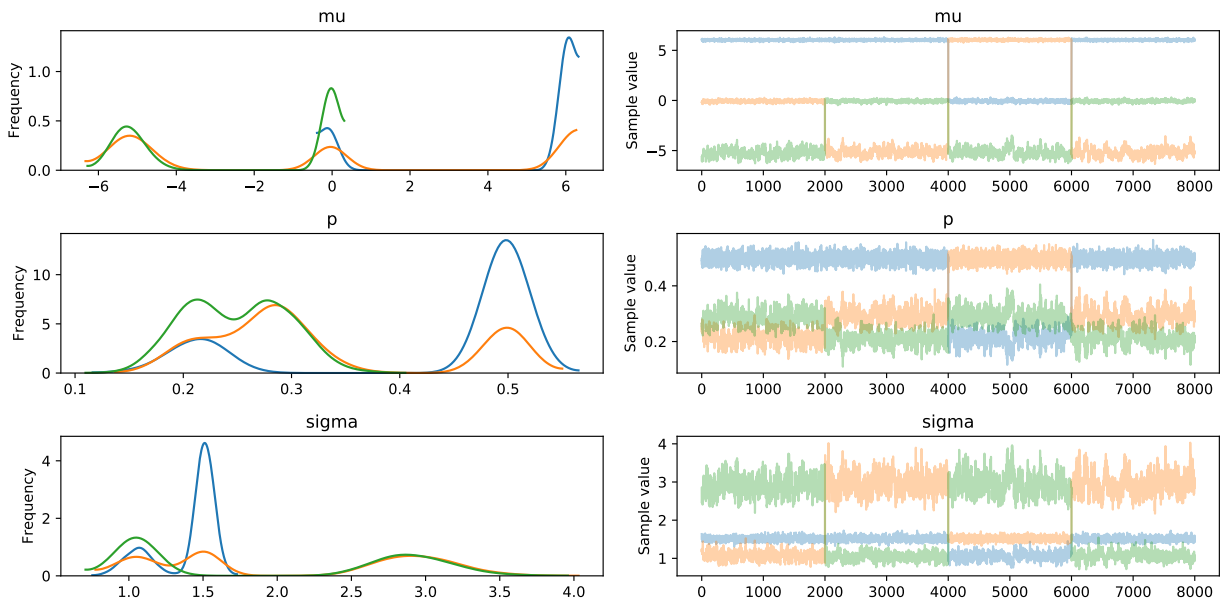

Figure 3: Summary plots for the mu, sigma, and p parameters of the `gmm` model with individual chains combined into a single chain for each parameter. Estimated posterior distributions are presented on the left column of plots, with trace plots for each parameter shown in the right column of plots.

same joint probability. Consequently, there is no way to guarantee that the ordering of the component parameters being sampled in the first chain is the same ordering of component parameters in any of the other chains, leading to warnings of divergent chains. To visualize this, we can plot the trace plots such that each chain follows after the other by specifying the following argument `combined=True`, like so:

```
pm.traceplot(trace,
varnames=['mu','p','sigma'],
combined=True)
```

The resulting trace plots are presented as Figure 3. For clarity, we omit the straight lines indicating the true values.

This behavior is expected in sampling from mixture models, but it can be confusing for researchers who do not visually inspect the trace plots (the authors strongly recommend that researchers always inspect them when possible) and is interested in

quantitatively evaluating convergence. One way to address this issue is by explicitly providing the same starting value across chains for a component variable using the `testval` argument so that the each chain is likely to have the same mean value assigned to each component. Due to the dependencies between each parameter, `testval` only needs to be specified for one parameter that is part of a set of related exchangeable variables. In this case, we change line 42 to the following

```
mu = pm.Normal('mu', mu=0., sd=10., shape=K, testval=[-2,0,2])
```

and running `gmm.py` should then produce similar results as before, but without the divergence warnings.

Divergence warnings can be a serious concern and are often a sign of model misspecification and/or one or more chains in the sampler "getting stuck" in local minima. To illustrate when this can occur, we can deliberately misspecify the prior over component means of this example by making it bimodal and constrained between zero and one. To demonstrate this, we can again change line 42 to

```
mu = pm.Beta('mu', .5, .5, shape=K)
```

Running the sampling process with this line changed should now return output with a number of divergence warnings, as expected.

3.1.3 The Posterior Predictive Check

Another useful technique to assess the convergence of a MCMC sampler is to sample data from the model estimated. That is, we can generate data from the model using the posterior distribution over data based on the parameters estimated by the sampling process. This distribution of generated data is called the posterior predictive distribution, and assessing its relation to the observed data is called conducting a *posterior predictive check* (Rubin, 1984). The data produced through a posterior predictive check are those data one

would expect if the proposed model were true and the sampler converged properly. One nice property of `PyMC3` is that conducting a posterior predictive check is straightforward given a trace. We demonstrate an example of this between lines 66 and 77 of `gmm.py`:

```
66 ## Posterior Predictive Checks
67 # Obtain posterior samples
68 pp = pm.sample_ppc(model=gmm, trace=trace)
69 # Plot original data
70 plt.hist(y, bins=20, alpha=0.5)
71 # Plot posterior predictives on top of that
72 plt.hist(np.random.choice(pp['Y_obs'].flatten(),size=len(y)), bins=20, alpha=0.5)
73 # Add legend and axes labels
74 plt.legend(['Data','Predictions'])
75 plt.xlabel('Simulated values')
76 plt.ylabel('Frequencies')
77 plt.show()
```

Running this code block yields Figure 4. Upon inspection we see that the model predictions closely resemble the data, aligning well with our expectations.

3.2 Model comparison

Although model evaluation via convergence checks is useful, it is very difficult to interpret the model's performance and test scientific hypotheses without comparing the model to some alternative. Thus, the model is compared against some other standard – typically one or more other models that differ from the main model in systematic ways (e.g., a 2-component mixture model vs. a 3-component mixture model). Generally, model selection is performed by comparing the models on an information criteria or through cross-validation. Both methods have the same goal, that is to estimate how well a model captures the "true" distribution over the model's parameters and data. As it usually is impossible to know either of these, we use surrogates, such as evaluating the average fit of some data given the model trained on the rest of the data (cross-validation) or the fit of the entire data set after correcting for the model's flexibility from its parameters (penalized posterior probability, also called information criteria). We will discuss both, but the latter

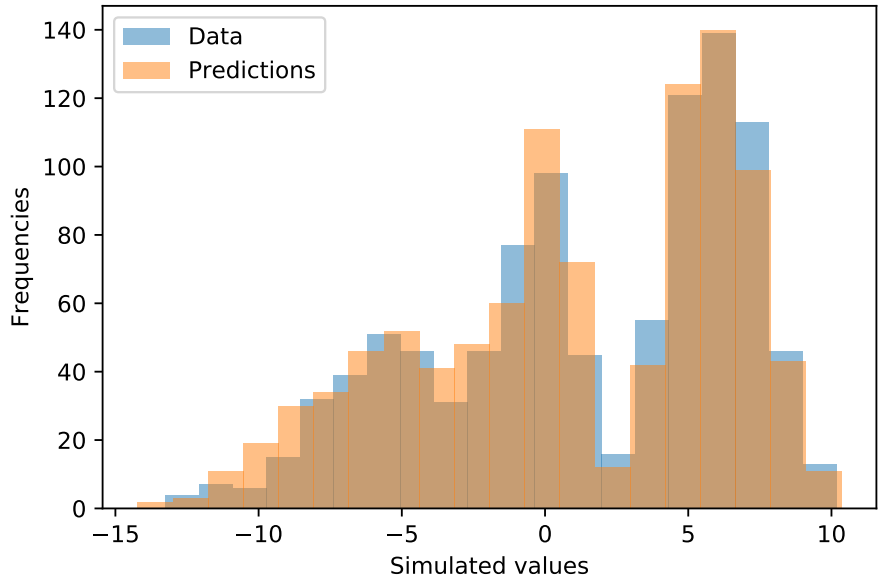

Figure 4: Posterior predictive samples from `gmm` model overlaying simulated data sampled from three normal distributions.

method is often used because it is simpler and faster than cross-validation, which can be computationally expensive when many possible randomly selected training and evaluation sets are used. Further, some information criteria can be viewed as an approximation to cross-validation without needing to perform multiple model fits (Stone, 1977).

`PyMC3` currently includes a function to compute the *Widely Applicable Information Criterion* (WAIC; Watanabe, 2010)[4] and also a cross-validation method known as *leave-one-out* (LOO; Vehtari et al., 2017), where model performance is evaluated on each data point after training the model on the rest of the data. As compared to WAIC, LOO is currently somewhat controversial (Gronau & Wagenmakers, 2019; Vehtari, Simpson, Yao, & Gelman, 2019), though both measures can be used and interpreted in a similar manner, and typically provides similar results. Here, we will focus on performing and interpreting

[4]Aside from the WAIC, the Deviance Information Criterion (DIC; Spiegelhalter, Best, Carlin, & Van Der Linde, 2002) is also commonly used in Bayesian model selection. As of this writing, `PyMC3` does not include direct DIC computation. However this is unlikely to be problematic since the WAIC is generally recommended over the DIC (e.g., see Gelman, Hwang, & Vehtari, 2014; Vehtari, Gelman, & Gabry, 2017).

WAICs.

In this section, we perform model comparison across simple variants of the previous `gmm` model, with the relevant script contained in `gmm_compare.py`. To motivate this, let us again assume that we do not know the generating process for our data, and consequently we may not know if specifying $K = 3$ in our earlier `gmm` model is the best assumption. For example, we may want to test whether an animal shelter has two or three species of animals given only the nose size of each animal in the shelter. If the data are also well accounted for by a simpler model (e.g., with $K < 3$), then those models are usually preferred. Building this set of models is straightforward – simply loop the model construction as many times as desired (three times in our example) such that on each iteration a new `pm.Model` object is built with a specific K number of components. The following code block begins from line 21 since the preceding lines set up the data in exactly the same way as before in `gmm.py`.

```
21 ## Build list of models
22 # We want a range of models ranging from K=1 to K=3
23 K_range = range(3)
24 # Initialize lists
25 gmm_list = []
26 trace_list = []
27 # Specify some sampling options
28 draws = 2000
29 nchains = 4
30
31 # Loop through K_range, building model and sampling each time
32 for ki in K_range:
33     # Add a new Model object to the list
34     gmm_list += [pm.Model()]
35     K = ki+1 # Here ki is the index, K is the number of groups the model assumes
36
37     # Name each model for easy identification during comparison
38     gmm_list[ki].name = '%d-Group' % K
39
40     with gmm_list[ki]:
41         # Prior over z - only applicable if K>1
42         if K>1:
43             p = pm.Dirichlet('p', a=np.array([1.]*K))
44
45             # z is the component that the data point is being sampled from.
```

```
            # Since we have N data points, z should be a vector with N elements.
            z = pm.Categorical('z', p=p, shape=N)

        # Prior over the component means and standard deviations
        mu = pm.Normal('mu', mu=0., sd=10., shape=K)
        sigma = pm.HalfCauchy('sigma', beta=1., shape=K)

        # Specify the likelihood
        if K>1:
            Y_obs = pm.Normal('Y_obs', mu=mu[z], sd=sigma[z], observed=y)
            # Specify the sampling algorithms to use
            step1 = pm.NUTS(vars=[p, mu, sigma])
            step2 = pm.ElemwiseCategorical(vars=[z])
            steps = [step1,step2]
        else:
            Y_obs = pm.Normal('Y_obs', mu=mu, sd=sigma, observed=y)
            # Specify the sampling algorithms to use - don't include z or p  because
    they don't exist when K==1
            step1 = pm.NUTS(vars=[mu, sigma])
            steps = [step1]

        # Start the sampler, and save results in a new element of trace_list
        trace_list += [pm.sample(draws=draws,
                                 nchains=nchains,
                                 step=steps,
                                 random_seed=seed)]
```

The model structure itself is exactly the same as before, except now three different instances of the model are created, each with an increasing number of components. Note that we do not define the `p` and `z` random variables when $K = 1$ as they are unnecessary and including them in the model slows convergence. Importantly, we are also adding the results of each sample (i.e., the returned `trace` object for each model) into a list so that the appropriate comparisons can be made after all models have been sampled.

With the samples ready, we can directly compare the models with a single function `pm.compare()`. Both model and trace objects of interest need to be passed as a dictionary as the first argument. The function returns a `pandas` dataframe containing the WAIC statistics. This dataframe can be used to summarise the results in a single plot with `pm.compare()` We demonstrate this with the following code block:

```
## Model comparison
# Convert model and trace into dictionary pairs
dict_pairs = dict(zip(gmm_list,trace_list))
# Perform WAIC (or LOO by setting ic='LOO') comparisons
compare = pm.compare(dict_pairs, ic='WAIC')
# Print comparisons
print(compare)
```

This code block returns

	WAIC	pWAIC	dWAIC	weight	SE	dSE	var_warn
3-Group	4182.13	191.34	0	1	42.6	0	1
2-Group	5067.13	237.31	885	0	38.87	22.22	1
1-Group	6159.3	1.63	1977.17	0	34.66	43.48	0

By default, comparisons are made on the WAIC, but this can be changed by specifying `ic='LOO'` if desired.

Looking at the output again, the primary result would be the first column, which presents the WAIC computed for each model. Lower numbers indicate better fitting models. In this case, we are able to conclude that the 3-component model is the best performing model out of this set. This aligns nicely with our intuitions – the data were generated using three distributions, so naturally the model with three components should fit the best. The second column from the left shows the effective number of parameters in each model. This represents the flexibility of the model, where larger values indicate more flexible models. The third column indicates the relative difference in WAIC values from the best-performing model. The fourth column shows the WAIC weights, which provide an alternative interpretation of model performance through a transformation the raw WAIC values into weighted values within this set of models (see Wagenmakers & Farrell, 2004). Weights take any value between 0 (relatively poorly fitting) and 1 (relatively well fitting). The fifth and sixth columns list the standard errors (reflecting the uncertainty) of the WAIC and dWAIC respectively. The final column indicates whether a warning was flagged – a value of 1 may indicate that the WAIC results are not reliable. For our purposes we can

ignore these warnings. We refer interested readers to Vehtari et al. (2017) for more details.

4 Multivariate Gaussian Mixture Modeling

In the previous section we explored the construction of univariate Gaussian mixture models and fit them to univariate data. However, many real-world stimuli do not vary along only a single dimension. In this section we will further demonstrate `PyMC3`'s robustness in model specification by fitting a multivariate Gaussian mixture model to an artificial bivariate data set.

The code for this section can be found in `mvgmm.py`. The data generation portion is similar to data from the previous sections, but here we use an ellipse-plotting function to indicate the 95% density region of the generating distribution, as seen in Figure 5.

```
1  import pymc3 as pm
2  import numpy as np
3  import matplotlib.pyplot as plt
4  from theano.tensor.nlinalg import matrix_inverse
5  from ellipse import plot_ellipse
6
7  # Seed the rng for exact reproducible results
8  seed = 68492
9  np.random.seed(seed)
10
11 ## Generate some data for three groups.
12 # Means, variances, covariances, and proportions
13 mus = np.array([[-4,2],[0,1],[6,-2]])
14 variance1   = [1,.4,1.5]
15 variance2   = [1,.8,5  ]
16 covariances = [.5,0, -1]
17 ps = np.array([0.2, 0.5, 0.3])
18 D = mus[0].shape[0]
19 # Total amount of data
20 N = 1000
21 # Number of groups
22 K = 3
23 # Form covariance matrix for each group
24 sigmas = [np.array([[variance1[i],covariances[i]],[covariances[i],variance2[i]]])
       for i in range(K)]
25 # Form group assignments
```

```
zs = np.array([np.random.multinomial(1, ps) for _ in range(N)]).T
xs = [z[:, np.newaxis] * np.random.multivariate_normal(m, s, size=N)
      for z, m, s in zip(zs, mus, sigmas)]
# Stack data into single array
data = np.sum(np.dstack(xs), axis=2)

## Plot them nicely
# Prepare subplots
fig, ax = plt.subplots(figsize=(8, 6))
# First, scatter
plt.scatter(data[:, 0], data[:, 1], c='g', alpha=0.5)
plt.scatter(mus[0, 0], mus[0, 1], c='r', s=100)
plt.scatter(mus[1, 0], mus[1, 1], c='b', s=100)
plt.scatter(mus[2, 0], mus[2, 1], c='y', s=100)
# Then, ellipses
plot_ellipse(ax,mus,sigmas)
ax.axis('equal')
plt.show()
```

Theoretically, the leap from specifying a univariate to multivariate Gaussian mixture model is not too large. The components are now defined as multivariate normal distributions with the means sampled from a multivariate normal prior. The component covariance matrices are slightly more complex. Due to sampling issues that arise from the conjugate inverse-Wishart prior, most PPLs use as the Lewandowsky-Kurowicka-Joe (LKJ) prior over the Cholesky decomposition of the covariance matrices (Lewandowski, Kurowicka, & Joe, 2009). A Cholesky decomposition takes a positive definite matrix $\mathbf{A}$ and factorizes it into a lower triangular matrix $\mathbf{L}$ and its conjugate transpose $\mathbf{L^*}$ (i.e., $\mathbf{A} = \mathbf{LL^*}$). The LKJ prior is parameterized by σ_{LKJ} which affects the component standard deviations and η which affects the component correlations. More specifically, $\eta = 1$ results in a uniform distribution on the correlation matrices and increasing values of η lead to a decreasing magnitude of correlations. Values of η are constrained to be positive. To improve sampling efficiency, we also marginalize over the component assignments $\boldsymbol{z}$, allowing the PyMC3 to avoid sampling from a discrete distribution. This reparameterized

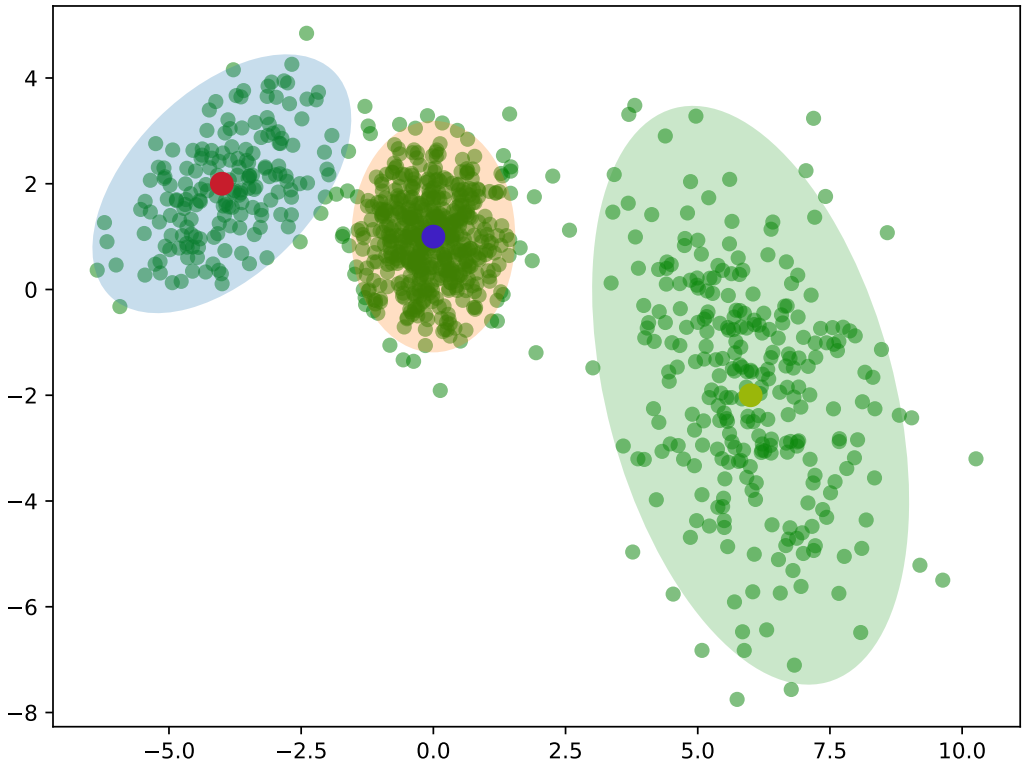

Figure 5: Simulated data generated from three bivariate Gaussian distributions. Ellipses indicate 95% density regions and large colored circles indicate the means of the generating distributions.

multivariate Gaussian mixture model can be more formally expressed as

$$
\begin{aligned}
\boldsymbol{p} &\sim \text{Dir}(\boldsymbol{\alpha}) \\
\mu_k &\sim \text{MvN}(\boldsymbol{\mu_0}, \mathbf{I}) \\
\sigma_{LKJ} &\sim \text{HalfCauchy}(1) \\
\Sigma_k &\sim \text{LKJCholeskyCov}(\eta = 2, \sigma_{LKJ}) \\
f(y_i|\boldsymbol{p}) &= \sum_{k=1}^{K} p_k \text{MvN}(y_i|\mu_k, \Sigma_k)
\end{aligned}
$$

where $\mathbf{I}$ is a D-sized vector of zeros, and $\boldsymbol{I}$ is a D-sized vector of ones, with D being the number of dimensions. Dir(.) is a Dirichlet distribution, MvN(.) is a multivariate Gaussian distribution, and LKJCholeskyCov is the LKJ prior over the Cholesky decomposition of covariance matrices.

With this model, we can also demonstrate an alternative method of indexing the component variables. In the earlier univariate models, we defined a single random variable for each parameter of interest as a vector where each element corresponded to a single component (e.g. `shape = K`). With a multivariate model, we could do something similar and have one axis of the variable correspond to the components, and the other to the dimensions (e.g. `shape = (K,D)`). Unfortunately, not all built-in distributions allow for this specification. Specifically, some multivariate distributions such as `pm.LKJCholeskyCov()` do not take a `shape` argument. To work around this, we can simply create a new random variable of size D for each component,[5] as seen in the following code block:

```
45 ## Build model and sample
46 # Number of iterations for sampler
47 draws = 2000
48 # Prepare lists of starting points for mu to prevent label-switching problem
49 testvals = [[-2,-2],[0,0],[2,2]]
```

[5] An alternative workaround would be to define new deterministic variables that stack the `pm.LKJCholeskyCov` samples on the third axis. However, there are no clear computational benefits of this workaround that we have found at this point.

```
50
51  # Model structure
52  with pm.Model() as mvgmm:
53      # Prior over component weights
54      p = pm.Dirichlet('p', a=np.array([1.]*K))
55
56      # Prior over component means
57      mus = [pm.MvNormal('mu_%d' % i,
58                         mu=pm.floatX(np.zeros(D)),
59                         tau=pm.floatX(0.1 * np.eye(D)),
60                         shape=(D,),
61                         testval=pm.floatX(testvals[i]))
62             for i in range(K)]
63
64      # Cholesky decomposed LKJ prior over component covariance matrices
65      packed_L = [pm.LKJCholeskyCov('packed_L_%d' % i,
66                                    n=D,
67                                    eta=2.,
68                                    sd_dist=pm.HalfCauchy.dist(1))
69                  for i in range(K)]
70
71      # Unpack packed_L into full array
72      L = [pm.expand_packed_triangular(D, packed_L[i])
73           for i in range(K)]
74
75      # Convert L to sigma and tau for convenience
76      sigma = [pm.Deterministic('sigma_%d' % i ,L[i].dot(L[i].T))
77               for i in range(K)]
78      tau = [pm.Deterministic('tau_%d' % i,matrix_inverse(sigma[i]))
79             for i in range(K)]
80
81      # Specify the likelihood
82      mvnl = [pm.MvNormal.dist(mu=mus[i],chol=L[i])
83             for i in range(K)]
84      Y_obs = pm.Mixture('Y_obs',w=p, comp_dists=mvnl,observed=data)
85
86      # Start the sampler!
87      trace = pm.sample(draws, step=pm.NUTS(), chains=4)
88
89  ## Plot traces
90  pm.traceplot(trace, varnames=['p',
        'mu_0','mu_1','mu_2','tau_0','tau_1','tau_2','sigma_0','sigma_1','sigma_2'])
91  plt.show()
```

Here we also introduce the `pm.Deterministic()` function. This function takes in an expression and stores it as a deterministic variable in the model. Up until this point, we

have primarily defined random variables within the model context. However, it is possible to not only have random variables sampling from each other – PyMC3 also allows us to directly perform operations between them. While these expressions can be successfully executed by themselves within the model context, if each expression is not passed as an argument into pm.Deterministic(), the model will not store samples from that expression. For example, in this model

```
with pm.Model() as model:
    mu = pm.Normal('mu', mu=0., sd=1.)
    mu_d= mu+5 #this just adds 5 to all samples of mu, but samples are not stored
    Y_obs = pm.Normal('Y_obs', mu=mu_d, sd=1,observed=data)
```

we will be able to infer mu through its relation to mu_d, but samples of mu_d themselves will not be stored. By moving the expression inside a pm.Deterministic() function (e.g., pm.Deterministic('mu_d', mu+5)), samples of the deterministic variables will be stored with the name that is specified and can be later accessed through the trace object.

One other novel aspect of this mvgmm model is the use of the built-in pm.Mixture() distribution function, which allows users to conveniently construct a mixture model with any component distribution and set of weights. This function automatically marginalizes over $\boldsymbol{z}$, so that does not need to be explicitly defined. However, this can be a drawback if we have an interest in sampling $\boldsymbol{z}$ directly. For instance, you may be interested in the posterior distribution of a particular data point (i.e., whether it is likely to be in one particular cluster or if there is much uncertainty), in which case explicitly defining $\mathbf{z}$ as we did in the earlier gmm model would be more appropriate than using pm.Mixture().

We can analyze the trace plots that emerge from the previous code block (Figure 6). Overall, divergence does not appear to be an issue. For each parameter corresponding to each component, we can see chains steadily sampling from the appropriate local region of the parameter space.

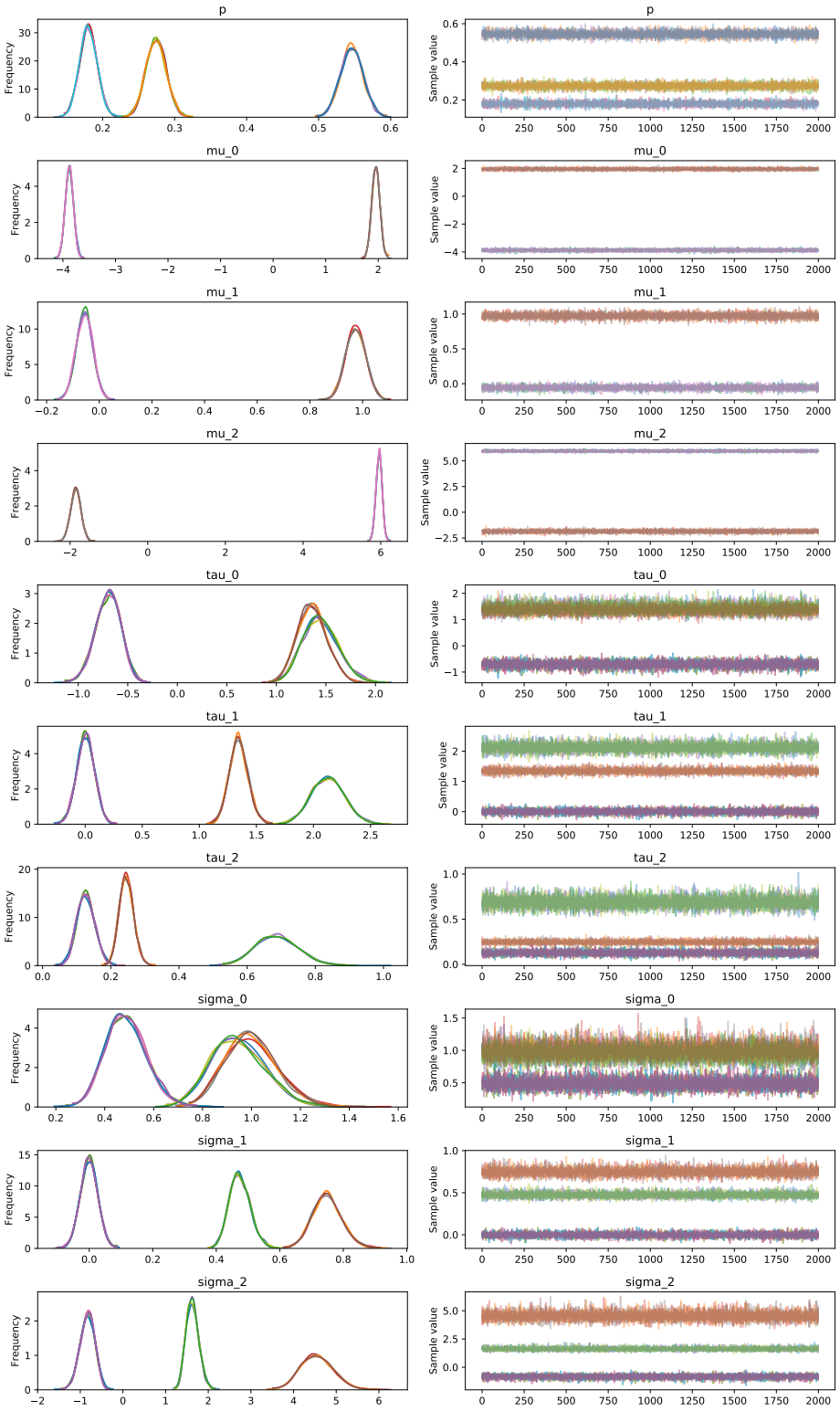

Figure 6: Summary plots for each parameter of the `mvgmm` model. Estimated posterior distributions are presented on the left column of plots, with corresponding trace plots for each parameter shown in the right column of plots.

5 Fitting data from an experiment

Up to this point, we have constructed artificial data that is explicitly generated from the same or similar process as our models. When working with data from an experiment instead of simulated data, the model will never perfectly capture the actual process that generates the data. In addition, empirical data are typically messier to work with. In this next section, we illustrate how `PyMC3` can be used to fit experimental data and some problems that can arise.

In this section, we fit bivariate Gaussian mixture models with increasing numbers of components to the results of a category generation experiment (analogous to those run in Austerweil, Conaway, Liew, & Kurtz, in preparation). In the experiment, stimuli were squares varying on two dimensions: size and color (white to gray to black; see Figure 7a). 21 participants were first shown four exemplars from Category A, which comprised stimuli tightly clustered around the center of the stimulus domain (Figure 7b). After training on each exemplar of the Category A three times in a randomized order (e.g., Figure 7c), participants were asked to generate eight exemplars from Category B (without ever seeing any exemplars other than those in Category A; e.g., Figure 7d). Generated exemplars were not restricted in any way other than the boundaries of the dimensions – participants were free to vary the dimensions of each generated exemplar as they wished as long as it was a member of a novel category. This data set is stored in the file `data.p` and the script corresponding to this section is `realdata.py`. Lines 1 to 42 of `realdata.py` begin with the importing of required packages and then loads and plots the data (Figure 8).

```
1 import pymc3 as pm
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import pickle
5 import re
6 import datetime
7 import os
8 from ellipse import plot_ellipse
9
```

```
# Seed the rng for exact reproducible results
seed = 68492
np.random.seed(seed)

## Load data
with open('data.p','rb') as f:
    data_all = pickle.load(f)

# Plot raw data
lim = 2.5 # Set the limits of the subplots
alpha = .5 # Set transparency of markers
markersize = 20 # Set size of markers
# Prepare subplots
fig, axx = plt.subplots(3,7,figsize=(25, 10))
# Create list of datasets containing only generated exemplars (not learned exemplars)
data = {}
for id,pid in enumerate(data_all.keys()):
    data[pid] = data_all[pid][20:]
# Loop through each participant and plot their data
for ip,pid in enumerate(data_all.keys()):
    # Identify an appropriate subplot
    ax = axx[np.unravel_index(ip, (3,7), 'C')]
    # Plot trained exemplars
    ax.scatter(data_all[pid][0:20, 0], data_all[pid][0:20, 1],s =
    markersize,color='blue',alpha=alpha,marker='o')
    # Plot novel exemplars
    ax.scatter(data_all[pid][20:, 0], data_all[pid][20:, 1],s =
    markersize,color='Red',alpha=alpha,marker='x')
    # Format axes
    ax.set_ylim(-lim,lim)
    ax.set_xlim(-lim,lim)
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    # Add title
    ax.set_title(pid)
```

The data set is imported as a Python dictionary with each key indicating a unique identifier for each participant. The corresponding value of each key is the set of exemplar dimension values stored as a two-dimensional array, with different exemplars separated by rows (ordered by trained exemplars first, then generated exemplars) and the dimensions separated by columns. Due to the small amount of data for each participant (four trained and eight generated exemplars), we have duplicated each data point five times with a small

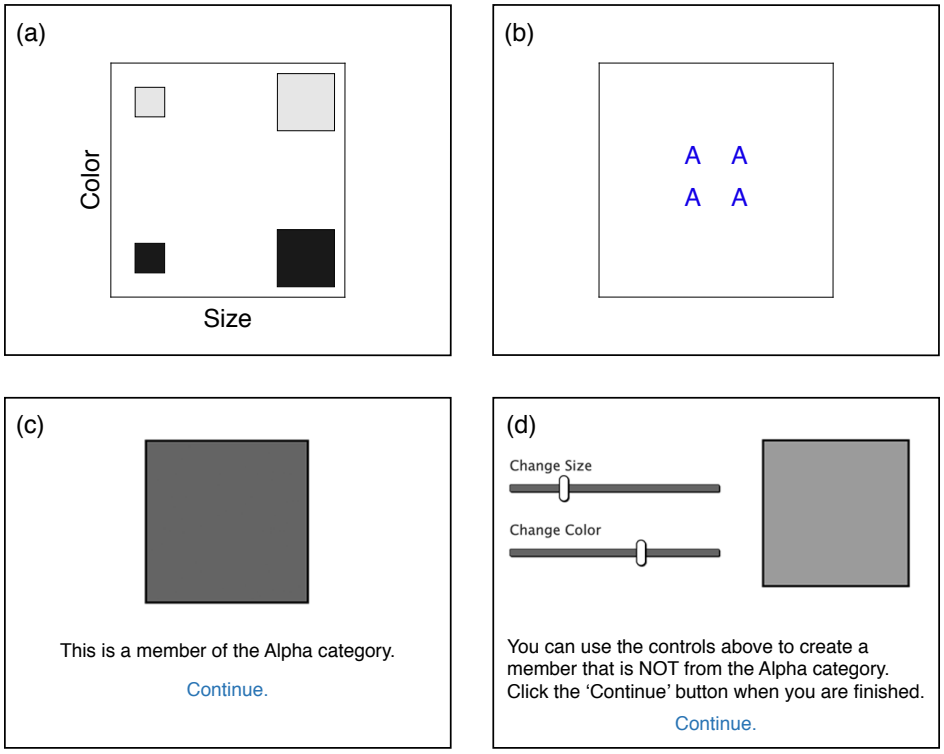

Figure 7: (a) Stimuli used for real-world experiment. Dimension and direction assignment (e.g., large to small or small to large) for color and size were counterbalanced over participants. (b) Category structure of Category A exemplars. (c) Example of trial in training phase. (d) Example of trial in generation phase.

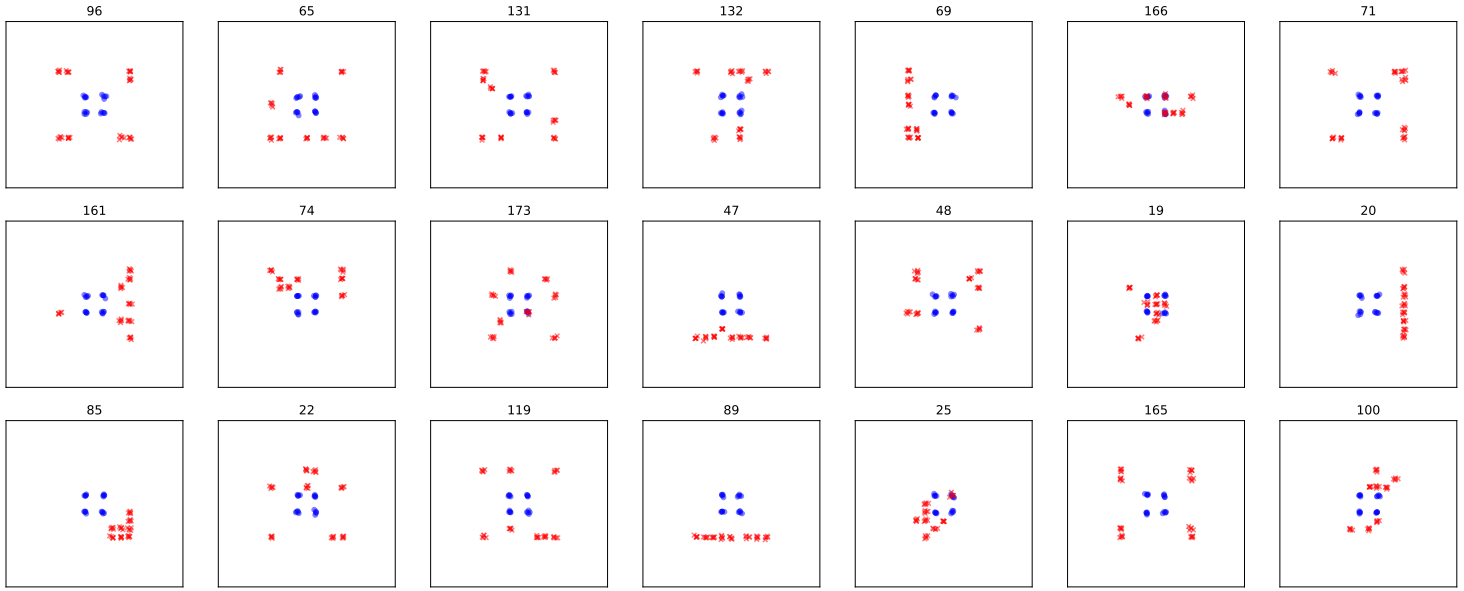

Figure 8: Real-world data set showing trained (blue) and generated (red) exemplars for each participant. Numbers above each panel indicate the unique identifier for each participant.

amount of noise added to increase the size of the data set and efficiency of the sampling algorithm. For this exercise, we will only fit the generated exemplars since the trained exemplars are fixed for every participant.

The next code section includes model specification. We assume a participant's representation of a category is prototype-like: that is the assumption that a category is encoded as a mixture of Gaussians, previously proposed by Rosseel (2002). Although we are using essentially the same models as in the previous section, there is one important difference to note here – we are now fitting the models to data from each individual participant. Consequently, for each variable, we are required to loop over each participant (for a given condition) on top of the loop over components. We implement this process by creating lists within lists for the relevant variables (lines 60 to 94; although note that the `mus_p` is not a nested list since we can sample the entire vector of k-component means directly from the multivariate Gaussian distribution, whereas this is currently not supported for the more complex `PyMC3` distributions such as the LKJ prior). Due to this extra step, readers may find that constructing the models themselves starts to take a little more time (e.g., where the models in earlier sections may be constructed within one second, with variables defined for each of 21 participants the models can now take more than a few seconds). We are fitting individual participants because (1) generated categories vary substantially from participant to participant (see Figure 8), and (2) we are interested in whether the generated exemplars from participants are best captured by a model with one, two, or three components.

In general, if the Category Bs overlap with Category A or are not well-fit by a single Gaussian, then we would expect a two or three component mixture model to perform better than the one component model. However, mixture models with more components are inherently more flexible, and consequently may end up performing worse compared to simpler models when measured on criteria that take into account model complexity (e.g., WAICs).

```
## Build models
for mi in range(n_models):
    print(f'Building Gaussian mixture model with {mi + 1} component(s).')
    gmm_all[mi].name = f'{mi + 1}-Group'  # Name each model for easy identification during comparison
    with gmm_all[mi]:
        k = mi+1
        if k>1:
            # Prior over component weights (only applicable with k>1)
            p = pm.Dirichlet('p', a=np.array([1.]*k), testval=np.ones(k)/k)

        # Prior over component means
        mus_p = [pm.MvNormal('mu_%d' % pid,
                             mu=pm.floatX(np.zeros(2)),
                             tau=pm.floatX(0.1 * np.eye(2)),
                             shape=(k,2))
                 for pi,pid in enumerate(data.keys())]

        # Cholesky decomposed LKJ prior over component covariance matrices
        packed_L = [[pm.LKJCholeskyCov('packed_L_%d_%d' % (pid,i),
                                       n=2,
                                       eta=2.,
                                       sd_dist=pm.HalfCauchy.dist(.01))
                     for i in range(k)]
                     for pi,pid in enumerate(data.keys())]

        # Unpack packed_L into full array
        L = [[pm.expand_packed_triangular(2, packed_L[pi][i])
            for i in range(k)]
            for pi,pid in enumerate(data.keys())]

        # Convert L to sigma for convenience
        sigma = [[pm.Deterministic('sigma_%d_%d' % (pid,i) ,L[pi][i].dot(L[pi][i].T))
                for i in range(k)]
                for pi,pid in enumerate(data.keys())]

        # Specify the likelihood
        if k>1:
            mvnl = [[pm.MvNormal.dist(mu=mus_p[pi][i],chol=L[pi][i])
                    for i in range(k)]
                    for pi in range(n_ppt)]
            Y_obs = [pm.Mixture('Y_obs_%d' % pid,w=p,
    comp_dists=mvnl[pi],observed=data[pid])
                     for pi,pid in enumerate(data.keys())]
        else:
            Y_obs = [pm.MvNormal('Y_obs_%d' % pid,mu=mus_p[pi][0],chol=L[pi][0],
                                 observed=data[pid])
```

```
                    for pi,pid in enumerate(data.keys())]
```

Like in Section 3, we call the `pm.sample()` function within a `with` statement separate from that used in constructing the model. Though this is by no means necessary, readers may find this practice (i.e., constructing all models first, then start the sampling process for each model at a later point) preferable compared to sampling immediately after model construction within the same context. Specifically, this can allow for easier debugging of the code (e.g., isolating the source of an error to either the model construction or sampling process), as well as a more organized way of running multiple sampling processes of the same models (e.g., readers may want to compared traces from the NUTS algorithm with those produced by the Metropolis-Hastings algorithm). Due to the long sampling times required here, we also save the resulting traces in `real_traces.p` so that they can be easily retrieved in the future without having to repeat the sampling processes.

```
## Run sampler
# Initialize list of traces
traces = []
draws = 2000
# Loop over each k-component model
for mi in range(n_models):
    print('Sampling from Model with {} component(s).'.format(mi+1))
    with gmm_all[mi]:
        step = pm.NUTS()

        #Start the sampler! For this larger dataset, it might take awhile -- allocate a few hours.
        traces += [pm.sample(draws, step=step,chains=4)]

        # Save the traces as a pickle so the hours spent running this are not lost. Warning: file can be very large
        with open('real_traces.p','wb') as f:
            pickle.dump(traces,f)
```

The subsequent code block extracts the relevant variables (`mus` and `sigmas`) for each participant and plots the 95% density regions accordingly for each k-component model. Results are visually summarised for each participant in Figures 9, 10, and 11 for the 1-, 2-,

and 3-component models respectively. In these figures, the learned exemplars are presented as blue markers and the generated exemplars as red markers. It appears that all models are capable of capturing the generated exemplars within the components' 95% density regions. Where they differ is in predicting how these exemplars should be categorized. Unsurprisingly, the 1-component model reveals a single component that includes most generated exemplars across all participants. While this may be adequate for exemplars with a row- or column-like structure (e.g., participants 20, 89, and 47), the 1-component model is inappropriate for widely-distributed exemplars (e.g., participants 48, 71, 95, and 165). Indeed, data from these participants appear more accurately captured by the components indicated by the 2- and 3-component models. Given that more participants appear to have produced widely-dispersed categories than row- or column-like categories, we might conclude that the 2- and 3-component models do an overall better job of capturing the data.

```
113 ## Draw the fitted plots (with 95% density regions)
114 for ki in range(len(traces)):
115     # Define the model index and number of components
116     my_k_i = ki # Model index
117     my_k = my_k_i+1 # Number of components
118     preamb_str = f'{my_k}-Group_' # Preamble of variable names
119     # Extract relevant trace object
120     trace = traces[my_k_i]
121     # Specify the tail-end number of sample iterations to include
122     num_samps_in = 500
123 
124     # Loop over number of models and make a separate plot each
125     for mi in range(n_models):
126         # Initialize subplots
127         fig, ax2 = plt.subplots(3,7,figsize=(25, 10))
128         for ip,pid in enumerate(data.keys()):
129             # Get reference to current individual subplot to make it easier to work
    with
130             ax = ax2[np.unravel_index(ip, (3,7), 'C')]
131 
132             # Get list of desired variable names
133             ms_str = [re.findall('mu_%d.*' % pid,string) for string in
    trace.varnames if len(re.findall('mu_%d.*' %pid,string))>0]
134             ss_str = [re.findall('sigma_%d.*' % pid,string) for string in
```

```
    trace.varnames if len(re.findall('sigma_%d.*'%pid,string))>0]
135             ss_str = [[f'{preamb_str}{tmp_str}'] for tmp_arr in ss_str for tmp_str
    in tmp_arr]
136
137             # Extract posteriors from trace
138             m_str = f'{preamb_str}{ms_str[0][0]}'
139             ms_post = np.array([np.array([trace[m_str][-num_samps_in:,i,0].mean(),
    trace[m_str][-num_samps_in:,i,1].mean()])
140                                 for i in range(my_k)])
141             ss_post =
    np.array([np.array([[trace[s_str[0]][-num_samps_in:,0,0].mean(),
    trace[s_str[0]][-num_samps_in:,0,1].mean()],
142
    [trace[s_str[0]][-num_samps_in:,1,0].mean(),
    trace[s_str[0]][-num_samps_in:,1,1].mean()]])
143                                 for s_str in ss_str])
144
145             if len(data[pid])>0:
146                 # Plot trained exemplars
147                 ax.scatter(data_all[pid][0:20, 0], data_all[pid][0:20, 1],s =
    20,color='blue',alpha=alpha,marker='o')
148                 # Plot generated exemplars
149                 ax.scatter(data_all[pid][20:, 0], data_all[pid][20:, 1],s =
    20,color='red',alpha=alpha,marker='x')
150
151             # Plot the ellipses
152             plot_ellipse(ax,ms_post,ss_post)
153             # Standardize axes
154             lim = 2.5
155             ax.get_xaxis().set_visible(False)
156             ax.get_yaxis().set_visible(False)
157             ax.set_ylim(-lim,lim)
158             ax.set_xlim(-lim,lim)
159             ax.set_title(pid)
160
161     # Save the figure for easy future access
162     plt.savefig('real_fit%d.pdf' % ki)
```

Quantitative analyses support our intuitions from visual inspection. Running a similar model comparison to before, the 3-component model performs the best, followed by the 2-component model.

```
164 ## Model comparison
165 #Convert model and trace into dictionary pairs
166 dict_pairs = dict(zip(gmm_all,traces))
```

```
#Perform WAIC comparison
compare = pm.compare(dict_pairs, ic='WAIC')
#Print comparison
print(compare)
```

This code block yields the following output:

```
            WAIC   pWAIC    dWAIC weight     SE    dSE var_warn
3-Group   558.79  991.14        0      1  76.28      0        1
2-Group  1522.53  755.75   963.74      0  87.59  57.36        1
1-Group  2499.55    79.1  1940.77      0  82.44  71.98        1
```

The impressive quantitative performance of the 3-component model is perhaps not surprising – increasing the number of components will always increase the absolute fit of the model. The predictive accuracy of this model was high enough such that its increased complexity did not reduce its relative performance as measured by WAIC. We challenge the reader to perform this exercise with models of even more components – at which point do you think the model's complexity will outweigh its benefit in absolute fit? In addition, it may be interesting to perform this analysis on an individual level to identify which participants are best fit by fewer components.

This section is one that will likely be the most computationally intensive. Given the increased number of variables to be estimated, the sampling process can take at least a few hours on a modern desktop machine. To address this, we move on to our final tutorial section where we describe how it is possible to enable `PyMC3` to take advantage of your machine's GPU (instead of relying solely on the CPU).

6 GPU Modeling

Over the last fifteen years, the performance of machine learning methods has drastically improved on a broad set of problems, ranging from object recognition (Krizhevsky, Sutskever, & Hinton, 2012) to machine translation (Vaswani et al., 2017). This jump in performance is large part due to a technological and engineering breakthrough. Although

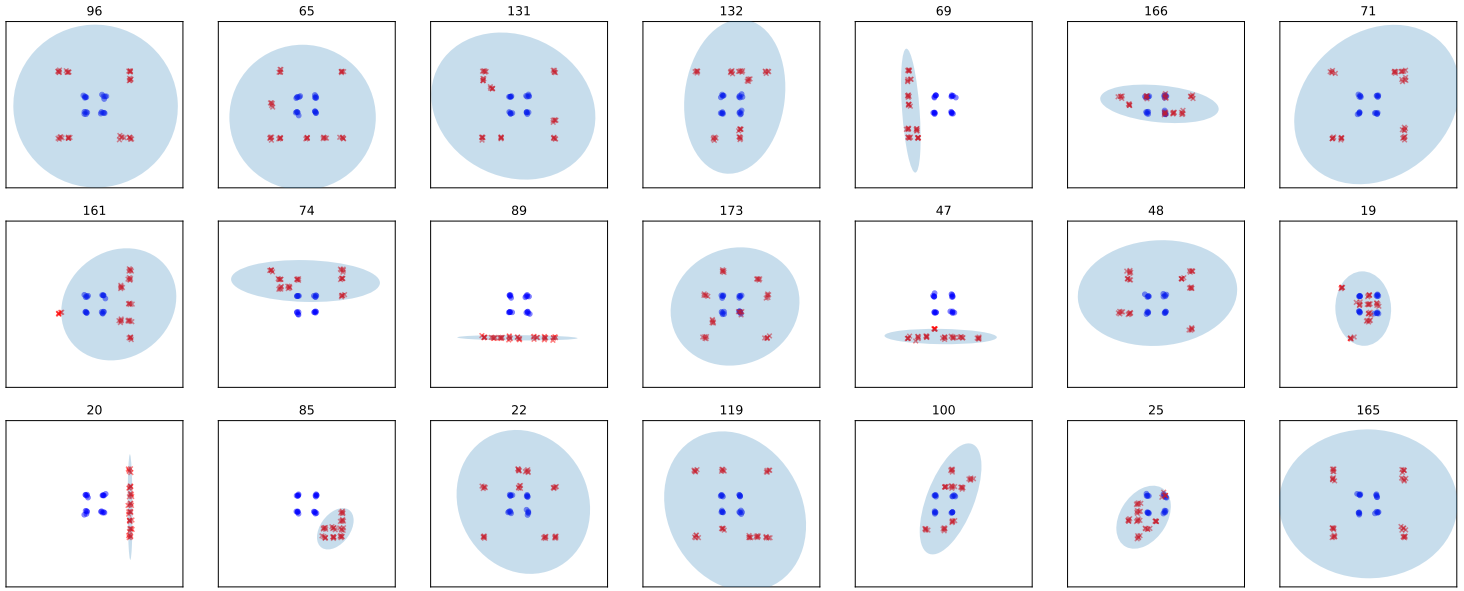

Figure 9: Fit results from 1-component model on real data. Blue markers indicate trained exemplars, red markers indicate generated exemplars. Colored ellipses indicate 95% density regions of each component.

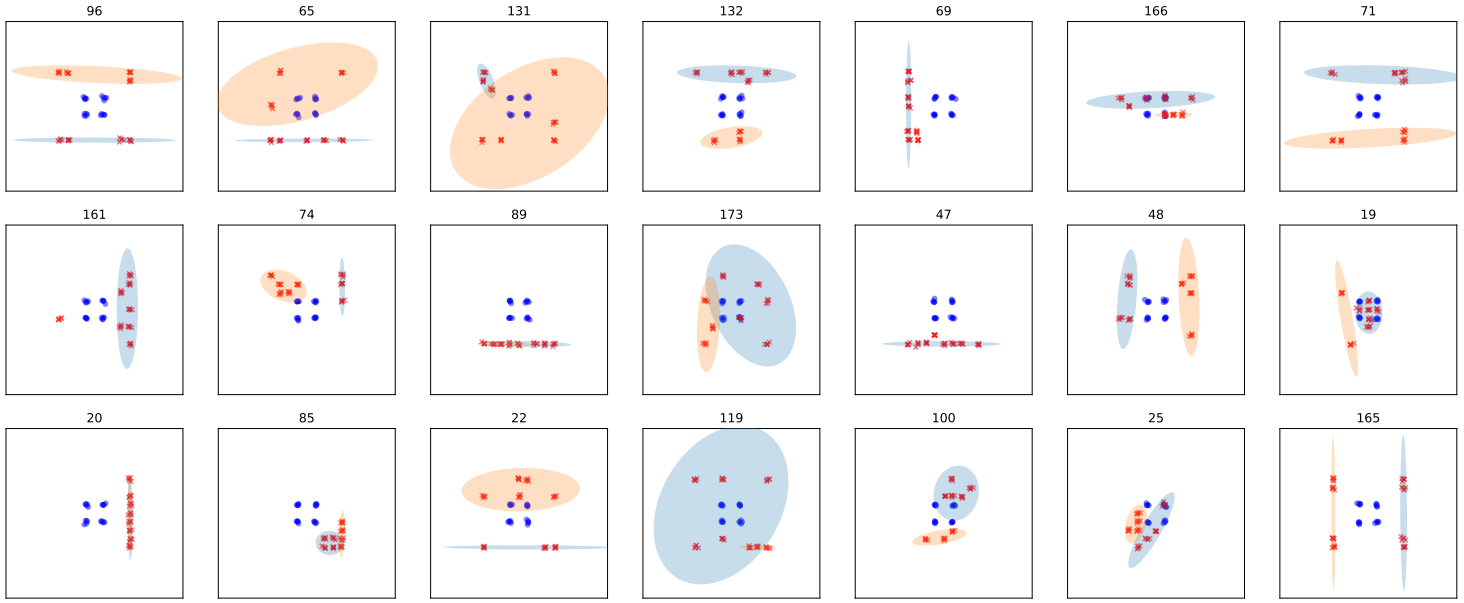

Figure 10: Fit results from 2-component model on real data. Blue markers indicate trained exemplars, red markers indicate generated exemplars. Colored ellipses indicate 95% density regions of each component.

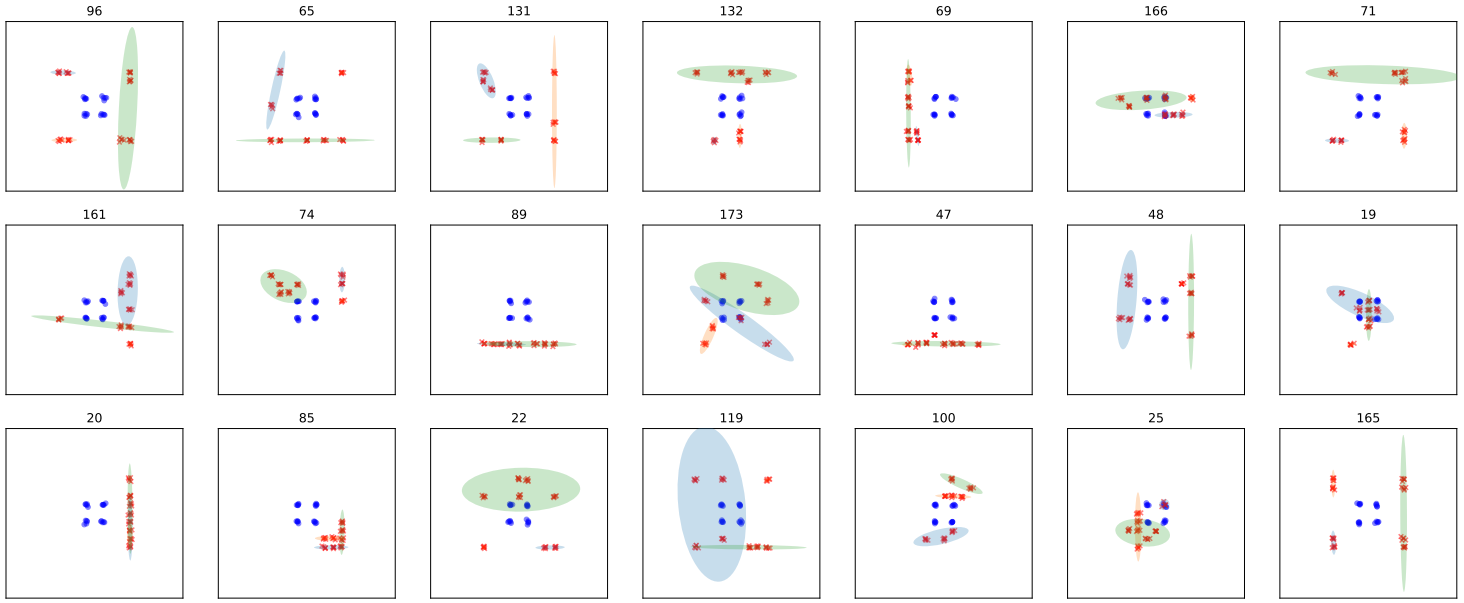

Figure 11: Fit results from 3-component model on real data. Blue markers indicate trained exemplars, red markers indicate generated exemplars. Colored ellipses indicate 95% density regions of each component.

many models have become more complex, this was not the main reason for the performance increase – at their core, many of the best performing models are larger versions of classic neural network models from the 1980s and early 1990s. It is largely due to recent advances in co-opting the fast parallelized computations on a Graphics Processing Unit (GPU) to train models, which is a component of computer hardware that specializes in rendering, or calculating the brightness value of each pixel in a visual display at each time point (elapsed time between time points is the inverse of the refresh rate) based on a higher-level description. The higher-level description is often a 3-D scene, which is composed of the position, shape, texture, and other properties of objects, the perspective of one or more observers, and one or more light sources. This is a problem in multi-view geometry, which ultimately results in many complex linear algebra operations over very large matrices (cite multi-view geometry textbook). Researchers realized that methods for training machine learning models can be reframed or modified to utilize the same methods that were already optimized to enable high-resolution, fast computer graphics. This allowed scientists to use vastly more data to train models with many, many more parameters than the models used two decades ago, and resulted in very large performance gains.

Although much of machine learning and optimization practice have been transformed by GPU-based methods, some areas, such as probabilistic programming methods, have only begun to take full advantage of this technological breakthrough (as of the writing of this article). It is only in the last few years that probabilistic programming languages have started to be built on optimization software that can be run on the GPU. As previously mentioned `PyMC3` is one of these languages, as it is built on the `Theano` optimization library. New probabilistic programming languages built on other GPU-compatible optimizers are being released as of the writing of this article, such as `Edward` (built on `TensorFlow`, which is incorporating the second version of `Edward` into its second version as `TensorFlow Probability`) and `Pyro` (built on `PyTorch`). Between these various languages, `PyMC3` is probably the most mature in terms of its richness of support for

various data analysis methods. The languages and libraries are under rapid development and so it is likely that they will provide the same or possibly more supported methods in the near future. Regardless, this section (and article) should still be helpful as many of the basic principles from this article transfer to other probabilistic programming languages.

While we have focused on the advantages of GPU-based optimization, it is worth noting three main disadvantages. The first disadvantage has to do with the architecture of the typical computer. Most general-purpose processing occurs on the Central Processing Unit (CPU) in conjunction with specialized fast memory stores, as well as the main memory store (RAM). To run optimization on the GPU, data, such as the program to be run and the training data, must be transferred from the CPU's memory stores to the GPU's memory stores. In standard computers, this is a high-bandwidth (can send large amounts of data at once), but slow process. As the GPU is limited in the complexity of computing functions available to it, depending on the complexity of optimization method, memory might need to be sent back and forth between the CPU and GPU several times (broadly, the more complex the probabilistic model, the more often data are transferred back and forth). Unless the data are large enough compared to the penalty in performance based on how often the optimization method needs to send data between the CPU and GPU, running the model on the CPU is likely to be faster. However, the increase in computation time for larger data sets grows much faster on the CPU than the GPU, which often results in a crossover point. Second, GPU-based programming is still early in its development. This results in two related issues: (a) it can be very difficult to set up your programming environment so that your optimization programs are running on the GPU, and (b) as of the writing of the article, running on the GPU was an afterthought, rather than writing the optimizer with the GPU in mind. This is the case for `PyMC3` using `Theano`, which does not take full advantage of the possible improvements from using a GPU (The PyMC Development Team, 2018). Lastly, current GPU-based programs are almost exclusively computed using the `CUDA` programming framework, which is proprietary to

NVIDIA. Thus, if your computer does not have a NVIDIA GPU card, you will not be able to run most GPU-enabled optimization.[6]

In this section, we demonstrate the relative sampling performance in PyMC3 between runs on a CPU and a GPU. The performance tests are run on a machine with 128 GB of RAM, a AMD Ryzen Threadripper 1950X 16-Core Processor CPU operating at 3622.327 MHz, and a NVIDIA Titan V GPU.

In order to enable PyMC3 (or more strictly speaking Theano) to utilize the GPU, certain attributes in Theano's configuration need to be specified. Detailed information on how to do this can be found on Theano's website[7] and on our README.md file..

6.1 Varying sample sizes

For our first set of performance tests, we run the same model we use in Section 5 with a series of $k = 1$, 2, 3, and 5 components (and $D = 2$), over an increasing number of participants. We increased the number of participants using a bootstrap method, where the trained and generated exemplars were separately sampled with replacement from the original 21 participants. Specifically, we ran the same model separately for data sets with n set to 21, 42, 105, 210, 1050, and 2100. This script is provided in the gpu_varyn.py file.

We obtained the sampling times presented in Figure 12 in log(seconds). Sampling times longer than two weeks are estimated based on the current sampling rate observed. Some computations on the CPU that took over two weeks were estimated to take longer than five months – for ease of interpretation these timings are not presented in the figure.

As expected, for smaller sample sizes (where $n \leq 105$), the CPU outperforms the GPU. This is consistent across all k except where $k = 1$, where the CPU is faster up to $n = 210$. Importantly, with increasing sample sizes, the corresponding decrease in speed appears linear for the GPU but is exponential for the CPU. Consequently, even for data

[6]Note that there is some support for optimization based on the OpenCL framework. However, as of the writing of this article, they tend to be much less stable, have less support, and be slower.

[7]http://deeplearning.net/software/theano/library/config.html

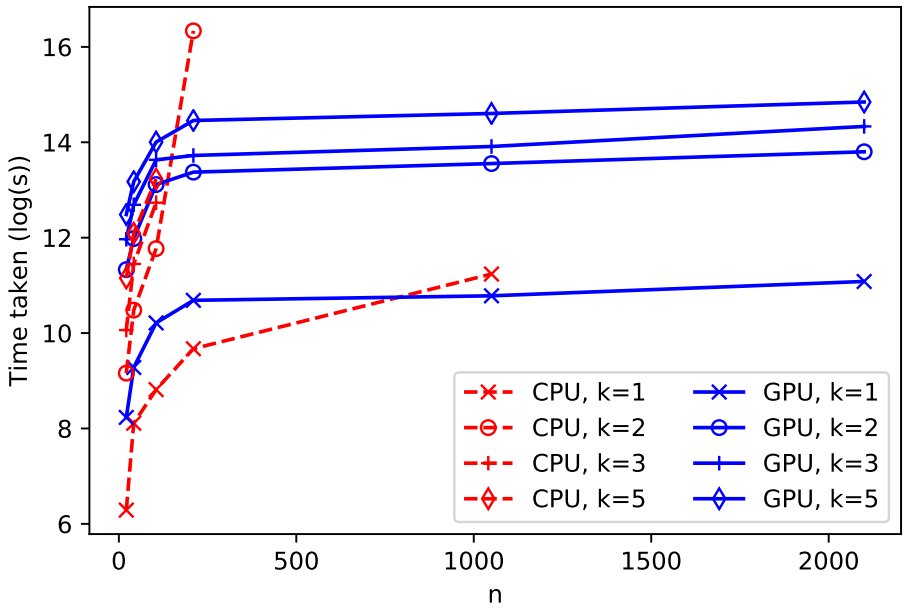

Figure 12: Total MCMC sampling time (in log(seconds)) for data sets of varying sizes. For all data sets used for simulations plotted in this figure, participant data were two continuous numbers ($D = 2$) and there were two mixture components ($k = 2$)

with very low dimensionality, the computational advantage offered by the GPU becomes quickly apparent with larger sample sizes.

6.2 Varying Dimensionality

Another way we can demonstrate the performance advantage afforded by the GPU is to increase the size of the data by its number of dimensions. Though we have only used data with only up to two dimensions in this tutorial, psychological data is by no means constrained to a small number of dimensions. For instance, data collected from questionnaires with hundreds of test items can be treated as data with high dimensionality.

The relevant Python script is contained in the `gpu_varyd.py` file. Here, we use a series of pre-generated data sets containing 100 samples from multivariate Gaussian distributions with the number of dimensions D ranging from 100 to 1000 (inclusive). The model used here is similar to that used in Section 4, with the exception that the number of

dimensions is not constrained to two. With each distribution we set the vector of means to a D-sized vector of ones and the covariance matrices to a D-by-D sized identity matrix. For these computations, we model the data separately with $k = 1$, 2 and 3.

The resulting sampling timings are presented in Figure 13. Consistent with previous analysis, the performance of the CPU exceeds that of the GPU across all k only with smaller data sets (in this case, $D < 500$). Similarly, while the speeds of the GPU computations for each k quickly plateau after $D > 200$, CPU speeds appear to continue on a linear trajectory (presented on a log scale, this is again an exponential trend).

Ultimately, while GPU acceleration is not guaranteed for all data sets (particularly those of smaller sizes), when applicable it can offer significant savings in computation time. With large data sets being increasingly easy to obtain (e.g., through crowd-sourcing platforms such as Amazon Mechanical Turk), the efficiency of processing such data becomes increasingly important. As demonstrated in this section, `PyMC3` is able to offer users increases in efficiency, not only through its accessible language, but also in its ability to leverage the resources of GPUs.

7 Discussion

`PyMC3` is a robust probabilistic programming language that allows for efficient Bayesian inference without a steep learning curve. As described in this tutorial, almost all of its most important aspects are straightforward in their syntax and function. The language is constantly being maintained with an active community of developers and users. With that said, official support for major releases of the `Theano` backend were discontinued in 2017. However, the developers of `PyMC3` have taken on the burden on providing limited support to `Theano` (specifically, to the extent that `PyMC3` depends on `Theano`). There is also relatively new development on the next iteration – `PyMC4` – that will be implemented using `TensorFlow` as the computational backend. However, `PyMC4` is still in its early stages (there

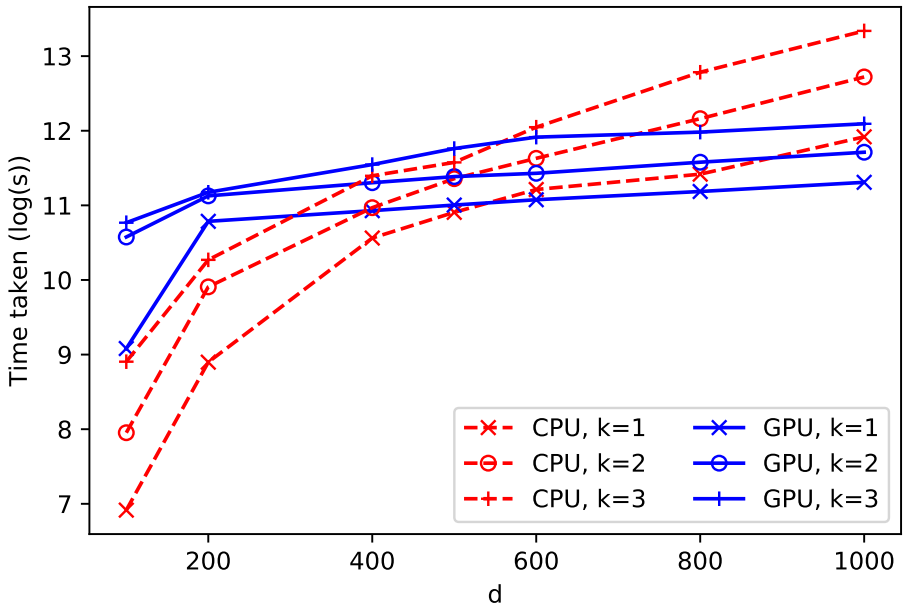

Figure 13: Total MCMC sampling time (in log(seconds)) for data sets of varying dimensionality.

is no stable release as of the writing of this article), and the developers of `PyMC3` assure users that the current project is still very much in active development. We find that the maturity of `PyMC3` as a probabilistic programming language should be appealing to most non-technical users, and the language should be remain relevant for the forseeable future.

While we have endeavored to cover the key elements of `PyMC3` that would be most useful for psychologists, `PyMC3` is able to offer much more than what is mentioned in this tutorial. For instance, one of its other key features is the ability to perform variational inference, which is a separate class of algorithms to MCMC that can allow for very fast posterior estimation under certain conditions[8]. In addition, `PyMC3` allows for straightforward construction of custom distributions or density functions – users are not only limited to the large library of built-in distributions and need not be forced to write custom distributions in a separate programming language (e.g., JAGS requires the construction of custom distributions and extensions to be written in C++). With that

[8]Interested readers may want to explore `Pyro` (Bingham et al., 2018), a separate Python package which specializes in variational inference techniques.

said, we believe that the essential functions covered here should easily prove to be a valuable set of tools for any psychological researcher.

References

Austerweil, J. L., Conaway, N., Liew, S. X., & Kurtz, K. J. (in preparation). Creating and learning something different: Similarity, contrast, and representativeness in categorization.

Bem, D. J. (2011). Feeling the future: experimental evidence for anomalous retroactive influences on cognition and affect. *Journal of personality and social psychology*, *100*(3), 407–425.

Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., ... Goodman, N. D. (2018). Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research*.

Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., ... Riddell, A. (2017). Stan: A probabilistic programming language. *Journal of statistical software*, *76*(1).

Ebersole, C. R., Atherton, O. E., Belanger, A. L., Skulborstad, H. M., Allen, J. M., Banks, J. B., ... others (2016). Many labs 3: Evaluating participant pool quality across the academic semester via replication. *Journal of Experimental Social Psychology*, *67*, 68–82.

Estimating the reproducibility of psychological science. (2015). *Science*, *349*(6251), aac4716.

Ferguson, T. S. (1973). A bayesian analysis of some nonparametric problems. *The annals of statistics*, 209–230.

Frank, M. C., Bergelson, E., Bergmann, C., Cristia, A., Floccia, C., Gervain, J., ... others (2017). A collaborative approach to infant research: Promoting reproducibility, best practices, and theory-building. *Infancy*, *22*(4), 421–435.

Gelfand, A. E., Hills, S. E., Racine-Poon, A., & Smith, A. F. (1990). Illustration of bayesian inference in normal data models using gibbs sampling. *Journal of the American Statistical Association*, *85*(412), 972–985.

Gelman, A. (2006). Prior distributions for variance parameters in hierarchical models (comment on article by browne and draper). *Bayesian analysis*, *1*(3), 515–534.

Gelman, A., Hwang, J., & Vehtari, A. (2014). Understanding predictive information criteria for bayesian models. *Statistics and computing*, *24*(6), 997–1016.

Gelman, A., & Rubin, D. B. (1992). Inference from iterative simulation using multiple sequences. *Statistical science*, *7*(4), 457–472.

Ghosal, S., Ghosh, J. K., & Ramamoorthi, R. V. (1999). Posterior consistency of Dirichlet mixtures in density estimation. *The Annals of Statistics*, *27*(1), 143–158.

Gronau, Q. F., & Wagenmakers, E.-J. (2019). Limitations of bayesian leave-one-out cross-validation for model selection. *Computational brain & behavior*, *2*(1), 1–11.

Hoffman, M. D., & Gelman, A. (2014). The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *Journal of Machine Learning Research*, *15*(1), 1593–1623.

JASP Team. (2018). *JASP (Version 0.9)[Computer software].* Retrieved from `https://jasp-stats.org/`

Jasra, A., Holmes, C. C., & Stephens, D. A. (2005). Markov chain monte carlo methods and the label switching problem in bayesian mixture modeling. *Statistical Science*, 50–67.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In *Advances in neural information processing systems 25* (pp. 1097–1105).

Kruschke, J. (2014). *Doing bayesian data analysis: A tutorial with r, jags, and stan.* Academic Press.

Lee, M. D., & Wagenmakers, E.-J. (2005). Bayesian statistical inference in psychology: Comment on trafimow (2003).

Lee, M. D., & Wagenmakers, E.-J. (2014). *Bayesian cognitive modeling: A practical course.* Cambridge university press.

Lewandowski, D., Kurowicka, D., & Joe, H. (2009). Generating random correlation matrices based on vines and extended onion method. *Journal of multivariate analysis*, *100*(9), 1989–2001.

Lunn, D., Spiegelhalter, D., Thomas, A., & Best, N. (2009). The bugs project: Evolution, critique and future directions. *Statistics in medicine*, *28*(25), 3049–3067.

Miller, J. W., & Harrison, M. T. (2013). A simple example of dirichlet process mixture inconsistency for the number of components. In *Advances in neural information processing systems* (pp. 199–206).

Nieuwenhuis, S., Forstmann, B. U., & Wagenmakers, E.-J. (2011). Erroneous analyses of interactions in neuroscience: a problem of significance. *Nature Neuroscience*, *14*(9), 1105–1107.

Nozek, B. A., Ebersole, C. R., DeHaven, A. C., & Mellor, D. T. (2018). The preregistration revolution. *Proceedings of the National Academy of Sciences*, *115*(11), 2600–2606.

Plummer, M. (2017). *Jags: Just another gibbs sampler.* Retrieved from `http://mcmc-jags.sourceforge.net`

Richardson, S., & Green, P. J. (1997). On bayesian analysis of mixtures with an unknown number of components (with discussion). *Journal of the Royal Statistical Society: series B (statistical methodology)*, *59*(4), 731–792.

Robert, C. P., & Castella, G. (2006). *Monte Carlo statistical methods.* Springer.

Rosseel, Y. (2002). Mixture models of categorization. *Journal of Mathematical Psychology*, *46*, 178–210.

Rubin, D. B. (1984). Bayesianly justifiable and relevant frequency calculations for the applies statistician. *The Annals of Statistics*, 1151–1172.

Salvatier, J., Wiecki, T. V., & Fonnesbeck, C. (2016). Probabilistic programming in python using pymc3. *PeerJ Computer Science*, *2*, e55.

Spiegelhalter, D. J., Best, N. G., Carlin, B. P., & Van Der Linde, A. (2002). Bayesian measures of model complexity and fit. *Journal of the Royal Statistical Society: Series*

B (Statistical Methodology), *64*(4), 583–639.

Stone, M. (1977). An asymptotic equivalence of choice of model by cross-validation and akaike's criterion. *Journal of the Royal Statistical Society. Series B (Methodological)*, 44–47.

The PyMC Development Team. (2018). *PyMC3 developer guide.* Retrieved 06-10-2019, from `https://docs.pymc.io/developer_guide.html`

Theano Development Team. (2016, May). Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, *abs/1605.02688*. Retrieved from `http://arxiv.org/abs/1605.02688`

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems 31* (pp. 5998–6008).

Vehtari, A., Gelman, A., & Gabry, J. (2017). Practical bayesian model evaluation using leave-one-out cross-validation and waic. *Statistics and Computing*, *27*(5), 1413–1432.

Vehtari, A., Simpson, D. P., Yao, Y., & Gelman, A. (2019). Limitations of âĂIJlimitations of bayesian leave-one-out cross-validation for model selectionâĂİ. *Computational Brain & Behavior*, *2*(1), 22–27.

Wagenmakers, E.-J., & Farrell, S. (2004). Aic model selection using akaike weights. *Psychonomic bulletin & review*, *11*(1), 192–196.

Wagenmakers, E.-J., Lee, M., Lodewyckx, T., & Iverson, G. J. (2008). Bayesian versus frequentist inference. In *Bayesian evaluation of informative hypotheses* (pp. 181–207). Springer.

Wagenmakers, E.-J., Marsman, M., Jamil, T., Ly, A., Verhagen, J., Love, J., ... Morey, R. D. (2018). Bayesian inference for psychology. Part I: Theoretical advantages and practical ramifications. *Psychonomic Bulletin & Review*, *25*, 35–57.

Wagenmakers, E.-J., Wetzels, R., Borsboom, D., & Van Der Maas, H. L. (2011). Why psychologists must change the way they analyze their data: the case of psi: comment

on bem (2011). *Journal of Personality and Social Psychology*, *100*(3), 426–432.

Watanabe, S. (2010). Asymptotic equivalence of bayes cross validation and widely applicable information criterion in singular learning theory. *Journal of Machine Learning Research*, *11*(Dec), 3571–3594.