Alan Agresti, Maria Kateri, Ranjini Grove, and Antonietta Mira

Py-Web Appendix of Foundations of Bayesian Statistics for Data Scientists, with R and Python

Contents

Py1

Getting Started with **`Python`** *for Bayesian Statistics*

Py1.1 Why Python?

Python is a widely used programming language for scientific computing, statistical analysis, and machine learning. In particular, it has become one of the predominant programming languages in data science, where it is used extensively for data manipulation, predictive modeling, and artificial intelligence. As this book is intended for readers with an interest in data science, Python provides a natural computational environment for implementing Bayesian methods.

Python is supported by a rich collection of libraries for numerical computation, visualization, and statistical modeling. Bayesian analysis is particularly well supported by packages such as `PyMC` for probabilistic programming, `NumPy` and `SciPy` for numerical computing, `ArviZ` for posterior analysis and model diagnostics, and `Matplotlib` for graphical visualization. Together, these libraries provide a flexible and powerful framework for Bayesian data analysis.

The purpose of this appendix is not to teach Python programming. Instead, it introduces the software environment used throughout this book and provides the information required to reproduce the examples and analyses. We focus on installing the necessary software, creating an isolated Python environment, and executing the accompanying code. Only the Python concepts needed to understand the examples are introduced; readers seeking a comprehensive introduction to Python are referred to the many books and online resources available.

Py1.2 Installing Python and Anaconda

To run the examples in this book, a recent installation of Python is required. We recommend installing the `Anaconda` distribution, which provides Python together with the `conda` package and environment manager. Anaconda simplifies the installation and maintenance of scientific software and makes it easy to create isolated environments for different projects.

Anaconda is available for Windows, macOS, and Linux and can be downloaded free of charge from the official website at `https://www.anaconda.com/download`. After downloading the appropriate installer for your operating system, the installation can usually be completed by accepting the default settings.

Once the installation has finished, open a terminal (or the Anaconda Prompt under Windows) and verify that `Python` and `conda` are available by executing:

```
python --version
conda --version
```

If both commands return the installed version numbers, the installation has been completed successfully.

In the following section, we create a dedicated PyMC environment containing all packages required for the examples in this book that are implemented using PyMC. Using a dedicated environment keeps the software for this book separate from other Python projects and contributes to a reproducible computational workflow.

Py1.3 PyMC: A Flexible Framework for Bayesian Modelling

The examples presented throughout the book are primarily implemented using bambi, which provides a convenient formula interface for Bayesian regression models similar to that of brms in R. Internally, however, bambi relies on PyMC to perform Bayesian inference and posterior sampling.

While bambi is well suited for the regression models considered in this book, it offers only a subset of the modelling capabilities available in PyMC. For this reason, the present appendix also illustrates selected examples using PyMC. Although these implementations generally require more explicit code, they provide greater flexibility in model specification, allowing users to define custom probability models, prior distributions, likelihoods, and sampling strategies. Familiarity with PyMC therefore provides a useful foundation for readers wishing to develop Bayesian models beyond those supported by bambi's formula interface.

PyMC is an open-source Python library for Bayesian modelling and probabilistic programming. It allows users to specify statistical models using probability distributions and automatically performs Bayesian inference using modern computational methods such as Markov chain Monte Carlo (MCMC) sampling and variational inference. In this book, PyMC is used to formulate and estimate Bayesian models, generate posterior distributions, and perform model diagnostics and visualization.

To ensure reproducibility and computational stability, it is recommended to run PyMC in a dedicated Python environment rather than in the default Anaconda (base) environment. This approach isolates the dependencies required for Bayesian computation from those used by other statistical and data analysis workflows.

We first open the Anaconda Prompt and create a new environment:

```
conda create -n pymc_env python=3.11
```

The environment name is pymc_env. Python version 3.11 is a suitable choice for current versions of PyMC and its associated scientific computing packages. We confirm the installation when prompted.

We activate the newly created environment:

```
conda activate pymc_env
```

The command prompt should now indicate that the new environment is active:

```
(pymc_env) C:\Users\your_user>
```

We install next PyMC and the main packages required for Bayesian analysis:

```
conda install -c conda-forge pymc arviz jupyterlab ipykernel
```

This installs:

- PyMC: a probabilistic programming framework for Bayesian modelling;

- `PyTensor`: the computational backend used by `PyMC` for numerical operations;
- `ArviZ`: tools for posterior analysis, diagnostics, and visualization;
- `Jupyter kernel support`: allows this environment to be used directly from `JupyterLab`.

To use the newly created environment from within `JupyterLab`, it must be registered as a Jupyter kernel. A kernel is the computational environment in which a notebook is executed. We register the `PyMC` environment using:

```
python -m ipykernel install --user \
--name pymc_env \
--display-name "Python (PyMC)"
```

After this step, `JupyterLab` will provide a kernel option named:

```
Python (PyMC)
```

We start `JupyterLab` from the activated environment:

```
jupyter lab
```

When creating a notebook for the examples in this book, we select the kernel:

```
Python (PyMC)
```

The installation can be verified within a Jupyter notebook using:

```
import pymc as pm
import arviz as az
import numpy as np

print("PyMC:", pm.__version__)
print("NumPy:", np.__version__)
```

If these commands execute without errors, the `PyMC` environment is correctly configured and ready for use.

Maintaining a separate `PyMC` environment reduces the risk of dependency conflicts between Bayesian modelling tools and other scientific Python packages. Since `PyMC` relies on compatible versions of packages such as `NumPy`, `SciPy`, `PyTensor`, and `ArviZ`, isolating these dependencies helps ensure that the analyses presented in this book remain reproducible.

Py1.4 Reproducible Computing

A fundamental principle of scientific computing is that computational analyses should be reproducible. In other words, given the same data, software, and computational environment, it should be possible to obtain the same numerical results. While this goal cannot always be achieved exactly across different computer systems or software versions, adopting a few simple practices greatly improves the reproducibility of statistical analyses.

Throughout this appendix, we recommend performing all Bayesian analyses within the dedicated `PyMC` environment created in the previous section. Using an isolated environment ensures that the versions of `PyMC` and its supporting libraries remain consistent and avoids unexpected changes caused by updates to unrelated software.

Another important aspect of reproducibility is the use of random number seeds. Bayesian

inference algorithms, including MCMC methods, rely on pseudo-random number generation. Consequently, repeated executions of the same program may produce different posterior samples, although these differences are typically negligible from a statistical perspective. By specifying a fixed random seed, the same sequence of pseudo-random numbers is generated, allowing analyses to be reproduced exactly.

For this reason, examples in this book specify a random seed whenever stochastic simulation is involved. We recommend adopting the same practice in your own analyses, particularly when conducting simulation studies, reproducing published results, or collaborating with others.

Finally, it is good practice to document the versions of the software packages used in an analysis. As statistical software continues to evolve, package updates may introduce new functionality, improve computational performance, or modify numerical algorithms. Recording software versions therefore facilitates the reproduction of analyses at a later date.

Py2

Python for Inference for Proportions

Py2.1 Bayes Factor for a Proportion Hypothesis Testing

In this section, we provide Python code to reproduce the analysis performed by the `proportionBF()` function from the `R BayesFactor` package. Since Python does not currently provide a direct analogue of this function, we implement the corresponding calculation by defining a suitable fucntion below. We apply the code to the same example used in `R` in Appendix Section A.2.1 of the book, allowing the analysis to be reproduced in Python while following the same Bayesian approach.

```
import numpy as np
from scipy.integrate import quad
from scipy.special import expit,logit
from scipy.stats import logistic

def proportionBF(y,N,p=0.5,r=0.5,nullInterval=None):
    l0=logit(p)
    lik=lambda l: expit(l)**y*(1-expit(l))**(N-y)
    pri=lambda l: logistic.pdf(l,loc=l0,scale=r)
    def ml(a,b):
        lo=-np.inf if a==0 else logit(a); hi=np.inf if b==1 else logit(b)
        pp=logistic.cdf(hi,loc=l0,scale=r)-logistic.cdf(lo,loc=l0,scale=r)
        return quad(lambda l:lik(l)*pri(l),lo,hi)[0]/pp
    m0=p**y*(1-p)**(N-y)
    if nullInterval is None:return ml(0,1)/m0
    a,b=nullInterval
    pa=logistic.cdf(logit(a),loc=l0,scale=r) if a>0 else 0
    pb=1-logistic.cdf(logit(b),loc=l0,scale=r) if b<1 else 0
    mla=ml(0,a) if a>0 else 0
    mlb=ml(b,1) if b<1 else 0
    return {"BF_interval":ml(a,b)/m0,"BF_complement":(mla*pa+mlb*pb)/(pa+pb)/m0}

# Implementation for the example:
bf=proportionBF(63,111,p=0.5,r=0.5,nullInterval=(0,0.5))
print(f"BF(p<0.5 vs p=0.5) = {bf['BF_interval']:.7f}")
print(f"BF(p>0.5 vs p=0.5) = {bf['BF_complement']:.7f}")

# Output:
BF(p<0.5 vs p=0.5) = 0.1000634
BF(p>0.5 vs p=0.5) = 1.1007848
```

Py3

Py for Bayesian Inference of Means

Py3.1 Plotting an Empirical Cumulative Density Function in Python

Figure B.3.1 (right) of the book's Python Appendix shows the plot of the empirical *cdf* (*ecdf*) of a simulated sample of $n = 10$ observations from a $\mathcal{N}(100, 16^2)$ distribution, as produced by standard fucntion, such as the ecdfplot function from the seaborn package. The corresponding code follows.

```
[15]: import numpy as np
      import matplotlib.pyplot as plt
      import seaborn as sns
      from scipy.stats import norm

      # Generate 10 random observations from a normal distribution, mean=100, std=16:
           np.random.seed(0)               # set seed for reproducibility
      y = np.random.normal(loc=100, scale=16, size=10)

      ax = plt.subplot() # Set up the figure and axis

      # Plot the empirical cdf:
      sns.ecdfplot(y, ax=ax, marker='o', ms=4, label='empirical cdf', color='blue')
      # Plot of the cdf of a normal distribution (mean=100, std=16):
      x = np.linspace(50, 150, 1000) # x values for the cdf curve, extending from 50 to 150
      ax.plot(x, norm.cdf(x, 100, 16), label='true cdf', color='red')

      ax.set_xlabel('y')   # add labels
      ax.set_ylabel('cdf')
      ax.legend()          # display the legend
      plt.show()           # Figure B.2 (right) in book, Section B.3.2
```

However, this approach misrepresents the *ecdf* as a step graph that fails to define a proper function of the observed values y, since the vertical segments assign multiple values to a single y (see Figure B.2 (right) in book). Python code for providing the correct graph for the *ecdf*, as shown in Figure Figure B.2 (right) in book, is provided in the book (Section B.3.2).

Py3.2 Comparing Two Means: Frequentist Approach

The Bayesian comparison of two means in Python is illustrated in Section B.3.5, by comparing the mean housework hours per week for men and women, discussed in Section 3.4.3 using R. Here, we provide the Python code for the frequentist analysis (using the stats module from the scipy library):

```
import pandas as pd
import numpy as np
from scipy import stats
url="http://bayes4ds.rwth-aachen.de/data/Household.dat"  # load data from book's URL
Household = pd.read_csv(url,delimiter=r"\s+")  # sep="\t": does not read two columns
print(list(Household.columns)) # check the variables in data frame
# Output: ['gender', 'time']
```

The Python code for the frequentist analysis (using the `stats` module from the `scipy` library) is provided in the book's web appendix. It includes the computation of the 95% confidence interval for $\mu_1 - \mu_2$, the difference between the population mean housework times for women (μ_1) and men (μ_2), along with relevant summary statistics and the results of the two-sample t-test.

```
summary = Household.groupby('gender')['time'].describe() # summarizes
print("Summary of 'time' by gender:")                    #   'time' variable
print(summary)                                           #   by 'gender'

y1 = Household['time'][Household['gender'] == 1]  # 'time' data for women
y2 = Household['time'][Household['gender'] == 0]  # 'time' data for men
n1, n2 = len(y1), len(y2)                      # n1=len(y1) and n2=len(y2)
test_val = stats.ttest_ind(y1, y2, equal_var=True) # indep. two-sample t-test

mean_diff = np.mean(y1)-np.mean(y2)  # derive limits of 95% CI for mu1-mu2
s = np.sqrt((np.var(y1, ddof=1)*(n1-1)+np.var(y2, ddof=1)*(n2-1)) / (n1+n2-2))
ci_low = mean_diff - stats.t.ppf(0.975, n1+n2-2) * s * np.sqrt(1/n1+1/n2)
ci_up = mean_diff + stats.t.ppf(0.975, n1+n2-2) * s * np.sqrt(1/n1+1/n2)

print(f"t-statistic={float(test_val[0])}, p-value={float(test_val[1])},
      df={round(test_val.df)}")
print(f"95% Confidence Interval for mu1-mu2: ({ci_low}, {ci_up})")

# Output: Summary of 'time' by gender:
        count       mean        std  min  25%  50%   75%    max
gender
0       582.0   8.309278   9.437386  0.0  3.0  6.0  10.0   72.0
1       694.0  11.874640  12.752392  0.0  4.0  8.0  15.0  100.0
t-statistic=5.583346613034059, p-value=2.8804807547765758e-08, df=1274
95% Confidence Interval for mu1-mu2: (2.3125957215595916, 4.81812711631438)
```

In the code above we derived the 95% confidence interval for $\mu_1 - \mu_2$ manually. Alternatively, the same analysis can be performed using the relevant function from the `statsmodels` library, which also provides the 95% confidence interval for $\mu_1 - \mu_2$, along with test statistics and summary output (not shown here):

```
import statsmodels.stats.api as sms
ci = sms.CompareMeans(sms.DescrStatsW(y1), sms.DescrStatsW(y2))
print(ci.tconfint_diff(usevar='pooled')) # 95% CI
```

Py3.3 Baysian Comparison of Two Means: Bayesian Approach

In Section B.3.5, we discussed for the household example the Bayesian comparison of mean housework hours per week for men and women, adapting the Monte Carlo method for a single mean, explained in Section B.3.3. Here, we show the code for analysing the same example in `PyMC`.

```
import pandas as pd
import numpy as np
import pymc as pm
import arviz as az

time  = Household['time'].values    # extract variables
Men   = Household['Men'].values
Women = Household['Women'].values

with pm.Model() as model:   # Model Specification
    b_men = pm.Normal('b_men', mu=10, sigma=4)     # priors for the coefficients
    b_women = pm.Normal('b_women', mu=10, sigma=4)
    sigma = pm.Exponential('sigma', 1/0.046)       # prior for the standard deviation

    mu = b_men * Men + b_women * Women  # expected value of outcome

    # likelihood (sampling distr.) of observations:
    Y_obs = pm.Normal('Y_obs', mu=mu, sigma=sigma, observed=time)

    # draw posterior samples for inference:
    fitted = pm.sample(2000, return_inferencedata=True)

print(az.summary(fitted))   # summarize the posterior distributions

# extract posterior samples for the coefficients:
b_men_samples = fitted.posterior['b_men'].values.flatten()
b_women_samples = fitted.posterior['b_women'].values.flatten()

# difference between the posterior samples of the coefficients:
diff_means = b_women_samples - b_men_samples

post_prob = np.mean(diff_means <= 0)  # posterior prob. that mu_women <= mu_men
print("P(b_women <= b_men) =", post_prob)

ci_low, ci_up = np.percentile(diff_means, [2.5, 97.5]) # 95% posterior interval
print(f"95% credible interval for b_women - b_men: ({ci_low}, {ci_up})")

# output:
Sampling 4 chains for 1_000 tune and 2_000 draw iterations (4_000 + 8_000 draws total)
took 31 seconds.
          mean     sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd  ess_bulk  \
b_men     8.341  0.432   7.553    9.179      0.004    0.005   11861.0
b_women  11.859  0.395  11.144   12.612      0.003    0.004   12855.0
sigma    10.472  0.185  10.122   10.806      0.002    0.002   12160.0

         ess_tail  r_hat
b_men      6226.0    1.0
b_women    6664.0    1.0
sigma      6292.0    1.0
P(b_women <= b_men) = 0.0
95% credible interval for b_women - b_men: (2.369724281345791, 4.676679314044213)
```

The histogram of the sample from the posterior means differences can be constructed as follows. The derived plot is provided in Figure Py3.1.

```
# Plot histogram of the differences
plt.hist(diff_means, bins=50, edgecolor='k', alpha=0.7)
plt.xlabel('Difference in Means (Women - Men)')
plt.ylabel('frequency')
# plt.title('Posterior Distribution of Difference in Means')
 plt.show()
```

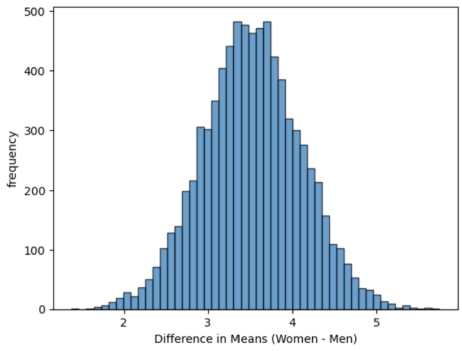

FIGURE Py3.1: Histogram of simulated values from the posterior distribution of $\mu_1 - \mu_2$.

Py3.4 Comparing Means Permitting Unequal Variances

We now re-visit the example from Section **??** that compares means while permitting the variances to be unequal, conducting the analysis in `Python`. The book's web appendix provides the frequentist analysis. We obtain the Bayesian posterior interval using the Monte Carlo procedure with a diffuse inverse gamma prior distribution for the variances, described in Section **??**, as shown below. Since the structure and usage of `Python` commands should now be familiar, we will omit the execution count numbers (e.g., `[34]`) that appear beside code blocks in this and subsequent examples.

```
import numpy as np
from scipy import stats

np.random.seed(0)               # set seed for reproducibility

m1, s1, n1 = 585.2, 89.6, 32    # summary statistics
m2, s2, n2 = 533.7, 65.3, 32

z1 = np.random.normal(size=n1)          # simulate data
z2 = np.random.normal(size=n2)
y1 = stats.zscore(z1, ddof=1)*s1 + m1   # ddof=1 standardizes using "n-1"
y2 = stats.zscore(z2, ddof=1)*s2 + m2   # (without it, "n" is used)

print("mean(y1):", round(np.mean(y1), 1)) # display sample stats
print("sd(y1):", round(np.std(y1), 1))
print("mean(y2):", round(np.mean(y2), 1))
print("sd(y2):", round(np.std(y2), 1))   # np.std(y2, ddof=1) devides by n-1

# Welch's t-test (does not assume equal variances)
t_stat1, p_val1 = stats.ttest_ind(y1, y2, equal_var=False)
# Calculate Welch degrees of freedom manually
s1_sq, s2_sq = np.var(y1, ddof=1)/n1, np.var(y2, ddof=1)/n2
df_welch = (s1_sq + s2_sq)**2 / ((s1_sq**2)/(n1 - 1) + (s2_sq**2)/(n2 - 1))
mean_diff = np.mean(y1) - np.mean(y2)
se_diff = np.sqrt(s1_sq + s2_sq)

# Confidence interval
ci_low, ci_high = stats.t.interval(0.95, df=df_welch, loc=mean_diff, scale=se_diff)

# Print results
print("Welch's t-test:")
```

```
print("t = {:.4f}".format(t_stat1))
print("df = {:.3f}".format(df_welch))
print("p-value = {:.5f}".format(p_val1))
print("95% confidence interval: ({:.5f}, {:.5f})".format(ci_low, ci_high))
```

Assuming equal variances for men and women, the respective code follows:

```
# --- Student's t-test (assumes equal variances) ---
t_stat_equal, p_val_equal = stats.ttest_ind(y1, y2, equal_var=True)
df_equal = n1 + n2 - 2
# Pooled variance and standard error
var1 = np.var(y1, ddof=1)
var2 = np.var(y2, ddof=1)
pooled_var = ((n1 - 1)*var1 + (n2 - 1)*var2) / df_equal
se_equal = np.sqrt(pooled_var * (1/n1 + 1/n2))
ci_equal = stats.t.interval(0.95, df=df_equal, loc=mean_diff, scale=se_equal)

print("\nStudent's t-test (equal variances):")
print(f"t = {t_stat_equal:.4f}, df = {df_equal}, p-value = {p_val_equal:.5f}")
print("95% confidence interval: ({:.5f}, {:.5f})".format(*ci_equal))
# -------- Output:
mean(y1): 585.2, sd(y1): 88.2, mean(y2): 533.7, sd(y2): 64.3
Welch's t-test:
t = 2.6276, df = 56.685, p-value = 0.01104, 95% confidence interval: (12.24834, 90.75166)

Student's t-test (equal variances):
t = 2.6276, df = 62, p-value = 0.01082, 95% confidence interval: (12.32154, 90.67846)
```

Py4

Python for Bayesian Inference in Linear Models

Py4.1 Posterior Predictive Distribution and Its Use for Model Checking

For the `Students` data, using responses on x = high school GPA and y = college GPA, we constructed in Section 4.4.6 a 95% prediction interval for college GPA at the mean high school GPA value of 3.3. We now construct such a prediction interval in `Python`:

```
import pandas as pd
import statsmodels.formula.api as smf
import numpy as np

Students = pd.read_csv("http://stat4ds.rwth-aachen.de/data/Students.dat", delimiter=r"\s+")
model = smf.ols('cogpa ~ hsgpa', data=Students).fit()

newdata = pd.DataFrame({'hsgpa': [3.3]})

pred = model.get_prediction(newdata)           # prediction with
pred_summary = pred.summary_frame(alpha=0.05)  # 95% prediction interval
print(pred_summary[['mean', 'obs_ci_lower', 'obs_ci_upper']])
      mean  obs_ci_lower  obs_ci_upper
 0  3.45156      2.764314      4.138805
```

This is a frequentist prediction interval, for which an explicit formula exists, and obviously coincides with the respective frequentist pediction interval in `R`(see Secton 4.4.6, p. 157 of the book). Considering Bayesian inference with improper priors, the posterior predictive intervals are close to the frequentist ones for highly disperse priors.

Py4.2 Implementation of Lasso via `glmnet` in `Python`

The same dataset `Students` was used in Section 4.5.2 to illustrate a penalized regression example, fitted in `R` using `glmnet`. In Section B.4.7 of the book's `Python` Appendix, we reproduced the Bayesian linear model fit using `bambi` and the frequentist lasso using `LassoCV` of the `scikit-learn` package. Alternatively, there exists a `glmnet` version for Python. However, it does not run on Windows. Instead, we can use the original `R` implementation, `r-glmnet`, by calling it from `Python` through the `rpy2` bridge. It is important to know, that (for the moment) `rpy2` and the `r-glmnet` package are not compatible with `Python` versions higher than 3.12. For this, we install these packages in a dedicated Anaconda environment using `Python` 3.10. This ensures compatibility between `rpy2`, `R`, and the required libraries. The environment can be created with the following commands in `Anaconda Prompt`:

```
conda create -n rlasso-env python=3.10
conda activate rlasso-env
conda install -c conda-forge r-base r-glmnet rpy2
```

In the sequel we install the Jupyter kernel support and we register the environment as a kernel:

```
conda install ipykernel
python -m ipykernel install --user --name rlasso-env --display-name "Python (rlasso-env)"
```

In order to see the new kernel (Pthon(rlasso-env), we need to close the Jupyter Notebook/JupyterLab completely and start it again.

Since `rpy2` bridges `Python` and `R`, it may occasionally encounter configuration issues. Because it depends on both Python and R installations, configuration issues (e.g., incompatible versions, incorrect environment variables, or missing packages). If problems arise, check that the `Python` environment, the `R` installation, and `R_HOME` are correctly configured.

In the sequel, we can repeat the glmnet analysis of Section 4.5.2 in Python, as shown below, by selecting first the new constructed kernel.

```
import pandas as pd
import numpy as np
from rpy2.robjects import pandas2ri
import rpy2.robjects as robjects
from rpy2.robjects.conversion import localconverter
from rpy2.robjects import r

# Read the data file from the same folder as the Jupyter notebook:
Students = pd.read_csv("Students.dat", sep=r"\s+")
Students.head()

X = Students[["gender", "age", "hsgpa", "abor", "dhome", "dres",
              "tv", "sport", "news", "aids", "veg", "ideol", "relig", "affirm"]]
y = Students["cogpa"]

with localconverter(robjects.default_converter + pandas2ri.converter):
   X_r = robjects.conversion.py2rpy(X)
   y_r = robjects.conversion.py2rpy(y)

X_r = r['as.matrix'](X_r)

glmnet = importr('glmnet')
cv_glmnet = robjects.r['cv.glmnet']

fit = cv_glmnet(X_r, y_r, alpha=1, nfolds=5)

# Extract lambda.min and lambda.1se
lambda_min = fit.rx2('lambda.min')[0]
lambda_1se = fit.rx2('lambda.1se')[0]

print(f"lambda.min = {lambda_min}")
print(f"lambda.1se = {lambda_1se}")

# Extract coefficients at lambda.min
coef = glmnet.coef_glmnet(fit.rx2('glmnet.fit'), s=lambda_min)  #s=lambda_1se
print(coef)
```

Py5

Python for Bayesian Inference in Generalized Linear Models

Py5.1 GLMs for Normal Distributed Responses in PyMC

In Section 5.2.2, we modeled the selling price (in thousands of dollars) of 100 homes in terms of the size of the house and whether the house is new or not, treating the response variable as either normally or gamma distributed. In case of normal distributed selling prices, the variance is assumed to be constant while for gamma distributred, the standard deviation is proportional to the mean. In the book's Python-Appendix, we reproduced the analysis for normal responses using bambi. Here, we provide the same analysis in pymc.

```
import numpy as np
import pandas as pd
import pymc as pm
import arviz as az
import matplotlib.pyplot as plt

# Read data
houses = pd.read_csv("http://bayes4ds.rwth-aachen.de/data/Houses.dat",
    delimiter=r"\s+")

# Center size
houses["size2"] = houses["size"] - houses["size"].mean()

plt.hist(houses["price"], bins=30)  # Histogram of response (s. Figure xx)

x = houses["size2"].values  # Extract variables
new = houses["new"].values
y = houses["price"].values

with pm.Model() as model_norm:
    # Priors
    beta0 = pm.Normal("Intercept", mu=0, sigma=1000)
    beta1 = pm.Normal("size2", mu=0, sigma=1000)
    beta2 = pm.Normal("new", mu=0, sigma=1000)
    beta3 = pm.Normal("size2:new", mu=0, sigma=1000)

    sigma = pm.Exponential("sigma", lam=0.001)

    # Linear predictor
    mu = (beta0 + beta1*x + beta2*new + beta3*x*new)

    # Likelihood
    price = pm.Normal("price", mu=mu, sigma=sigma, observed=y)

    # Posterior sampling
    idata = pm.sample(draws=2000, tune=1000, target_accept=0.9,
        random_seed=123, return_inferencedata=True)
```

The histogram of the selling prices is provided in Figure Py5.1 while information of the posterior samples drawn by `pm.sample()` above is given in Figure Py5.2.

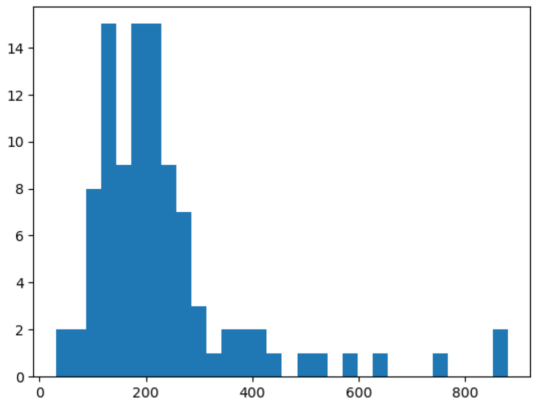

FIGURE Py5.1: Histogram of the selling prices (in thousand dollars) of appartments in Florida.

Progress	Draws	Divergences	Step size	Grad evals	Sampling Speed	Elapsed	Remaining
————————————	3000	0	0.57	7	741.30 draws/s	0:00:04	0:00:00
————————————	3000	0	0.65	7	612.25 draws/s	0:00:04	0:00:00
————————————	3000	0	0.64	7	507.92 draws/s	0:00:05	0:00:00
————————————	3000	0	0.58	7	407.92 draws/s	0:00:07	0:00:00

```
Sampling 4 chains for 1_000 tune and 2_000 draw iterations (4_000 + 8_000 draws total) took 22 seconds.
```

FIGURE Py5.2: Console output produced by `PyMC` during MCMC sampling for the model fitted on the Houses dataset. The progress bars display the status of the four Markov chains, while the final line summarizes the total number of tuning iterations, posterior draws, and computation time.

The posterior summary of the sampled parameter values is obtained using the command `az.summary()`, as shown below.

```
az.summary(idata, var_names=["Intercept", "size2", "new", "size2:new", "sigma"],
           hdi_prob=0.95, round_to=3)
```

The resulting summary is presented in Table Py5.1. For each model parameter, the table reports the posterior mean, posterior standard deviation, and the 95% highest density interval (HDI). In addition, several diagnostics (s. Chapter 6 of the book) are provided to assess the quality of the MCMC estimation, including Monte Carlo standard errors (MCSE), effective sample sizes (ESS), and the potential scale reduction factor ($\hat{R}$). Values of $\hat{R}$ close to one, together with large effective sample sizes, indicate satisfactory convergence and efficient posterior sampling.

TABLE Py5.1: Posterior summary obtained from the PyMC model. The table reports posterior means, standard deviations, 95% highest density intervals (HDIs), and MCMC diagnostics.

	mean	sd	hdi_2.5%	hdi_97.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
Intercept	222.028	8.693	203.847	237.803	0.097	0.099	7988.992	5631.183	1.000
size2	0.157	0.015	0.128	0.185	0.000	0.000	7581.611	6051.622	1.000
new	33.292	33.438	-29.441	102.300	0.406	0.343	6788.038	6400.567	1.000
size2:new	0.092	0.033	0.028	0.158	0.000	0.000	5904.516	6000.592	1.001
sigma	79.159	5.828	68.399	90.938	0.063	0.070	8670.480	5913.515	1.001

Py5.2 GLMs for Gamma Distributed Responses in PyMC

As already commented in B.5.1 of the book's Python Appendix, the Gamma model can be fragile in bambi/PyMC due to shape parameter sampling issues.

Next follows the model fit in bambi for the Houses dataset, assuming a gamma distributed response variable:

```
import pandas as pd
import arviz as az
import bambi as bmb
import matplotlib.pyplot as plt
import numpy as np

az.style.use("arviz-darkgrid")
np.random.seed(1234)

houses = pd.read_csv("http://bayes4ds.rwth-aachen.de/data/Houses.dat", delimiter=r"\s+")
houses['size2'] = houses['size'] - houses['size'].mean()  # center size around its mean
houses.head()

priors = {
    "Intercept": bmb.Prior("Normal", mu=5.5, sigma=2),  # prior on log price scale
    "new": bmb.Prior("Normal", mu=0, sigma=10),
    "size2": bmb.Prior("Normal", mu=0, sigma=10),
    "alpha": bmb.Prior("Exponential", lam=0.01)  }

model_gamma = bmb.Model(
    "price ~ size2 + new", data=houses, family="gamma", link="log", categorical="new",
       priors=priors)
fitted_gamma = model_gamma.fit(draws=4000, tune=4000, target_accept=0.95,
      idata_kwargs={"log_likelihood": True})
```

The above model fit did not converge, reporting the following:

```
Sampling 4 chains for 4_000 tune and 4_000 draw iterations (16_000 + 16_000 draws total) took 246 seconds.
There were 8000 divergences after tuning. Increase `target_accept` or reparameterize.
The rhat statistic is larger than 1.01 for some parameters. This indicates problems during sampling. See https://arx
iv.org/abs/1903.08008 for details
The effective sample size per chain is smaller than 100 for some parameters.  A higher number is needed for reliable
rhat and ess computation. See https://arxiv.org/abs/1903.08008 for details
```

The problem occurs due to the Gamma shape parameter α, which controls the dispersion of the response distribution and can be difficult to estimate when the data provide limited

information about the variability. To improve numerical stability and avoid sampling problems, we can either fix the shape parameter to a constant value (e.g., `alpha = 10.0`) or give an informative prior. A constant value for `alpha` is not supported in `bambi` and can fitted in `PyMC`. In `bambi`, we assign a moderately informative prior to this parameter:

$$\alpha \sim \text{Gamma}(10, 1).$$

This prior has expectation $E(\alpha) = 10$ and variance $\text{Var}(\alpha) = 10$. Thus, the prior favors moderate values of the shape parameter while still allowing the data to update the amount of uncertainty about the dispersion. Compared with a very diffuse prior, this regularizing prior reduces the probability of extreme values of α that can lead to an unstable posterior geometry and improve the convergence of the MCMC sampling. The corresponding code follows.

```
import numpy as np
import pandas as pd
import arviz as az
import bambi as bmb
import matplotlib.pyplot as plt

np.random.seed(1234)

# ArviZ plotting style
az.style.use("arviz-darkgrid")

houses = pd.read_csv("http://bayes4ds.rwth-aachen.de/data/Houses.dat",
    delimiter=r"\s+")

houses["size_z"] = (                              # Standardize size
    houses["size"] - houses["size"].mean()) / houses["size"].std()

# Gamma regression model:
priors = { # priors are on the log(price) scale
    "Intercept": bmb.Prior("Normal", mu=5.5, sigma=1),

    # regression coefficients
    "size_z": bmb.Prior("Normal", mu=0, sigma=2),
    "new": bmb.Prior("Normal", mu=0, sigma=2),

    # Gamma shape parameter
    "alpha": bmb.Prior("Gamma", alpha=10, beta=1)}

model_gamma = bmb.Model("price ~ size_z + new + size_z:new",
    data=houses, family="gamma", link="log", categorical="new",
    priors=priors)

print(model_gamma)     # display model structure

# Fit model:
fitted_gamma = model_gamma.fit(draws=4000, tune=4000,
    target_accept=0.95, random_seed=1234,
    idata_kwargs={"log_likelihood": True})

summary = az.summary(fitted_gamma, hdi_prob=0.95, round_to=2)
print(summary)   # posterior summary
```

Recall that in the Gamma regression model with a log link, centering the predictor `size` around its mean is useful for making the intercept more interpretable, because the intercept then corresponds to the expected log-price for a house with average size. However, centering alone does not change the scale or variability of the predictor. If the range of `size` is large,

the regression coefficient may still be associated with a very different scale than the other model parameters, which can lead to slow or unstable MCMC sampling.

To improve numerical stability, we therefore standardized `size` above. The standardized predictor `size_z` has mean zero and standard deviation one. This transformation does not change the information in the data or the fitted values of the model, but it places the regression coefficient on a more convenient scale and facilitates efficient exploration of the posterior distribution by the sampler.

It is worth noting that the convergence problems observed with the original specification were not necessarily caused by the prior on the Gamma shape parameter α. After standardizing the predictor `size`, the model can also be estimated successfully using the more diffuse prior

$$\alpha \sim \text{Exponential}(0.01).$$

This illustrates the important role of predictor scaling in Bayesian computation.

In a Gamma regression with a log link, the expected value of the response is related to the predictors through the linear predictor η_i:

$$\log(\mu_i) = \eta_i = \beta_0 + \beta_1 x_{1i} + \cdots + \beta_p x_{pi},$$

where μ_i denotes the expected value of the response for observation i, and $\beta_0, \ldots, \beta_p$ are the regression coefficients. Equivalently,

$$\mu_i = \exp(\eta_i).$$

Thus, changes in the predictors affect the expected response through an exponential transformation. If a predictor is measured on a large scale, relatively small changes in its coefficient can produce substantial changes in η_i and, consequently, in the expected response μ_i. This can make the posterior distribution more difficult for the MCMC algorithm to explore efficiently and may lead to slow convergence or sampling problems.

Standardization places predictors on a comparable scale by giving them mean zero and standard deviation one. This does not change the information contained in the data or the fitted values of the model, but it leads to regression coefficients of more comparable magnitude and improves the efficiency of the sampling algorithm. Consequently, convergence problems may disappear even without modifying the prior specification for α.

Py5.3 Histograms of the Draws from the Parameters' Posterior Distributions

In Section B.5.5 of the book's Python Appendix, we reproduced in `bambi` the binary logistic regression model presented in Section 5.3.5 for the endometrial cancer risk-factor example. We also provided the commands for generating plots of the posterior distributions of the model parameters and histograms of draws from these posterior distributions, but did not include the resulting plots. These plots are now provided in Figures Py5.3 and Py5.4, respectively.

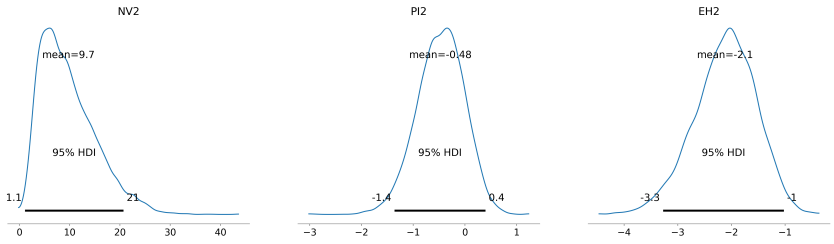

FIGURE Py5.3: Posterior distributions of the parameters of the binary logistic regression model for the endometrial cancer risk-factor example, fitted using `bambi`.

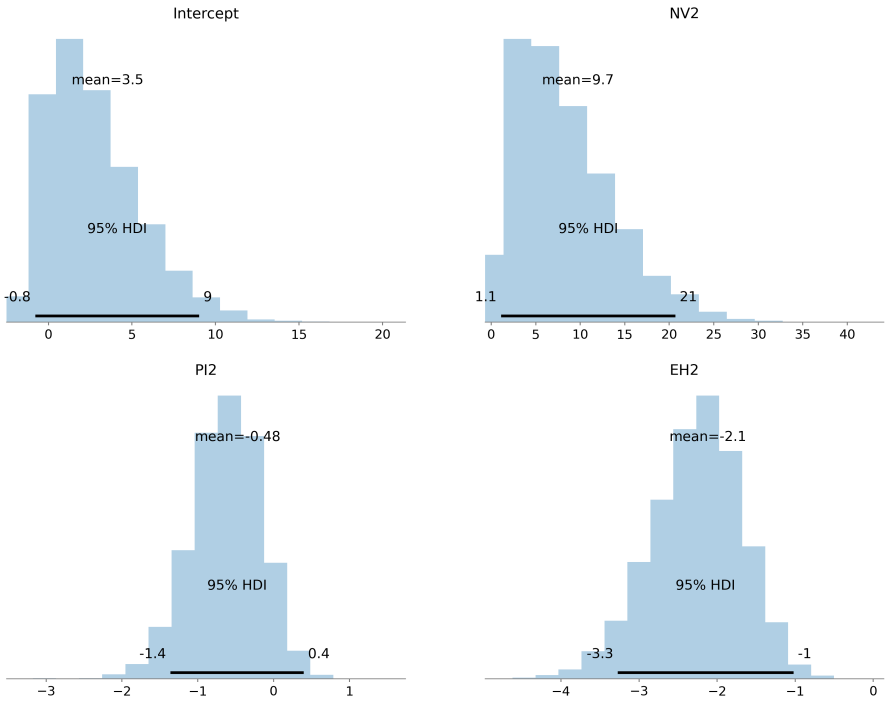

FIGURE Py5.4: Histograms of draws from the posterior distributions of the parameters of the binary logistic regression model for the endometrial cancer risk-factor example, fitted using `bambi`.

Py5.4 Figure 5.5 produced in Python

In Section 5.4.2 of the book we present the Poisson modeling of horseshoe crab satellite counts and in Figure 5.5 we provide plots of weight and color against the number of satellites, plotting the data after jittering their observed values (i.e., adding a small amount of random noise) due to the discreteness of Y (numbero f satelites) and c (color). This figure can be produced in python as shown below.

In Section 5.4.2 of the book, we present a Poisson regression model for the number of satellite crabs observed among horseshoe crabs. Figure 5.5 displays the relationship between weight and color and the number of satellites. Because both the response variable Y (the number of satellites) and the predictor variable color are discrete, the observed values are jittered in the plots by adding a small amount of random noise to improve visualization. The corresponding figure can be reproduced in Python using the code below.

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

np.random.seed(123)

Crabs = pd.read_csv("http://bayes4ds.rwth-aachen.de/data/Crabs.dat",
    sep=r"\s+")

# Convert color category to integers 0-3
color = Crabs["color"].astype(int) - 1

colors = ["cyan", "deepskyblue", "royalblue", "midnightblue"]
labels = ["medium light", "medium", "medium dark", "dark"]

def jitter(x, amount):  # function equivalent to R's jitter()
    return x + np.random.uniform(-amount, amount, size=len(x))

# Plot 1: Color vs number of satellites

plt.figure(figsize=(4, 2.5))

plt.scatter(jitter(Crabs["color"], 0.15),
    jitter(Crabs["sat"], 0.5),
    color=[colors[i] for i in color], s=20)

plt.xticks([1, 2, 3, 4], labels)

plt.xlabel("Color")
plt.ylabel("Number of satellites")

plt.tight_layout()
plt.show()

# Plot 2: Weight vs number of satellites

plt.figure(figsize=(4, 2.5))

plt.scatter(Crabs["weight"],
    jitter(Crabs["sat"], 0.5),
    color=[colors[i] for i in color], s=20)

plt.xlabel("Weight")
plt.ylabel("Number of satellites")
```

```
for c, lab in zip(colors, labels):  # legend
    plt.scatter([], [], color=c, s=20, label=lab)

plt.legend(loc="upper right", frameon=True,
    fontsize=8, markerscale=0.7)
plt.show()
```

Py5.5 Separation Plots for Logistic Regression

In Section 5.3.4, we discussed complete and quasi-complete separation in logistic regression and their consequences for parameter estimation. A *separation plot* provides a graphical tool for assessing how well a logistic regression model separates the outcome categories.

The observations are first ordered according to their fitted probabilities, from the smallest to the largest. The colored areas in the plot represent the observed outcome categories along this ordering. When the model separates the outcome categories well, the plot shows one color dominating the lower range of predicted probabilities and the other color dominating the higher range. Substantial overlap between the two colored areas indicates weaker discrimination between the outcome categories. Conversely, almost complete separation of the two areas may indicate complete or quasi-complete separation.

To verify the extent of separability in the estimated results for the endometrial cancer example, fitted in `bambi` in Section B.5.5 of the book's Python Appendix with the model fit saved under `fit_bayes`, we consider the model's predictions and construct the separation plot, offered in `bambi`, showing it in Figure Py5.5:

```
model.predict(fit_bayes, kind="mean")
ax = az.plot_separation(fit_bayes, y="HG", figsize=(9,0.5))
plt.savefig("fig5_separ_py.png", dpi=300)
```

FIGURE Py5.5: Separation plot for the response variable HG of the endometrial cancer data. Dark blue corresponds to HG=1 and light blue to HG=0.

Figure Py5.5 suggests that the model has good discriminatory ability without showing signs of complete or quasi-complete separation that could lead to estimation problems. The absence of a clear separation between the two colored areas indicates that, although the model can distinguish reasonably well between the outcome categories, the predictors do not perfectly separate the two classes.

Py6

Diagnostics in `Python`

Py6.1 Diagnostics for the Gamma Regression Model on the Houses Example

In the previous chapter, we fitted a gamma regression model in Python to house selling prices in Florida. Here, we provide the commands for obtaining the marginal posterior distributions and trace plots of the model parameters, as well as the corresponding rank plots. The produced plots are given in Figures Py6.1 and Py6.2, respectively.

```
az.plot_trace(fitted_gamma)
plt.show()

az.plot_rank(fitted_gamma)
plt.show()
```

The rank plot is another MCMC diagnostic, designed particularly to assess whether the chains are sampling from the same stationary distribution.

For each parameter, all draws from all chains are pooled and assigned ranks. The rank plot then shows the distribution of these ranks separately for each chain. If the chains have converged and are sampling from the same posterior distribution, the ranks from each chain should be approximately uniformly distributed and the distributions for the different chains should be similar. Systematic differences between chains indicate that the chains may be sampling from different distributions, suggesting non-convergence or poor mixing.

The rank plot provides a different perspective on the MCMC chains than the trace plot. In particular, whereas the trace plot assesses whether the chains appear stationary and well mixed throughout the sampling process, the rank plot assesses whether the chains appear to sample from the same posterior distribution. For a well-converged MCMC analysis, one would generally expect stable, well-mixed traces and approximately uniform, similar rank distributions across chains.

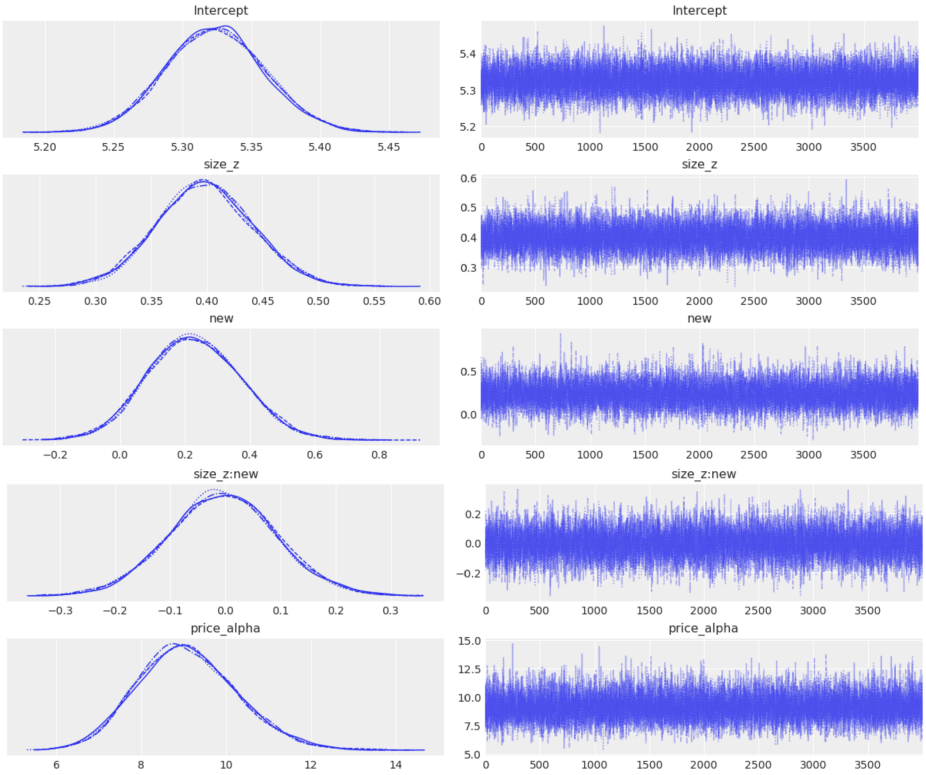

FIGURE Py6.1: Trace plots and marginal posterior distributions for the parameters of the fitted model for house ceiling prices in Florida.

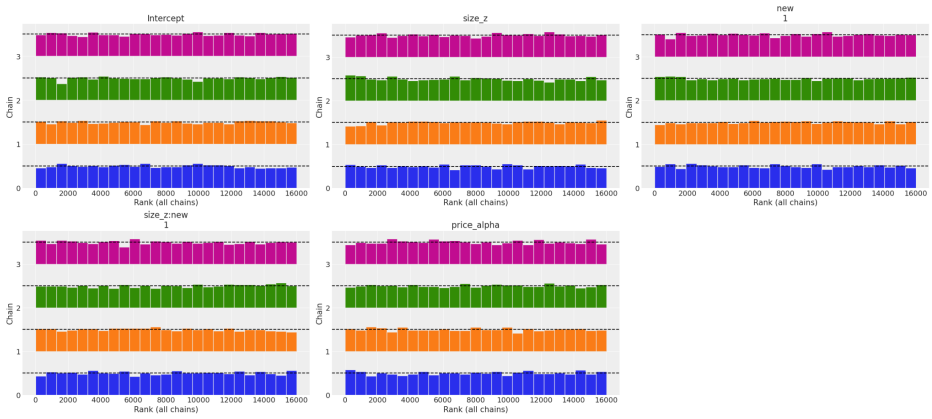

FIGURE Py6.2: Rank plots for the parameters of the fitted model for house ceiling prices in Florida.

Py7

Extending Bayesian Models with `Python`

Py7.1 Posterior Predictive Distribution and Its Use for Model Checking

In B.7.6 of the book's Appendix, we pictured in Figure B.7.5 the *Bayesian stochastic block model* (SBM) fitted to Zachary's karate club network with three clusters. The code for constructing it is provided next, using `networkx`.

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt

# Zachary's karate club network
G = nx.karate_club_graph()
# Binary adjacency matrix
A = nx.to_numpy_array(G, dtype=int)
A = (A > 0).astype(int)
# Remove self-connections
np.fill_diagonal(A, 0)
n = A.shape[0]
# Number of latent groups
K = 3
print("Number of nodes:", n)
print("Number of edges:", A.sum() // 2)
print("Unique values in A:", np.unique(A))

def bayesian_sbm(A, K, n_iter=10000, burn=3000, thin=10,
    alpha=1.0, a=1.0, b=1.0, seed=1234):
    rng = np.random.default_rng(seed)
    n = A.shape[0]
    # Make sure that A is binary
    A = (A > 0).astype(int)
    z = rng.integers(    # initial group assignments
        0, K, size=n)
    pi = rng.dirichlet(  # initial group proportions
        np.repeat(alpha, K) )
    P = rng.beta(        # initial block probabilities
        a, b, size=(K, K) )
    P = (P + P.T) / 2   # undirected network
    # Storage for posterior draws
    z_samples = []
    P_samples = []
    pi_samples = []
    # ========================================================
    # Gibbs sampler
    # ========================================================
    for iteration in range(n_iter):
        # ----------------------------------------------------
        # Step 1: update block probabilities
        # ----------------------------------------------------
        for k in range(K):
```

```
            for l in range(k, K):
                if k == l:
                    # Nodes currently assigned to group k
                    nodes = np.where(z == k)[0]
                    if len(nodes) >= 2:
                        ii, jj = np.triu_indices(
                            len(nodes),
                            k=1)
                        subA = A[np.ix_(nodes, nodes)]
                        y = subA[ii, jj]
                        n_edges = int(y.sum())
                        n_possible = len(y)
                    else:
                        n_edges = 0
                        n_possible = 0
                else:
                    # Nodes in groups k and l
                    nodes_k = np.where(z == k)[0]
                    nodes_l = np.where(z == l)[0]
                    if (len(nodes_k) > 0
                        and len(nodes_l) > 0 ):
                        y = A[np.ix_(nodes_k, nodes_l)]
                        n_edges = int(y.sum())
                        n_possible = y.size
                    else:
                        n_edges = 0
                        n_possible = 0
                # Beta posterior
                P[k, l] = rng.beta(
                    a + n_edges,
                    b + n_possible - n_edges)
                # Symmetry
                P[l, k] = P[k, l]
        # ------------------------------------------------
        # Step 2: update group proportions
        # ------------------------------------------------
        counts = np.bincount(z, minlength=K)
        pi = rng.dirichlet(alpha + counts)
        # ------------------------------------------------
        # Step 3: update group membership of each node
        # ------------------------------------------------
        for i in rng.permutation(n):
            log_prob = np.empty(K)
            # All nodes except i
            mask = np.ones(n, dtype=bool)
            mask[i] = False
            A_i = A[i, mask]
            z_other = z[mask]
            for k in range(K):
                p = np.clip(P[k, z_other],
                    1e-10,
                    1 - 1e-10)
                log_prob[k] = (np.log(pi[k])
                    +np.sum(A_i * np.log(p)
                        +(1 - A_i) * np.log(1 - p)))
            # Numerical stabilization
            log_prob -= np.max(log_prob)
            prob = np.exp(log_prob)
            prob /= prob.sum()
            # Draw new group membership
            z[i] = rng.choice(K,  p=prob)
        # ------------------------------------------------
        # Store posterior draws
        # ------------------------------------------------
```

```
            if (iteration >= burn
                and
                (iteration - burn) % thin == 0):
                z_samples.append(z.copy())
                P_samples.append(P.copy())
                pi_samples.append(pi.copy())
        return (np.asarray(z_samples),
            np.asarray(P_samples),
            np.asarray(pi_samples))

z_samples, P_samples, pi_samples = bayesian_sbm(A, K=3,
    n_iter=10000, burn=3000, thin=10,
    alpha=5.0,     # 1.0
    a=1.0, b=1.0,
    seed=1234)
print("Number of posterior draws:", len(z_samples))
print("Shape of z_samples:", z_samples.shape)
print("Shape of P_samples:", P_samples.shape)

# Posterior probability that two nodes belong to the same latent group
C = np.mean(z_samples[:, :, None] == z_samples[:, None, :], axis=0)
print("Shape of co-clustering matrix:", C.shape)

# Construct a co-clustering matrix for each MCMC draw
C_draws = np.empty((len(z_samples), n, n))

for s, z in enumerate(z_samples):
    C_draws[s] = (z[:, None] == z[None, :])

# Find the draw closest to the posterior mean co-clustering matrix
distances = np.mean((C_draws - C) ** 2, axis=(1, 2))
representative = np.argmin(distances)
z_rep = z_samples[representative]
print("Representative group sizes:")
for k in range(K):
    print("Group", k + 1, ":", np.sum(z_rep == k), "nodes")

P_mean = P_samples.mean(axis=0)
print("Posterior mean block-probability matrix:")
print(np.round(P_mean, 3))

# Fixed layout for reproducibility
pos = nx.spring_layout(G, seed=1234)
# Three colors
colors = ["#E41A1C",   # red
          "#4DAF4A",   # green
          "#377EB8"]    # blue
fig, ax = plt.subplots(figsize=(10, 7))
# Edges
nx.draw_networkx_edges(G, pos, ax=ax, edge_color="gray",
    alpha=0.70, width=1.0)
# Nodes
nx.draw_networkx_nodes(G, pos, ax=ax,
    node_color=[colors[z_rep[i]]
        for i in range(n)],
    node_size=500, edgecolors="black", linewidths=0.0)
# Node labels
nx.draw_networkx_labels(G, pos, ax=ax, font_size=9)

# ax.set_title(
#    "Bayesian stochastic block model\n"
#    "Zachary's karate club network",
#   fontsize=14)
```

```
ax.axis("off")
plt.tight_layout()
plt.savefig("karate_bayesian_sbm_K3.pdf",  # save in pdf file
    bbox_inches="tight")
plt.savefig("karate_bayesian_sbm_K3.png",  # save in png file
    dpi=300,bbox_inches="tight")
plt.show()          # to show on the screen
```

The network fitted here is pictured in Figure Py7.1 and is the same as the one presented in Figure 7.9 (right) of the book, fitted using `R`. The only difference is the node labeling: `R` labels the nodes from 1 to 34, whereas `networkX` labels them from 0 to 33 by default. The node labels can be recoded in Python from $0, \ldots, 33$ to $1, \ldots, 34$ as follows:

```
G = nx.relabel_nodes(G, {i: i + 1 for i in G.nodes()})
```

This recoding changes only the node labels and does not alter the structure of the network.

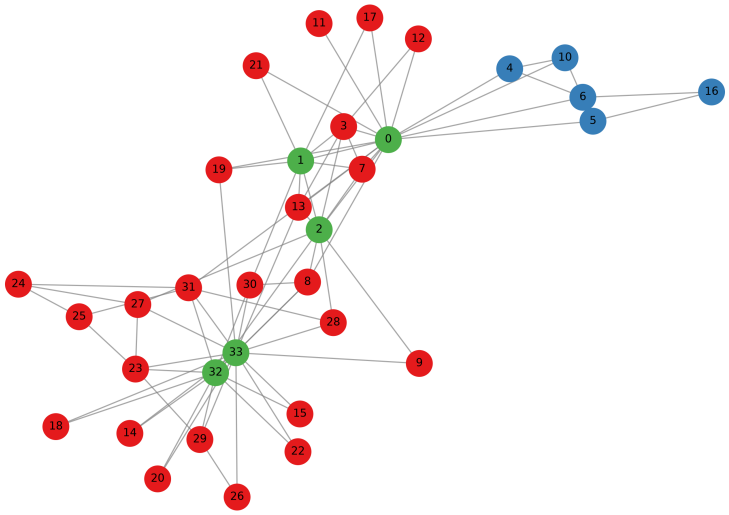

FIGURE Py7.1: Bayesian SBM fitted to Zachary's karate club network with $K = 3$ clusters.