

BAYESIAN MODELS for Astrophysical Data

Using R, JAGS, Python, and Stan

Joseph M. Hilbe, Rafael S. de Souza,
and Emille E. O. Ishida

Bayesian Models for Astrophysical Data

Using R, JAGS, Python, and Stan

This comprehensive guide to Bayesian methods in astronomy enables hands-on work by supplying complete R, JAGS, Python, and Stan code, to use directly or to adapt. It begins by examining the normal model from both frequentist and Bayesian perspectives and then progresses to a full range of Bayesian generalized linear and mixed or hierarchical models, as well as additional types of models such as ABC and INLA. The book provides code that is largely unavailable elsewhere and includes details on interpreting and evaluating Bayesian models. Initial discussions offer models in synthetic form so that readers can easily adapt them to their own data; later the models are applied to real astronomical data. The consistent focus is on hands-on modeling, analysis of data, and interpretations that address scientific questions. A must-have for astronomers, its concrete approach will also be attractive to researchers in the sciences more generally.

Joseph M. Hilbe is Solar System Ambassador with NASA's Jet Propulsion Laboratory, Adjunct Professor of Statistics at Arizona State University, and Professor Emeritus at the University of Hawaii. He is currently President of the International Astrostatistics Association (IAA) and was awarded the IAA's 2016 Outstanding Contributions to Astrostatistics medal, the association's top award. Hilbe is an elected Fellow of both the American Statistical Association and the IAA and is a full member of the American Astronomical Society. He has authored 19 books on statistical modeling, including leading texts on modeling count and binomial data. His book, *Modeling Count Data* (2014, Cambridge) received the 2015 PROSE honorable mention for books in mathematics.

Rafael S. de Souza is a researcher at Eötvös Loránd University. He is currently Vice-President for development of the International Astrostatistics Association and was awarded the IAA's 2016 Outstanding Publication in Astrostatistics award. He has authored dozens of scientific papers, serving as the leading author for over 20.

Emille E. O. Ishida is a researcher at the Université Clermont-Auvergne (Université Blaise Pascal). She is cochair of the Cosmostatistics Initiative and coordinator of its Python-related projects. She is a specialist in machine learning applications to astronomy with special interests in type Ia supernovae spectral characterization, classification, and cosmology. She has been the lead author of numerous articles in prominent astrophysics journals and currently serves as chair of the IAA public relations committee.

Bayesian Models for Astrophysical Data

Using R, JAGS, Python, and Stan

JOSEPH M. HILBE

Arizona State University and Jet Propulsion Laboratory, California Institute of Technology

RAFAEL S. DE SOUZA

Eötvös Loránd University, Budapest

EMILLE E. O. ISHIDA

Université Clermont-Auvergne (Université Blaise Pascal), France



CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

4843/24, 2nd Floor, Ansari Road, Daryaganj, Delhi – 110002, India

79 Anson Road, #06–04/06, Singapore 079906

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning, and research at the highest international levels of excellence.

www.cambridge.org

Information on this title: www.cambridge.org/9781107133082

© Joseph M. Hilbe, Rafael S. de Souza, and Emille E. O. Ishida 2017

This publication is in copyright. Subject to statutory exception and to the provisions of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

First published 2017

Printed in the United Kingdom by TJ International Ltd, Padstow, Cornwall

A catalog record for this publication is available from the British Library.

Library of Congress Cataloging-in-Publication Data

Names: Hilbe, Joseph M., 1944- | De Souza, Rafael S. | Ishida, Emille E. O.

Title: Bayesian models for astrophysical data using R, JAGS, Python, and Stan / Joseph M. Hilbe, Jet Propulsion Laboratory and Arizona State University, Rafael S. de Souza, MTA Eötvös Loránd University, Emille E.O. Ishida, Université de Clermont-Ferrand II (Université Blaise Pascal), France.

Description: Cambridge : Cambridge University Press, 2017. | Includes bibliographical references and index.

Identifiers: LCCN 2017009823 | ISBN 9781107133082 (alk. paper)

Subjects: LCSH: Statistical astronomy. | Statistical astronomy—Data processing. | Astronomy—Data processing.

Classification: LCC QB149 .H55 2017 | DDC 520.1/519542–dc23

LC record available at <https://lccn.loc.gov/2017009823>

ISBN 978-1-107-13308-2 Hardback

Additional resources for this publication at www.cambridge.org/bayesianmodels

Cambridge University Press has no responsibility for the persistence or accuracy of URLs for external or third-party internet web sites referred to in this publication and does not guarantee that any content on such web sites is, or will remain, accurate or appropriate.

In Memoriam

Joseph M. Hilbe
(30th December, 1944 – 12th March, 2017)

Our mentor, colleague, and friend.
May we all live up to your legacy.
Farewell Joe. It was an honour to meet you.

Contents

Preface

1 Astrostatistics

- 1.1 The Nature and Scope of Astrostatistics
- 1.2 The Recent Development of Astrostatistics
- 1.3 What is a Statistical Model?
- 1.4 Classification of Statistical Models

2 Prerequisites

- 2.1 Software
- 2.2 R
- 2.3 JAGS
- 2.4 Python
- 2.5 Stan

3 Frequentist vs. Bayesian Methods

- 3.1 Frequentist Statistics
 - 3.1.1 Fitting a Linear Regression in R
 - 3.1.2 Fitting a Linear Regression in Python
- 3.2 Basic Theory of Bayesian Modeling
 - 3.2.1 Example: Calculating a Beta Prior and Posterior Analytically
 - 3.2.2 Fitting a Simple Bayesian Normal Model using R
 - 3.2.3 Fitting a Simple Bayesian Normal Model using Python
- 3.3 Selecting Between Frequentist and Bayesian Modeling

4 Normal Linear Models

- 4.1 The Gaussian or Normal Model
 - 4.1.1 Bayesian Synthetic Normal Model in R using JAGS
 - 4.1.2 Bayesian Synthetic Normal Model in R using JAGS and the Zero Trick
 - 4.1.3 Bayesian Synthetic Normal Model in Python using Stan
 - 4.1.4 Bayesian Synthetic Normal Model using Stan with a Customized Likelihood
- 4.2 Multivariate Normal Model
 - 4.2.1 Multivariate Linear Regression in R using JAGS
 - 4.2.2 Multivariate Linear Regression in Python using Stan
- 4.3 Bayesian Errors-in-Measurements Modeling
 - 4.3.1 Generating Data with Errors using R
 - 4.3.2 Build Model ignoring Errors in R using JAGS

- 4.3.3 Build Model including Errors in R using JAGS
- 4.3.4 Bayesian Errors-in-Measurements Modeling in Python using Stan

5 GLMs Part I – Continuous and Binomial Models

- 5.1 Brief Overview of Generalized Linear Models
- 5.2 Bayesian Continuous Response Models
 - 5.2.1 Bayesian Lognormal Model
 - 5.2.2 Bayesian Gamma Models
 - 5.2.3 Bayesian Inverse Gaussian Models
 - 5.2.4 Bayesian Beta Model
- 5.3 Bayesian Binomial Models
 - 5.3.1 Bayesian Bernoulli Logit Models
 - 5.3.2 Bayesian Bernoulli Probit Models
 - 5.3.3 Bayesian Grouped Logit or Binomial Model
 - 5.3.4 Bayesian Grouped Probit Model
 - 5.3.5 Bayesian Beta–Binomial Models

6 GLMs Part II – Count Models

- 6.1 Bayesian Poisson Models
 - 6.1.1 Poisson Models with R
 - 6.1.2 Poisson Models with JAGS
 - 6.1.3 Poisson Models in Python
 - 6.1.4 Poisson Models in Python using Stan
- 6.2 Bayesian Negative Binomial Models
 - 6.2.1 Modeling the Negative Binomial using JAGS
 - 6.2.2 Negative Binomial Models in Python using pymc3
 - 6.2.3 Modeling the Negative Binomial in Python using Stan
- 6.3 Bayesian Generalized Poisson Model
 - 6.3.1 Generalized Poisson Model using JAGS
 - 6.3.2 Generalized Poisson Model using Stan
- 6.4 Bayesian Zero-Truncated Models
 - 6.4.1 Bayesian Zero-Truncated Poisson Model
 - 6.4.2 Zero-Truncated Poisson in Python using Stan
 - 6.4.3 Bayesian Zero-Truncated Negative Binomial Model
- 6.5 Bayesian Three-Parameter NB Model (NB-P)
 - 6.5.1 Three-Parameter NB-P Model using JAGS
 - 6.5.2 Three-Parameter NB-P Models in Python using Stan

7 GLMs Part III – Zero-Inflated and Hurdle Models

- 7.1 Bayesian Zero-Inflated Models
 - 7.1.1 Bayesian Zero-Inflated Poisson Model
 - 7.1.2 Bayesian Zero-Inflated Negative Binomial Model
- 7.2 Bayesian Hurdle Models
 - 7.2.1 Bayesian Poisson–Logit Hurdle Model
 - 7.2.2 Bayesian Negative Binomial–Logit Hurdle Model
 - 7.2.3 Bayesian Gamma–Logit Hurdle Model
 - 7.2.4 Bayesian Lognormal–Logit Hurdle Model

8 Hierarchical GLMMs

- 8.1 Overview of Bayesian Hierarchical Models/GLMMs
- 8.2 Bayesian Gaussian or Normal GLMMs
 - 8.2.1 Random Intercept Gaussian Data
 - 8.2.2 Bayesian Random Intercept Gaussian Model in R using JAGS
 - 8.2.3 Bayesian Random Intercept Normal Model in R using JAGS
 - 8.2.4 Bayesian Random Intercept Normal Model in Python using Stan
- 8.3 Bayesian Binary Logistic GLMMs
 - 8.3.1 Random Intercept Binary Logistic Data
 - 8.3.2 Bayesian Random Intercept Binary Logistic Model with R
 - 8.3.3 Bayesian Random Intercept Binary Logistic Model with Python
 - 8.3.4 Bayesian Random Intercept Binary Logistic Model in R using JAGS
 - 8.3.5 Bayesian Random Intercept Binary Logistic Model in Python using Stan
- 8.4 Bayesian Binomial Logistic GLMMs
 - 8.4.1 Random Intercept Binomial Logistic Data
 - 8.4.2 Bayesian Random Intercept Binomial Logistic Model in R using JAGS
 - 8.4.3 Bayesian Random Intercept Binomial Logistic Model in Python using Stan
- 8.5 Bayesian Poisson GLMMs
 - 8.5.1 Random Intercept Poisson Data
 - 8.5.2 Bayesian Random Intercept Poisson Model with R
 - 8.5.3 Bayesian Random Intercept Poisson Model in Python
 - 8.5.4 Bayesian Random Intercept Poisson Model in R using JAGS
 - 8.5.5 Bayesian Random Intercept Poisson Model in Python using Stan
 - 8.5.6 Bayesian Random-Intercept–Random-Slopes Poisson Model
- 8.6 Bayesian Negative Binomial GLMMs
 - 8.6.1 Random Intercept Negative Binomial Data
 - 8.6.2 Random Intercept Negative Binomial MLE Model using R
 - 8.6.3 Bayesian Random Intercept Negative Binomial Model using Python
 - 8.6.4 Bayesian Random Intercept Negative Binomial Model in R using JAGS
 - 8.6.5 Bayesian Random Intercept Negative Binomial Model in Python using Stan

9 Model Selection

- 9.1 Information Criteria Tests for Model Selection
 - 9.1.1 Frequentist and Bayesian Information Criteria
 - 9.1.2 Bayesian Deviance Statistic
 - 9.1.3 pD and Deviance Information Criteria (DIC)
- 9.2 Model Selection with Indicator Functions
- 9.3 Bayesian LASSO

10 Astronomical Applications

- 10.1 Normal Model, Black Hole Mass, and Bulge Velocity Dispersion
 - 10.1.1 Data
 - 10.1.2 The Statistical Model Formulation
 - 10.1.3 Running the Model in R using JAGS
 - 10.1.4 Running the Model in Python using Stan
- 10.2 Gaussian Mixed Models, Type Ia Supernovae, and Hubble Residuals
 - 10.2.1 Data
 - 10.2.2 Statistical Model Formulation
 - 10.2.3 Running the Model in R using JAGS
 - 10.2.4 Running the Model in Python using Stan

- 10.3 Multivariate Normal Mixed Model and Early-Type Contact Binaries
 - 10.3.1 Data
 - 10.3.2 The Statistical Model Formulation
 - 10.3.3 Running the Model in R using JAGS
 - 10.3.4 Running the Model in Python using Stan
- 10.4 Lognormal Distribution and the Initial Mass Function
 - 10.4.1 Data
 - 10.4.2 Statistical Model Formulation
 - 10.4.3 Running the Model in R using JAGS
 - 10.4.4 Running the Model in Python using Stan
- 10.5 Beta Model and the Baryon Content of Low Mass Galaxies
 - 10.5.1 Data
 - 10.5.2 The Statistical Model Formulation
 - 10.5.3 Running the Model in R using JAGS
 - 10.5.4 Running the Model in Python using Stan
- 10.6 Bernoulli Model and the Fraction of Red Spirals
 - 10.6.1 Data
 - 10.6.2 The Statistical Model Formulation
 - 10.6.3 Running the Model in R using JAGS
 - 10.6.4 Running the Model in Python using Stan
- 10.7 Count Models, Globular Cluster Population, and Host Galaxy Brightness
 - 10.7.1 Data
 - 10.7.2 The Statistical Poisson Model Formulation
 - 10.7.3 Running the Poisson Model in R using JAGS
 - 10.7.4 The Statistical Negative Binomial Model Formulation
 - 10.7.5 Running the Negative Binomial Model in R using JAGS
 - 10.7.6 The Statistical NB-P Model Formulation
 - 10.7.7 Running the NB-P Model in R using JAGS
 - 10.7.8 Running the NB-P Model in Python using Stan
- 10.8 Bernoulli Mixed Model, AGNs, and Cluster Environment
 - 10.8.1 Data
 - 10.8.2 Statistical Model Formulation
 - 10.8.3 Running the Model in R using JAGS
 - 10.8.4 Running the Model in Python using Stan
- 10.9 Lognormal–Logit Hurdle Model and the Halo–Stellar-Mass Relation
 - 10.9.1 Data
 - 10.9.2 The Statistical Model Formulation
 - 10.9.3 Running the Model in R using JAGS
 - 10.9.4 Running the Model in Python using Stan
- 10.10 Count Time Series and Sunspot Data
 - 10.10.1 Data
 - 10.10.2 Running the Normal AR(1) Model in R using JAGS
 - 10.10.3 Running the Negative Binomial AR Model in R using JAGS
 - 10.10.4 Running the Negative Binomial AR Model in Python using Stan
- 10.11 Gaussian Model, ODEs, and Type Ia Supernova Cosmology
 - 10.11.1 Data
 - 10.11.2 The Statistical Model Formulation
 - 10.11.3 Running the Model in R using Stan

- 10.11.4 Errors in Measurements
- 10.12 Approximate Bayesian Computation
 - 10.12.1 Distance
 - 10.12.2 Population Monte Carlo ABC
 - 10.12.3 Toy Model
 - 10.12.4 CosmoABC
- 10.13 Remarks on Applications

11 The Future of Astrostatistics

Appendix A Bayesian Modeling using INLA

Appendix B Count Models with Offsets

Appendix C Predicted Values, Residuals, and Diagnostics

References

Index

Preface

Bayesian Models for Astrophysical Data provides those who are engaged in the Bayesian modeling of astronomical data with guidelines on how to develop code for modeling such data, as well as on how to evaluate a model as to its fit. One focus in this volume is on developing statistical models of astronomical phenomena from a Bayesian perspective. A second focus of this work is to provide the reader with statistical code that can be used for a variety of Bayesian models.

We provide fully working code, not simply code snippets, in R, JAGS, Python, and Stan for a wide range of Bayesian statistical models. We also employ several of these models in real astrophysical data situations, walking through the analysis and model evaluation. This volume should foremost be thought of as a guidebook for astronomers who wish to understand how to select the model for their data, how to code it, and finally how best to evaluate and interpret it. The codes shown in this volume are freely available online at www.cambridge.org/bayesianmodels. We intend to keep it continuously updated and report any eventual bug fixes and improvements required by the community. We advise the reader to check the online material for practical coding exercises.

This is a volume devoted to applying Bayesian modeling techniques to astrophysical data. Why Bayesian modeling? First, science appears to work in accordance with Bayesian principles. At each stage in the development of a scientific study new information is used to adjust old information. As will be observed when reviewing the examples later in this volume, this is how Bayesian modeling works. A posterior distribution created from the mixing of the model likelihood (derived from the model data) and a prior distribution (outside information we use to adjust the observed data) may itself be used as a prior for yet another enhanced model. New information is continually being used in models over time to advance yet newer models. This is the nature of scientific discovery. Yet, even if we think of a model in isolation from later models, scientists always bring their own perspectives into the creation of a model on the basis of previous studies or from their own experience in dealing with the study data. Models are not built independently of the context, so bringing in outside prior information to the study data is not unusual or overly subjective. Frequentist statisticians choose the data and predictors used to study some variable – most of the time based on their own backgrounds and external studies. Bayesians just make the process more explicit.

A second reason for focusing on Bayesian models is that recently there has been a rapid move by astronomers to Bayesian methodology when analyzing their data. Researchers in many other disciplines are doing the same, e.g., in ecology, environmental science, health outcomes analysis, communications, and so forth. As we discuss later, this is largely the case because computers are

now finally able to engage with complex MCMC-based algorithms, which entail thousands of sampling iterations and many millions of calculations in arriving at a single posterior distribution. Moreover, aside from faster computers with much greater memory, statisticians and information scientists have been developing ever more efficient estimation algorithms, which can now be found in many commercial statistical software packages as well as in specially developed Bayesian packages, e.g., JAGS, Stan, OpenBUGS, WinBUGS, and MLwiN. Our foremost use in the book is of JAGS and Stan. The initial release of JAGS by Martyn Plummer was in December 2007. Stan, named after Stanislaw Ulam, co-developer of the original MCMC algorithm in 1949 with Nicholas Metropolis ([Metropolis and Ulam, 1949](#)), was first released in late August 2012. However, the first stable release of Stan was not until early December 2015, shortly before we began writing this volume. In fact, Stan was written to overcome certain problems with the convergence of multilevel models experienced with BUGS and JAGS. It is clear that this book could not have been written a decade ago, or even five years ago, as of this writing. The technology of Bayesian modeling is rapidly advancing, indicating that astrostatistics will be advancing with it as well. This book was inspired by the new modeling capabilities being jointly provided by the computer industry and by statisticians, who are developing better methods of analyzing data.

Bayesian Models for Astrophysical Data differs from other books on astrostatistics. The book is foremost aimed to provide the reader with an understanding of the statistical modeling process, and it displays the complete JAGS and, in most cases, Stan code for a wide range of models. Each model is discussed, with advice on when to use it and how best to evaluate it with reference to other models. Following an overview of the meaning and scope of statistical modeling, and of how frequentist and Bayesian models differ, we examine the basic Gaussian or normal model. This sets the stage for us to then present complete modeling code based on synthetic data for what may be termed Bayesian generalized linear models. We then extend these models, discussing two-part models, mixed models, three-parameter models, and hierarchical models. For each model we show the reader how to create synthetic data based on the distributional assumptions of the model being evaluated. The model code is based on the synthetic data but because of that it is generic and can easily be adapted to alternative synthetic data or to real data. We provide full JAGS and Stan code for each model. In the majority of the examples in this volume, JAGS code is run from the R environment and Stan from within the Python environment. In many cases we also display the code for, and run, stand-alone R and Python models.

Following our examination of models, including continuous, binary, proportion, grouped, and count response models we address model diagnostics. Specifically, we discuss information criteria including the Bayesian deviance, the deviance information criterion (DIC), and the pD and model predictor selection methods, e.g., the Kuo and Mallick test and Bayesian LASSO techniques. In Chapter 10 on applications we bring in real astronomical data from previously published studies and analyze them using the models discussed earlier in the book. Examples are the use of time series for sunspot events, lognormal models for the stellar initial mass function, and errors in variables for the analysis of supernova properties. Other models are discussed as well, with the likelihood-free approximate Bayesian computation (ABC) presented alongside a pure Python computation as an alternative for analyzing Sunyaev–Zeldovich surveys.

This book should be thought of as a guidebook to help researchers develop more accurate and useful models for their research studies. In this regard we do not go into deep details of an underlying astrophysical model but use it as a template, so the researcher can understand how to

connect a given statistical model with the data at hand and, therefore, how to apply to it to his or her own problem. It is a new approach to learning how to apply statistics to astronomical research, but we believe that the pedagogy is sound.

It is important for astrostatisticians and astroinformaticists to read a variety of books and articles related to the statistical analysis of astronomical data. Doing so enhances the analyst's research acumen. Below we briefly point out the themes or approaches given in three other books that have recently been published on astrostatistics. We do not regard these books as competitors. Rather, they should be thought of as complements to this book.

In our view, [Feigelson and Babu \(2012a\)](#) is generally regarded by astronomical and general astrostatistical communities as the standard text on astrostatistics from the frequentist perspective. The text provides readers with a wide range of both parametric and non-parametric methods for evaluating astronomical data; R is used throughout for examples. We recommend this text to astronomers who wish to have an up-to date volume on astrostatistics from the frequentist tradition. The authors are planning a second edition, which will likely be out in 2018.

The other two books we recommend to accompany the present text are volumes in the *Springer Series on Astrostatistics*, which is co-sponsored by the International Astrostatistics Association (IAA). [Andreon and Weaver \(2015\)](#) provides a Bayesian approach to the physical sciences, with an emphasis on astronomy. It is closest to this volume in approach. JAGS code is displayed in the book for many examples. The emphasis is on Gaussian-based models, although examples using Bayesian Poisson and logistic models are examined. The book provides a number of nicely presented guidelines and theory. Our third recommendation is the general purpose astrostatistics text by [Chattpadhyay and Chattpadhyay \(2014\)](#), which is frequentist oriented but has a very nice component on Monte Carlo simulation. The R code is used for examples. All three books are excellent resources, but they substantially differ from this volume.

The aim of our book is to provide astrostatisticians with the statistical code and information needed to create insightful and useful models of their data. Much of the code presented here cannot be found elsewhere, but it can be used to develop important models of astronomical data. It is central to us to provide readers with better statistical tools and code for advancing our understanding of the Universe. If this book has achieved this goal to any degree, it has been worth the effort.

We assume that readers have a basic background in frequency-based statistical modeling, and maximum likelihood estimation in particular. We do not expect that readers have gone beyond normal or basic linear regression, but having at least some background in logistic regression or Poisson regression will be helpful. It is not necessary, though. It will also be helpful if the reader knows the basics of R and Python programming. We use R as the base language for examples but make every attempt to explain the code as we go along. For most examples complete commented Python scripts are provided, allowing the reader to benefit also from a direct comparison between these programming languages. In addition, we do not expect that readers have a background in Bayesian modeling – the subject of the text – but the more you know already, the better.

Owing to these assumptions we cover frequency-based modeling concepts rather quickly,

touching on only major points to be remembered when contrasting frequency-based and Bayesian methodologies. We provide an overview of Bayesian modeling and of how it differs from frequentist-based modeling. However, we do not focus on theory. There are a plethora of books and other publications on this topic. We do attempt, however, to provide sufficient background on Bayesian methodology to allow the reader to understand the logic and purpose of the Bayesian code discussed in the text. Our main emphasis is to provide astrostatisticians with the code and the understanding to employ models on astrophysical data that have previously not been used, or have only seldom been used – but which perhaps should be used more frequently. We provide modeling code and diagnostics using synthetic data as well as using real data from the literature.

We should mention that researchers in disciplines other than astrophysics may also find the book useful. The code and discussion using synthetic data are applicable to nearly all disciplines.

We are grateful to a number of our colleagues for their influence on this work. Foremost we wish to acknowledge Alain F. Zuur of Highlands Statistics in Newburgh, Scotland for his contributions to several of the JAGS models used in the book. The codes for various models are adaptations of code from [Zuur, Hilbe, and Ieno \(2013\)](#), a book on both the frequentist and the Bayesian approaches to generalized linear models and generalized linear mixed models for ecologists. Moreover, we have adopted a fairly uniform style or format for constructing JAGS models on this basis of the work of Dr. Zuur and the first author of this volume, which is reflected in [Zuur, Hilbe, and Ieno \(2013\)](#).

We would like to express our appreciation to Diana Gillooly, statistics editor at Cambridge University Press, for accepting our proposal to write this book. She has been supportive throughout the life of the book's preparation. We thank Esther Miguéliz, our Content Manager, and Susan Parkinson, our freelance copy-editor, for their dedicated work in improving this book in a number of ways. Their professionalism and assistance is greatly appreciated. We also wish to express our gratitude to John Hammersley, CEO of Overleaf (WriteLatex Ld), who provided us with Overleaf Pro so that we could work simultaneously on the manuscript. This new technology makes collaborative authorship endeavors much easier than in the past. Since we live and work in Arizona, Hungary, and France respectively, this was an ideal way to write the book.

Finally, we each have those in our personal lives who have contributed in some way to the creation of this volume.

The third author would like to thank Wolfgang Hillebrandt and Emmanuel Gangler for providing unique working environments which enabled the completion of this project. In addition, she would like to acknowledge all those who supported the Cosmostatistics Initiative and its Residence Programs, where many applications described in this volume were developed. Special thanks to Alberto Krone-Martins, Alan Heavens, Jason McEwen, Bruce Bassett, and Zsolt Frei as well as her fellow co-authors.

The second author thanks all members of the Cosmostatistics Initiative. Particular thanks to Ewan Cameron, Alberto Krone-Martins, Maria Luiza Linhares Dantas, Madhura Killedar, and Ricardo Vilalta, who have demonstrated their incredible support for our endeavor.

The first author wishes to acknowledge Cheryl Hilbe, his wife, who has supported his taking time from other family activities to devote to this project. In addition, he also wishes to expressly thank Eric Feigelson for advice and support over the past seven years as he learned about the astrophysical community and the unique concerns of astrostatistics. Professor Feigelson's experience and insights have helped shape how he views the discipline. He also acknowledges the true friendship which has evolved between the authors of this volume, one which he looks forward to continuing in the coming years. Finally, he dedicates this book to two year old Kimber Lynn Hilbe, his granddaughter, who will likely be witness to a future world that cannot be envisioned by her grandfather.

1 Astrostatistics

1.1 The Nature and Scope of Astrostatistics

Astrostatistics is at the same time one of the oldest disciplines, and one of the youngest. The Ionian Greek philosopher Thales of Miletus is credited with correctly predicting a total solar eclipse in central Lydia, which he had claimed would occur in May of 585 BCE. He based this prediction on an examination of records maintained by priests throughout the Mediterranean and Near East. The fact that his prediction was apparently well known, and the fact that the Lydians were engaged in a war with the Medes in Central Lydia during this period, brought his prediction notice and fame. Thales was forever after regarded as a sage and even today he is named the father of philosophy and the father of science in books dealing with these subjects.

Thales' success spurred on others to look for natural relationships governing the motions of astronomical bodies. Of particular note was Hipparchus (190–120 BCE) who, following on the earlier work of Aristarchus of Samos (310–230 BCE) and Eratosthenes (276–147 BCE), is widely regarded as the first to clearly apply statistical principles to the analysis of astronomical events. Hipparchus also is acknowledged to have first developed trigonometry, spherical trigonometry, and trigonometric tables, applying these to the motions of both the moon and sun. Using the size of the moon's parallax and other data from the median percent of the Sun covered by the shadow of the Earth at various sites in the area, he calculated the distance from the Earth to the Moon as well as from the Earth to the Sun in terms of the Earth's radius. His result was that the median value is 60.5 Earth radii. The true value is 60.3. He also calculated the length of the topical year to within six minutes per year of its true value.

Others in the ancient world, as well as scientists until the early nineteenth century, also used descriptive statistical techniques to describe and calculate the movements and relationships between the Earth and astronomical bodies. Even the first application of a normal or ordinary least squares regression was to astronomy. In 1801 Hungarian Franz von Zach applied the new least squares regression algorithm developed by Carl Gauss for predicting the position of Ceres as it came into view from its orbit behind the Sun.

The development of the first inferential statistical algorithm by Gauss, and its successful application by von Zach, did not immediately lead to major advances in inferential statistics.

Astronomers by and large seemed satisfied to work with the Gaussian, or normal, model for predicting astronomical events, and statisticians turned much of their attention to deriving various probability functions. In the early twentieth century William Gosset, Karl Pearson, and Ronald Fisher made the most advances in statistical modeling and hypothesis testing. Pearson developed the mathematics of goodness-of-fit and of hypothesis testing. The Pearson χ^2 test is still used as an assessment of frequentist based model fit.¹ In addition, Pearson developed a number of tests related to correlation analysis, which is important in both frequentist and Bayesian modeling. Fisher (1890–1962) is widely regarded as the father of modern statistics. He is basically responsible for the frequentist interpretation of hypothesis testing and of statistical modeling. He developed the theories of maximum likelihood estimation and analysis of variance and the standard way that statisticians understood p-values and confidence intervals until the past 20 years. Frequentists still employ his definition of the p-value in their research. His influence on twentieth century statistics cannot be overestimated.

It is not commonly known that Pierre-Simon Laplace (1749–1827) is the person foremost responsible for bringing attention to the notion of Bayesian analysis, which at the time meant employing inverse probability to the analysis of various problems. We shall discuss Bayesian methodology in a bit more detail in Chapter 3. Here we can mention that Thomas Bayes (1702–1761) developed the notion of inverse probability in unpublished notes he made during his lifetime. These notes were discovered by Richard Price, Bayes’s literary executor and a mathematician, who restructured and presented Bayes’ paper to the Royal Society. It had little impact on British mathematicians at the time, but it did catch the attention of Laplace. Laplace fashioned Bayes’ work into a full approach to probability, which underlies current Bayesian statistical modeling. It would probably be more accurate to call Bayesian methodology Bayes–Laplace methodology, but simplification has given Bayes the nominal credit for this approach to both probability and statistical modeling.

After enthusiastically promoting inverse probability, Laplace abandoned this work and returned to researching probability theory from the traditional perspective. He also made major advances in differential equations, including his discovery of Laplace transforms, which are still very useful in mathematics. Only a few adherents to the Bayesian approach to probability carried on the tradition throughout the next century and a half. Mathematicians such as Bruno de Finetti (Italy) and Harold Jeffreys (UK) promoted inverse probability and Bayesian analysis during the early part of the twentieth century, while Dennis Lindley (UK), Leonard J Savage (US), and Edwin Jaynes, a US physicist, were mainstays of the tradition in the early years of the second half of the twentieth century. But their work went largely unnoticed.

The major problem with Bayesian analysis until recent times has been related to the use of priors. Priors are distributions representing information from outside the model data that is incorporated into the model. We shall be discussing priors in some detail later in the text. Briefly though, except for relatively simple models the mathematics of calculating so-called posterior distributions was far too difficult to do by hand, or even by most computers, until computing speed and memory became powerful enough. This was particularly the case for Bayesian models, which are extremely demanding on computer power. The foremost reason why most statisticians and analysts were not interested in implementing Bayesian methods into their research was that computing technology was not advanced enough to execute more than fairly simple problems. This

was certainly the case with respect to the use of Bayesian methods in astronomy.

We shall provide a brief overview of Bayesian methodology in Chapter 3. Until such methods became feasible, however, Fisherian or frequentist methodology was the standard way of doing statistics. For those who are interested in understanding the history of this era of mathematics and statistics, we refer you to the book *Willful Ignorance: The Mismeasure of Uncertainty* (Weisberg 2014).

Astronomy and descriptive statistics, i.e., the mathematics of determining mean, median, mode, range, tabulations, frequency distributions, and so forth, were closely tied together until the early nineteenth century. Astronomers have continued to employ descriptive statistics, as well as basic linear regression, to astronomical data. But they did not concern themselves with the work being done in statistics during most of the twentieth century. Astronomers found advances in telescopes and the new spectroscope, as well as in calculus and differential equations in particular, to be much more suited to understanding astrophysical data than hypothesis testing and other frequentist methods. There was a near schism between astronomers and statisticians until the end of the last century.

As mentioned in the Preface, astrostatistics can be regarded as the statistical analysis of astronomical data. Unlike how astronomers utilized statistical methods in the past, primarily focusing on descriptive measures and to a moderate extent on linear regression from within the frequentist tradition, astrostatistics now entails the use, by a growing number of astronomers, of the most advanced methods of statistical analysis that have been developed by members of the statistical profession. However, we still see astronomers using linear regression on data that to a statistician should clearly be modeled by other, non-linear, means. Recent texts on the subject have been aimed at informing astronomers of these new advanced statistical methods.

It should be mentioned that we have incorporated astroinformatics under the general rubric of astrostatistics. Astroinformatics is the study of the data gathering and computing technology needed to gather astronomical data. It is essential to statistical analysis. Some have argued that astroinformatics incorporates astrostatistics, which is the reverse of the manner in which we envision it. But the truth is that gathering information without an intent to analyze it is a fairly useless enterprise. Both must be understood together. The International Astronomical Union (IAU) has established a new commission on astroinformatics and astrostatistics, seeming to give primacy to the former, but how the order is given depends somewhat on the interests of those who are establishing such names. Since we are focusing on statistics, although we are also cognizant of the important role that informatics and information sciences bring to bear on statistical analysis, we will refer to the dual studies of astrostatistics and astroinformatics as simply astrostatistics.

In the last few decades the size and complexity of the available astronomical information has closely followed Moore's law, which states roughly that computing processing power doubles every two years. However, our ability to acquire data has already surpassed our capacity to analyze it. The Sloan Digital Sky Survey (SDSS), in operation since 2000, was one of the first surveys to face the big data challenge: in its first data release it observed 53 million individual objects. The Large Synoptic Survey Telescope (LSST) survey is aiming to process and store 30 terabytes of data each night for a period of ten years. Not only must astronomers (astroinformaticists) deal with such

massive amounts of data but also it must be stored in a form in which meaningful statistical analysis can proceed. In addition, the data being gathered is permeated with environmental noise (due to clouds, moon, interstellar and intergalactic dust, cosmic rays), instrumental noise (due to distortions from telescope design, detector quantum efficiency, a border effect in non-central pixels, missing data), and observational bias (since, for example, brighter objects have a higher probability of being detected). All these problem areas must be dealt with prior to any attempt to subject the data to statistical analysis. The concerns are immense, but nevertheless this is a task which is being handled by astrostatisticians and information scientists.

1.2 The Recent Development of Astrostatistics

Statistical analysis has developed together with the advance in computing speed and storage. Maximum likelihood and all such regression procedures require that a matrix be inverted. The size of the matrix is based on the number of predictors and parameters in the model, including the intercept. Parameter estimates can be determined using regression for only very small models if done by hand. Beginning in the 1960s, larger regression and multivariate statistical procedures could be executed on mainframe computers using SAS, SPSS, and other statistical and data management software designed specifically for mainframe systems. These software packages were ported to the PC environment when PCs with hard drives became available in 1983. Statistical routines requiring a large number of iterations could take a long time to converge. But as computer speeds became ever faster, new regression procedures could be developed and programmed that allowed for the rapid inversion of large matrices and the solution to complex modeling projects.

Complex Bayesian modeling, if based on Markov chain Monte Carlo (MCMC) sampling, was simply not feasible until near the turn of the century. By 2010 efficient Bayesian sampling algorithms were implemented into statistical software, and the computing power by then available allowed astronomers to analyze complex data situations using advanced statistical techniques. By the closing years of the first decade of this century astronomers could take advantage of the advances in statistical software and of the training and expertise of statisticians.

As mentioned before, from the early to middle years of the nineteenth century until the final decade of the twentieth century there was little communication between astronomers and statisticians. We should note, though, that there were a few astronomers who were interested in applying sophisticated statistical models to their study data. In 1990, for example, Thomas Loredo wrote his PhD dissertation in astrophysics at the University of Chicago on “From Laplace Supernova SN 1987A: Bayesian inference in astrophysics” (see the book [Feigelson and Babu, 2012a](#)), which was the first thorough application of Bayesian modeling to astrophysical data. It was, and still is, the seminal work in the area and can be regarded as one of the founding articles in the new discipline of astrostatistics.

In 1991 Eric Feigelson and Jogesh Babu, an astronomer and statistician respectively at Pennsylvania State University, collaboratively initiated a conference entitled *Statistical Challenges in Modern Astronomy*. The conference was held at their home institution and brought together astronomers and a few statisticians for the purpose of collaboration. The goal was also to find a forum to teach astronomers how to use appropriate statistical analysis for their study projects.

These conferences were held every five years at Penn State until 2016. The conference site has now shifted to Carnegie Mellon University under the direction of Chad Schafer. During the 1990s and 2000s a few other conferences, workshops, and collaborations were held that aimed to provide statistical education to astronomers. But they were relatively rare, and there did not appear to be much growth in the area.

Until 2008 there were no astrostatistics working groups or committees authorized under the scope of any statistical or astronomical association or society. Astrostatistics was neither recognized by the IAU nor recognized as a discipline in its own right by the principal astronomical and statistical organizations. However, in 2008 the first interest group in astrostatistics was initiated under the International Statistical Institute (ISI), the statistical equivalent of the IAU for astronomy. This interest group, founded by the first author of the book, and some 50 other interested astronomers and statisticians, met together at the 2009 ISI World Statistics Congress in Durban, South Africa. The attendees voted to apply for the creation of a standing committee on astrostatistics under the auspices of the ISI. The proposal was approved at the December 2009 meeting of the ISI executive board. It was the first such astrostatistics committee authorized by an astronomical or statistical society.

In the following month the ISI committee expanded to become the ISI Astrostatistics Network. Network members organized and presented two invited sessions and two special theme sessions in astrostatistics at the 2011 World Statistics Congress in Dublin, Ireland. Then, in 2012, the International Astrostatistics Association (IAA) was formed from the Network as an independent professional scientific association for the discipline. Also in 2012 the Astrostatistics and Astroinformatics Portal (ASAIP) was instituted at Pennsylvania State under the editorship of Eric Feigelson and the first author of this volume. As of January 2016 the IAA had over 550 members from 56 nations, and the Portal had some 900 members. Working Groups in astrostatistics and astroinformatics began under the scope of IAU, the American Astronomical Society, and the American Statistical Association. In 2014 the IAA sponsored the creation of its first section, the Cosmostatistics Initiative (COIN), led by the second author of this volume. In 2015 the IAU working group became the IAU Commission on Astroinformatics and Astrostatistics, with Feigelson as its initial president. Astrostatistics, and astroinformatics, is now a fully recognized discipline. Springer has a series on astrostatistics, the IAA has begun agreements with corporate partnerships, Cambridge University Press is sponsoring IAA awards for contributions to the discipline, and multiple conferences in both astrostatistics and astroinformatics are being held – all to advance the discipline. The IAA headquarters is now at Brera Observatory in Milan. The IAA website is located at: <http://iaa.mi.ox-brera.inaf.it> and the ASAIP Portal URL is <https://asaip.psu.edu>. We recommend the readers of this volume to visit these websites for additional resources on astrostatistics.

1.3 What is a Statistical Model?

Statistics and statistical modeling have been defined in a variety of ways. We shall define it in a general manner as follows:

Statistics may be generically understood as the science of collecting and analyzing data for the purpose of classification, prediction, and of attempting to quantify and understand the uncertainty inherent in phenomena underlying data

Note that this definition ties data collection and analysis under the scope of statistics. This is analogical to how we view astrostatistics and astroinformatics. In this text our foremost interest is in parametric models. Since the book aims to develop code and to discuss the characterization of a number of models, it may be wise to define what is meant by statistics and statistic models. Given that many self-described data scientists assert that they are not statisticians and that statistics is dying, we should be clear about the meaning of these terms and activities.

Statistical models are based on probability distributions. Parametric models are derived from distributions having parameters which are estimated in the execution of a statistical model. Non-parametric models are based on empirical distributions and follow the natural or empirical shape of the data. We are foremost interested in parametric models in this text.

The theoretical basis of a statistical model differs somewhat from how analysts view a model when it is estimated and interpreted. This is primarily the case when dealing with models from a frequentist-based view. In the frequentist tradition, the idea is that the data to be modeled is generated by an underlying probability distribution. The researcher typically does not observe the entire population of the data that is being modeled but rather observes a random sample from the population data, which itself derives from a probability distribution with specific but unknown fixed parameters. The parameters specifying both the population and sample data consist of the distributional mean and one or more shape or scale parameters. The mean is regarded as a location parameter. For the normal model the variance σ^2 is the scale parameter, which has a constant value across the range of observations in the data. Binomial and count models have a scale parameter but its value is set at unity. The negative binomial has a dispersion parameter, but it is not strictly speaking a scale parameter. It adjusts the model for extra correlation or dispersion in the data. We shall address these parameters as we discuss various models in the text.

In frequentist-based modeling the slopes or coefficients that are derived in the estimation process for explanatory predictors are also considered as parameters. The aim of modeling is to estimate the parameters defining the probability distribution that is considered to generate the data being modeled. The predictor coefficients, and intercept, are components of the mean parameter.

In the Bayesian tradition parameters are considered as randomly distributed, not as fixed. The data is characterized by an underling probability distribution but each parameter is separately estimated. The distribution that is used to explain the predictor and parameter data is called the likelihood. The likelihood may be mixed with outside or additional information known from other studies or obtained from the experience or background of the analyst. This external information, when cast as a probability distribution with specified parameters, is called the prior distribution. The product of the model (data) likelihood and prior distributions is referred to as the posterior distribution. When the model is simple, the posterior distribution of each parameter may be be analytically calculated. However, for most real model data, and certainly for astronomical data, the posterior must be determined through the use of a sampling algorithm. A variety of MCMC sampling algorithms or some version of Gibbs sampling are considered to be the standard sampling algorithms used in Bayesian modeling. The details of these methods go beyond the scope of this text. For those interested in sampling algorithms we refer you to the articles [Feroz and Hobson \(2008\)](#) and

Foreman-Mackey *et al.* (2013) and the books Gamerman and Lopes (2006), Hilbe and Robinson (2013), and Suess and Trumbo (2010).

1.4 Classification of Statistical Models

We mentioned earlier that statistical models are of two general varieties – parametric and non-parametric. Parametric models are based on a probability distribution, or a mixture of distributions. This is generally the case for both frequentist-based and Bayesian models. Parametric models are classified by the type of probability distribution upon which a model is based. In Figure 1.1 we provide a non-exhaustive classification of the major models discussed in this volume.

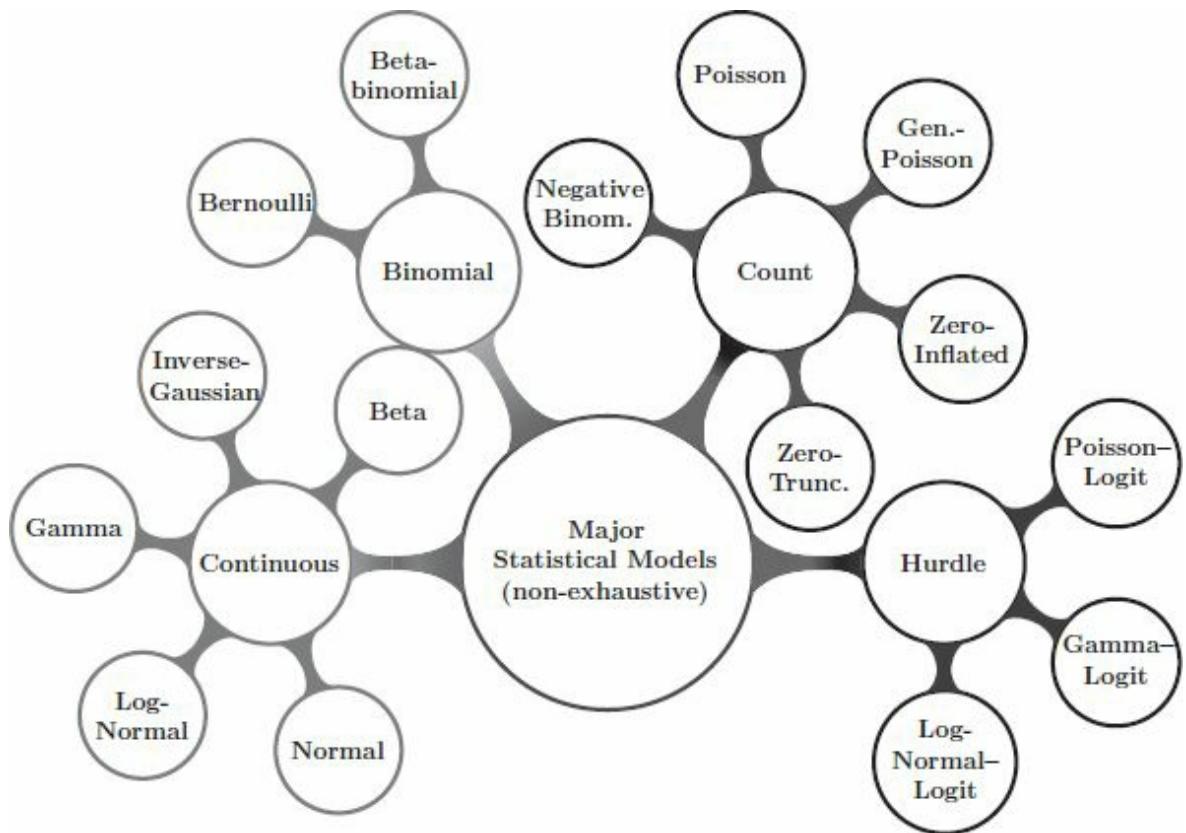


Figure 1.1 Major classifications of statistical regression models.

Astronomers utilize, or should utilize, most of these model types in their research. In this text examples will be given, and code provided, for all. We aim to show how only relatively minor changes need to be given to the basic estimation algorithm in order to develop and expand the models presented in Figure 1.1. Regarding data science and statistics, it is clear that when one is engaged in employing statistical models to evaluate data – whether for better understanding of the data or to predict observations beyond it – one is doing statistics. Statistics is a general term characterizing both descriptive and predictive measures of data and represents an attempt to quantify the uncertainty inherent in both the data being evaluated and in our measuring and modeling mechanisms. Many statistical tools employed by data scientists can be of use in astrostatistics, and we encourage astronomers and traditional statisticians to explore how they can be used to obtain a

better evaluation of astronomical data. Our focus on Bayesian modeling can significantly enhance this process.

Although we could have begun our examination of Bayesian models with single-parameter Bernoulli-based models such as Bayesian logistic and probit regression, we shall first look at the normal or Gaussian model. The normal distribution has two characteristic parameters, the mean or location parameter and the variance or scale parameter. The normal model is intrinsically more complex than single-parameter models, but it is the model most commonly used in statistics – from both the frequentist and Bayesian traditions. We believe that most readers will be more familiar with the normal model from the outset and will have worked with it in the past. It therefore makes sense to begin with it. The Bayesian logistic and probit binary response models, as well as the Poisson count model, are, however, easier to understand and work with than the normal model.

Further Reading

- Feigelson, E. D. and J. G. Babu (2012). *Statistical Challenges in Modern Astronomy V*. Lecture Notes in Statistics. Springer.
- Hilbe, J. M. (2012). “Astrostatistics in the international arena.” In: *Statistical Challenges in Modern Astronomy V*, eds. E. D. Feigelson and J. G. Babu. Springer, pp. 427–433.
- Hilbe, J. M. (2016). “Astrostatistics as new statistical discipline – a historical perspective.” www.worldofstatistics.org/files/2016/05/WOS_newsletter_05252016.pdf (visited on 06/16/2016).
- McCullagh, P. (2002). “What is a statistical model?” *Ann. Statist.* 30(5), 1225–1310. DOI: [10.1214/aos/1035844977](https://doi.org/10.1214/aos/1035844977).
- White, L. A. (2014). “The rise of astrostatistics.” www.symmetrymagazine.org/article/november-2014/the-rise-of-astrostatistics (visited on 06/16/2016).

¹ In Chapter 5 we shall discuss how this statistic can be used in Bayesian modeling as well.

2 Prerequisites

2.1 Software

The subtitle of this book is “using R, JAGS, Python, and Stan.” These software packages are used by astronomers more than any other Bayesian modeling software. Other packages are commonly used by those in other disciplines, e.g., WinBUGS, OpenBUGS, MLwiN, Minitab, SAS, SPSS, and recently, Stata. Minitab, SAS, SPSS, and Stata are general commercial statistical packages. WinBUGS is no longer being supported, future development being given to OpenBUGS. It is freeware, as are R, JAGS, Python, and Stan. MLwiN provides hierarchical and Bayesian hierarchical modeling capabilities and is free for academics.

In this chapter we provide an overview of each package discussed in this text. JAGS and Stan can be run within the R or Python environment. Most scripts discussed in this book use JAGS from within R and Stan from within Python. It is important, however, to state that this is merely a presentation choice and that the alternative combination (Stan from R and JAGS from Python) is also possible. We chose the first combination for didactic reasons, as an opportunity for the interested reader to familiarize themselves with a second programming language. In two different contexts (Chapters 8 and 10) we also show how Stan can be used from within R.

The R environment has become the most popular all purpose statistical software worldwide. Many statistical departments require their graduate students to learn R before gaining an advanced degree. Python, however, has quickly been gaining adherents. It is a powerful and user-friendly tool for software development but does not have nearly as many already supported procedures as R.

Astronomers are pretty much split between using R and Python for the statistical analysis of their study data. Each software package has its own advantage. R can be used by itself for a large range of statistical modeling tasks, but until recently its Bayesian capability has been limited to relatively simple models. It is certainly possible, however, for more complex Bayesian models to be written in R, but for the most part R has been used as a framework within which specific Bayesian packages are run, e.g., JAGS, INLA, bmsr, and even Stan. Bayesian astronomers nearly always turn to JAGS or Python for executing complex models. Stan is a new tool that is quickly gaining popularity. Again, when using JAGS it is easiest to run it from within R so that other statistical analysis using R can easily be run on the data. Likewise, Stan can be run from within R or Python,

as we shall observe in this volume.

We will demonstrate Bayesian models using R, JAGS, Python, and Stan, providing the code for each in order to allow readers the ability to adapt the code for their own purposes. In some cases, though, we will provide JAGS code within R, or Stan from within Python, without showing the code for a strictly R or Python model. The reason is that few researchers would want to use R or Python alone for these models but would want to use JAGS and R, or Stan and Python together, for developing a useful Bayesian model with appropriate statistical tests. In fact, using a Bayesian package from within R or Python has become a fairly standard way of creating and executing Bayesian models for most astronomers.

Note also that the integrated nested Laplace approximation (INLA) Bayesian software package is described in Appendix A of this book. The INLA algorithm samples data by approximation, not by MCMC-based methods. Therefore the time required to estimate most Bayesian models is measured in seconds and even tenths of a second, not minutes or hours.

The INLA package is becoming popular in disciplines such as ecology, particularly for its Bayesian spatial analysis and additive modeling capabilities. However, there are important limitations when using INLA for general-purpose Bayesian modeling, so we are not providing a discussion of the package in the main part of the text. We felt, however, that some mention should be made of the method owing to its potential use in the spatial analysis of astrophysical data. As INLA advances in its capabilities, we will post examples of its use on the book's web site.

Many models we describe in this book have not been previously discussed in the literature, but they could well be used for the better understanding of astrophysical data. Nearly all the models examined, and the code provided, are for non-linear modeling endeavors.

2.2 R

The R package is a general purpose statistical software environment. It can be downloaded from the Comprehensive R Archive Network (CRAN)¹ without cost. The software is maintained by the R Development Core Team (see the manual [R Development Core Team, 2008](#)), a group of statistical programmers who freely provide their services to manage and direct the future of the package. Except for a core of statistical routines that serves as the base package, most of the advanced statistical procedures or functions have been developed by users.

Packages newly submitted to CRAN are run through a software check to make certain that functions produce as advertised and that the appropriate help and ancillary files are working. Every attempt to maintain accuracy is made. As of 18 June 2016 there are 8611 packages on CRAN, but only relatively few are actually used by statisticians for their studies. Many CRAN packages provide support materials for published books but a number are orphaned, i.e., they themselves are no longer supported.

Currently the majority of books being published on statistical modeling, or on statistical analysis

in general, use R for examples. Moreover, the majority of journal articles in statistics use R for demonstrating a statistical procedure. We recommend that astronomers know how to engage in statistical modeling using R even if they prefer other languages for their personal research analysis.

Earlier, we intimated that R packages containing R functions for Bayesian modeling are rather limited as of the time of writing. We show examples of using R for running various Bayesian models, but by far the majority of R users employ JAGS or OpenBUGS within the R environment when creating and running Bayesian models. Astronomers and those in the physical sciences appear to use JAGS from within R in preference to OpenBUGS, which is commonly used in the social sciences. In any case, many astronomers use R or Python for both data management and general statistical analysis, as well as for Bayesian modeling.

We highly recommend that, when using R for building and evaluating a model, the code be typed or pasted into the R editor. Click on `File` and then on `New Script` in the R menu selection at the top of the program window. Sections of the code or the entire program can be run by selecting the code to be executed, clicking on the right side of the mouse to display a menu selection, and clicking on `Run line` or `selection`. The output will be displayed on the R main screen. It is easy to amend the code and to rerun the code if necessary. When demonstrating R and JAGS code we assume that the code is being run from the R editor. If we provide a line of R code with `>` in front of the code, it is to be understood that this is typed on the R command line and is not in the editor.

The R code below demonstrates how to develop synthetic model variables and how to run a simple linear regression on them. The final two lines set up a graphics page for four figures and then display a fitted vs. residuals plot, a normal QQ plot, a scale-location plot, and lastly a residuals vs. leverage plot, as shown in Figure 2.1. These default residual diagnostic plots are displayed by simply typing `plot()`, with the name of the regression models placed between the parentheses. Note that, in the code below, the `c()` operator places the elements typed within the parentheses into a vector. The elements must be of the same type or, in R language, mode. The “less than” symbol with a negation sign attached to the right (`<-`) is an assignment operator. The more traditional “`=`” sign may also be used, but `<-` is more common. We first create three variables, with `y` as the model response (dependent) variable and the others as predictors. The `lm` operator activates the linear model function. The comments to the right of the code explain the operations.

Code 2.1 Example of linear regression in R.

```
=====
# Data
y <- c(13,15,9,17,8,5,19,23,10,7,10,6) # response variable
x1 <- c(1,1,1,1,1,0,0,0,0,0,0,0)        # binary predictor
x2 <- c( 1,1,1,1,2,2,2,2,3,3,3,3)       # categorical predictor

# Fit
mymodel <- lm(y ~ x1 + x2)    # linear regression of y on x1 and x2

# Output
summary(mymodel)      # summary display
par(mfrow=c(2, 2))    # create a 2 by 2 window
plot(mymodel)         # display of fitted vs. residuals plot, normal QQ plot
                      # scale-location plot and residuals vs. leverage plot
=====
```

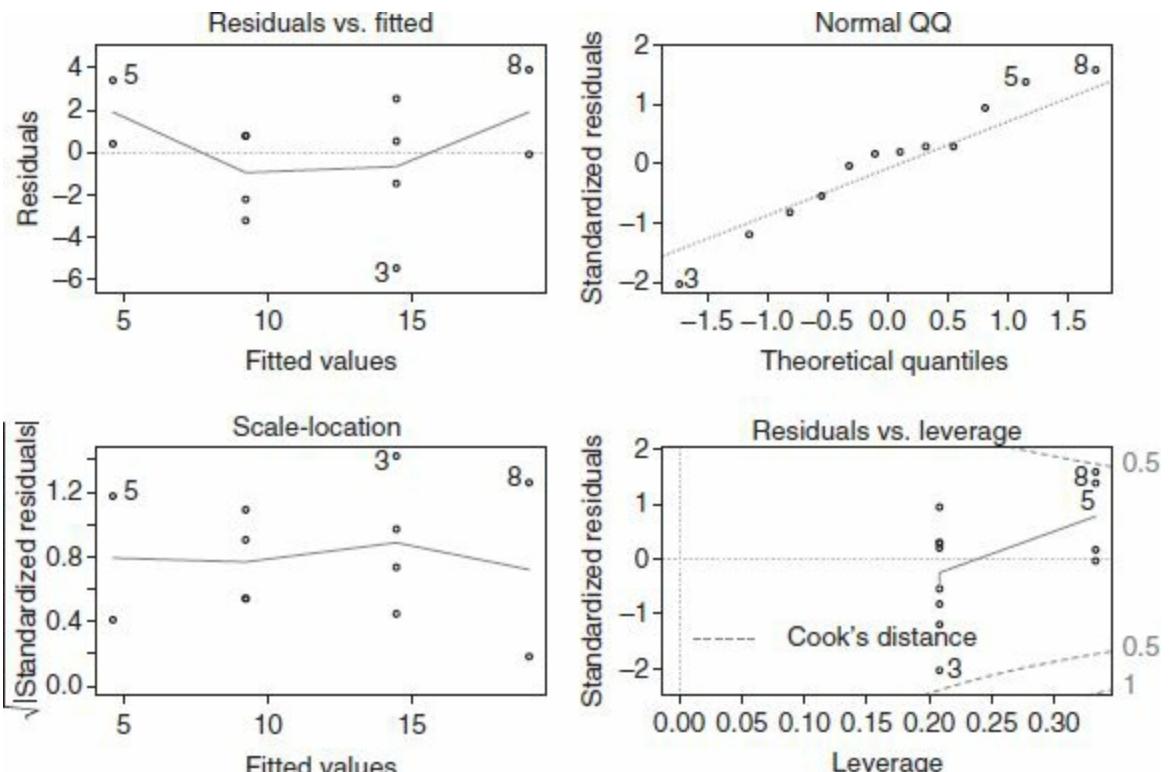


Figure 2.1 Default residual diagnostic plots in R.

Table 2.1 lists the main packages that you should obtain in order to run the examples in the subsequent chapters. All the packages can be installed through CRAN using the command `install.packages()`. Note that JAGS needs to be installed outside R. The function `jagsresults` can be loaded from your local folder and is part of the package `jagstools` available on Github:²

```
> source("../auxiliar_functions/jagsresults.R")
```

Table 2.1 List of the main R packages which should be installed in order to run the examples shown in subsequent chapters.

Name of package	Description
JAGS ^a	Analysis of Bayesian hierarchical models
R2jags ^b	R interface to JAGS
rstan ^c	R interface to Stan
lattice ^d	Graphics library
ggplot2 ^e	Graphics library
MCMCpack ^f	Functions for Bayesian inference
mcmcplots ^g	Plots for MCMC Output

^a<http://mcmc-jags.sourceforge.net>,

^b<https://cran.r-project.org/web/packages/R2jags>,

^c<https://cran.r-project.org/web/packages/rstan/index.html>,

^d<http://lattice.r-forge.r-project.org/>,

^e<http://ggplot2.org>,

^f<http://mcmcpack.berkeley.edu>,

^g<https://cran.r-project.org/web/packages/mcmcplots>

2.3 JAGS

JAGS is an acronym for *Just Another Gibbs Sampler*. It was designed and is maintained by Martyn Plummer and is used strictly for Bayesian modeling. The key to running JAGS from within R is to install and load a library called `r2jags`. The code is similar in appearance to R. Typically, when run from within R, the data is brought into the program using R. The JAGS code begins by defining a matrix for the vector of model predictors, x . In Code 2.2, we show how a typical JAGS model looks.

Code 2.2 Bayesian linear regression in JAGS.

```
=====
# Data
X <- model.matrix(~ x1 + x2, data = mymodel) # place all predictors into
                                                matrix
K <- ncol(X)
model.data <- list(Y = mymodel$y, N = nrow(mymodel), X = X, K = K)
sink("MOD.txt")                                # write data to a file

# Fit
cat("model{
    # priors

        for (i in 1:N){
            # log-likelihood

        }
    }", fill = TRUE)
sink()
inits <- function () {
    list(
        # Initial values
    )
}

}
params <- c({ "beta", others })      # parameters to be displayed in output
J <- jags(data = model.data,          # main JAGS sampling
           inits = inits,
           parameters = params,
           model.file = "MOD.txt",
           n.thin = 1,                      # number of thinning
           n.chains = 3,                    # 3 chains
           n.burnin = 10000,                # burn-in sample
           n.iter = 20000)                 # samples used for posterior

# Displays model output
print(J)   # or for a nicer output and defined credible intervals
print(J, intervals=c(0.025, 0.975), digits=3)
=====
```

Analysts have structured codes for determining the posterior means, standard deviations, and credible intervals for the posterior distributions of Bayesian models in a variety of ways. We shall implement a standard format or paradigm for developing a Bayesian model using JAGS. The same structure will be used, with some variations based on the peculiarities of the posterior distribution being sampled. We shall walk through an explanation of a Bayesian normal model in Chapter 4.

2.4 Python

Python is an object-oriented programming language created by the Dutch computer programmer

Guido van Rossum³ in the early 1990s. The name is a reference to the British comedy show *Monty Python’s Flying Circus* and was only associated with the snake symbol afterwards.⁴ Python is based on ABC, a Dutch programming language designed for teaching those with no background in software development. Beyond the user-friendly philosophy guiding the development of the language, extensibility is one of the most important aspects, and this allowed the rapid growth of its currently large community of users and developers. Guido van Rossum is up to this date known within the community as “benevolent dictator for life,” having the final word in what goes into Python. Throughout its development until version 2.7, Python releases were reasonably backward compatible, a feature which changed in Python 3. In order to avoid compatibility issues all examples shown in this book can be run in either versions, except for the Bayesian models using the `rpy2` package.

In astronomy, Python has become increasingly popular, especially since the late 1990s when the Space Telescope Science Institute⁵ adopted it as the base language from which to extend or substitute the existing astronomical data analysis tools. Since then multiple efforts have been employed by different groups, aiming at the construction of Python-based astronomical packages (e.g. Astropy,⁶ AstroML,⁷ etc.). It is not our goal to present here a detailed description of the language, but we do provide all the necessary ingredients to run the Python examples and to compare them with their equivalent R counterparts. The Python codes in this volume can be used as presented. They were designed to be intuitive for the beginning Python user or the R expert. We refer the reader to the book [Ivezić, Connolly, Vanderplas *et al.* \(2014, Appendix IV\)](#) and references therein for a more detailed introduction to the capabilities of Python in the astronomical context. In what follows we describe only the basic features needed to run the examples provided.

Python can be installed in all main operating systems by following the instructions on the official website.⁸ Once you are sure you have the desired version of Python it is also important to get `pip`, the recommended installer for Python packages.⁹ The official website can guide you through the necessary steps.¹⁰ Windows users may prefer the graphical interface `pip-Win`.¹¹

You will now be able to install a few packages which will be used extensively for examples in subsequent chapters. Table 2.2 lists the packages¹² you should get in order to run most of the examples.¹³ All the packages can be installed through `pip`. Beyond the tools listed in Table 2.2, we also recommend installing IPython,¹⁴ an enhanced interactive Python shell or environment which makes it easier for the beginning user to manipulate and check small portions of the code. It can be installed via `pip` and accessed by typing `$ ipython` on the command line.

Table 2.2 List of Python packages which should be installed in order to run the examples shown in subsequent chapters.

Name of package	Description
<code>matplotlib^a</code>	Plotting library
<code>numpy^b</code>	Basic numerical library
<code>pandas^c</code>	Data structure tools
<code>pymc3^d</code>	Full Python probabilistic programming tool

pystan ^e	Python interface to Stan
scipy ^f	Advanced numerical library
statsmodels ^g	Statistics library

^a<http://matplotlib.org/>

^bwww.numpy.org/

^c<http://pandas.pydata.org/>

^d<https://pymc-devs.github.io/pymc3/>

^e<https://pystan.readthedocs.org/en/latest/>

^fwww.scipy.org/

^g<http://statsmodels.sourceforge.net/>

Throughout this volume, Python-related code will be always presented as complete scripts. This means that the reader can copy the entire script in a .py file using the text editor of choice and run the example on the command line by typing

```
python my_file.py
```

It is also possible to run the script from within IPython by typing

```
In [1]: %run my_file.py
```

This is highly recommended since the Python shell stores results and information about intermediate steps which can be used for further analysis. The same result can be achieved by pasting code directly into the shell. This last alternative is appropriate if you wish to run parts of the script separately for checking, plotting, etc. In order to paste code snippets directly into the shell use

```
In [1]: %paste
```

Keep in mind that the standard `ctrl+v` shortcut *will not* work properly within IPython. Finally, we emphasize that code snippets are divided into sections. The three basic components are `# Data`, `# Fit`, and `# Output`. The `# Data` section shows all the steps necessary for data preparation (how to simulate simple synthetic data for introductory examples, how to read real data from standard astronomical data formats, or how to prepare the data to be used as input for more complex modules). Sometimes, even in cases where previously generated toy data is reused, the `# Data` section is shown again. This is redundant, but it allows the reader to copy and run entire snippets without the need to rebuild them from different parts of the book. Once data are available, the `# Fit` section shows examples of how to construct the model and perform the fit. The `# Output` section shows how to access the results via screen summaries or graphical representations.

Note that we will not present extracts or single-line Python code beyond this chapter. Sometimes such code will be shown in R for the sake of argument or in order to highlight specific parts of a model. However, we believe that presenting such extracts in both languages would be tedious. Given the similarities between R and Python we believe that the experienced Python user will find no difficulties in interpreting these lines.

As an example, we show below the equivalent Python code for the example of linear regression presented in Section 2.2.

The `import` statements in the first two lines load the packages which will be used in the calculations below. You can also recognize the sections of the code we mentioned before. Each Python package has a specific format in which the input data should be passed to its main functions. In our example, `statsmodels` require data organized as a dictionary (`mydata` variable in Code 2.3).

Code 2.3 Example of linear regression in Python.

```
=====
import numpy as np
import statsmodels.formula.api as smf

# Data
y = np.array([13,15,9,17,8,5,19,23,10,7,10,6]) # response variable
x1 = np.array([1,1,1,1,1,0,0,0,0,0,0,0])        # binary predictor
x2 = np.array([1,1,1,2,2,2,2,3,3,3,3,3])        # categorical predictor

mydata = {}                                       # create data dictionary
mydata['x1'] = x1
mydata['x2'] = x2
mydata['y'] = y

# Fit
results = smf.ols(formula='y ~ x1 + x2', data=mydata).fit()

# Output
print(str(results.summary()))
=====
```

Here we will focus on Bayesian modeling (see Chapter 3). Our goal is to demonstrate Python's capabilities for Bayesian analysis when using pure Python tools, e.g. `pymc3`, and in its interaction with Stan (Section 2.5). Through our examples of synthetic and real data sets we shall emphasize the latter, showing how Python and Stan can work together as a user-friendly tool allowing the implementation of complex Bayesian statistical models useful for astronomy. This will be done using the `pystan` package, which requires the data to be formatted as a dictionary. The package may be called using an expression such as

```
In [1]: fit = pystan.stan(model_code=stan_code, data=mydata)
```

or

```
In [1]: fit = pystan.stan(file='stan_code.txt', data=mydata)
```

In the first option, `model_code` receives a multiple-line string specifying the statistical model to be fitted to the data (here called `stan_code`, see Section 2.5) and in the second option `file` receives the name of the file where the model is stored. In what follows we will always use the first option in order to make the model definition an explicit part of our examples. However, the reader may find the second option safer from an organizational point of view, especially when dealing with more complex situations. A closer look at Stan and its modeling capabilities is given below.

2.5 Stan

Stan is a probabilistic programming language whose name was chosen in honor of Stanislaw Ulam, one of the pioneers of the Monte Carlo method. It is written in c++ with available interfaces in R, Python, Stata, Julia, MatLab, and the shell (CmD). As we mentioned before, we will provide

examples of Stan code using its Python interface `pystan` and in a few situations using its R interface, `rstan`. However, the models can be easily adapted to other interfaces. Stan uses a variation of the Metropolis algorithm called Hamiltonian Monte Carlo, which allows for an optimized sampling of non-standard posterior distributions (see the book Kruschke, 2015). It is a recent addition to the available Bayesian modeling tools. The examples presented in this volume were implemented using version 2.14.

Before we begin, it is necessary to know what kinds of element we are allowed to handle when constructing a model. Stan¹⁵ provides a large range of possibilities and in what follows we will briefly describe only the features used to manipulate the models described in this book.

A complete Stan model is composed of six code blocks (see Table 2.3). Each block must contain instructions for specific tasks. In Section 2.4 we explained how a Stan code block can be given as input to the Python package `pystan`. Here we will describe each code block, independently of the Python interface, and highlight its role within the Stan environment.

Table 2.3 Description of Stan code blocks.

Name of code block	Purpose
Data	External input
Transformed data	Data pre-processing
Parameters	Guidelines for sampling space determination
Transformed parameters	Parameter pre-processing
Model	Guidelines for building posterior
Generated quantities	Post-processing

The main components of a Stan code are as follows:

- The `data` block holds the declaration for all input data to be subsequently used in the Stan code. The user must define within this block all variables which will store different elements of the data. Notice that the actual connection between the data dictionary and Stan is done on the fly through the `pystan` package. An example of a simple `data` block is

```
data{
  int<lower=0> nobs;    # input integer
  real beta0;            # input real
  int x[nobs];          # array of integers
  vector[nobs] y;       # vector of floats
}
```

Here `x` is an array of integers and `y` is a vector. In Stan, arrays are one-dimensional structures of any type, while vectors are necessarily one-dimensional collections of reals. For matrix arithmetic, linear algebra, and multivariate distributions, only vectors are allowed. Notice that it is possible to introduce constraints in this block (`nobs` is non-negative). At this stage, upper and lower limits are used merely for checking. If an input violates the constraints declared in the `data` block the compilation is aborted before any calculation is performed.

- The `transformed data` block allows for data pre-processing. Any temporary variables used to store a transformation performed on the data *without the involvement of parameters*

should be defined here. For example,

```
transformed data {
  real xnew[nobs];                                # new temporary variable
  for (i in 1:nobs) xnew[i] = x[i] * beta0; # data only transformation
}
```

In the snippet above we also make use of a loop running through the vector elements with index `i`. This syntax will be useful throughout the book.

- The `parameters` block defines the domain to be sampled when calculating the posterior. This might require the use of upper and/or lower limits such as

```
parameters {
  real beta1;          # unconstrained variable
  real<upper=0> beta2; # beta2 <= 0
  real<lower=0> sigma; # sigma >= 0
}
```

The limits imposed on the parameters `beta2` and `sigma` are already a form of prior. Stan is able to handle improper priors, meaning that, if we do not define a prior for these parameters in the `model` block, it will consider the prior to be uniform over the entire domain allowed by the constraints. However, the reader should be aware of the numerical problems that might occur if the priors lead to improper posterior distributions.^{[16](#)}

- The `transformed parameters` block allows the construction of intermediate parameters which are built from the original ones and are frequently used to make the likelihood less verbose. This is the case for a linear predictor; for example,

```
transformed parameters {
  vector[nobs] lambda;
  for (i in 1:nobs) lambda[i] = beta1 * xnew[i] + beta2;
}
```

- The `model` block is the core of the code structure, in which the recipe for the posterior is given:

```
model {
  beta1 ~ normal(0,10);    # prior for beta1
  beta2 ~ normal(0, 10);   # prior for beta2
  sigma ~ gamma(1.0, 0.8); # gamma prior for
  y ~ normal(lambda, sigma); # likelihood
}
```

Here there is some freedom in the order in which the statements are presented. It is possible to define the likelihood before the priors. Note that we have an apparent contradiction here. Parameters `beta1` and `beta2` were assigned the same prior, but `beta1` can take any real value while `beta2` is non-positive (see our definition in the `parameters` block).

We chose this example to illustrate that, in Stan, prior distributions may be truncated through the combination of a constrained variable with an unbounded distribution. Thus, from the `parameters` and `model` blocks above, we imposed a normal prior over `beta1` and an half-normal prior over `beta2`. Moreover, we did not explicitly define a prior for `sigma` because, in the absence of a specific prior definition, Stan assigns improper priors over the entire variable domain. Thus, if we wish to use a non-informative prior over `sigma`, merely defining it with the appropriate constraint is sufficient although not recommended ([Stan, 2016](#)).

Another important point is the vectorization of sampling statements (y and `lambda` are vectors). If any other argument of a sampling statement is a vector, it needs to be of the same size. If the other arguments are scalars (as in the case of `sigma`), they are used repeatedly for all the elements of vector arguments (Team Stan, 2016, Chapter 6).

In this example we used the normal and gamma distributions for priors, but other distributions are also possible. Figure 2.2 shows the major distributions available in both JAGS and Stan, many of which will be covered throughout the book. We will also show examples of how to implement and customize distributions for non-standard data situations.

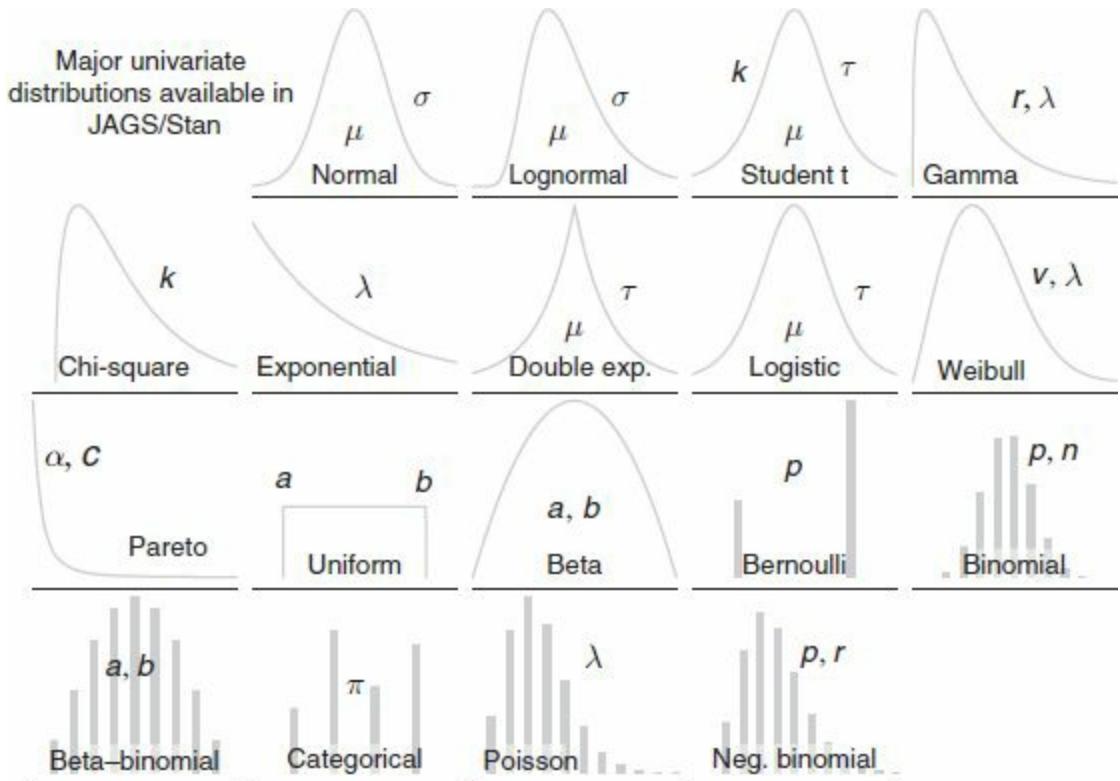


Figure 2.2 Major univariate distributions available in JAGS and Stan.

- The generated quantities block should be used when one wants to predict the value of the response variable, given values for the explanatory variable and the resulting posterior. In the case where we wish to predict the response variable value in the first 1000 values of the `xnew` vector, we have

```
generated quantities{
  vector[nobs] y_predict;
  for (n in 1:nobs)
    y_predict[n] = normal_rng(xnew[n] * beta1 + beta2, sigma);
}
```

In this block the distribution name must be followed by `_rng` (random number generator). This nomenclature can be used only inside the `generated quantities` block. Also important to emphasize is that, unlike the sampling statements in the `model` block, `_rng` functions cannot be vectorized.

Putting together all the data blocks in a single-string object allows us to use Python (or R) as an interface to Stan, as explained in Section 2.4. The online material accompanying this book contains a complete script allowing the use of the elements above in a synthetic model example. A similar code is presented in Chapter 4 in the context of a normal linear model.

Important Remarks

Before one starts manipulating codes in Stan it is important to familiarize oneself with the language style guide. We urge readers to go through Appendix C of the *Stan User’s Manual* (Team Stan, 2016) before writing their own Stan programs. The recommendations in that appendix are designed to improve readability and space optimization. In the examples presented in this volume we do our best to follow them and apologize in advance for eventual deviations.

The reader should also be aware of the types of errors and warning messages that might be encountered. Appendix D of Team Stan (2016) describes the messages prompted by the Stan engine (not those originating from R, Python, or other interpreters). We call special attention to a message which looks like this:

```
Information Message: The current Metropolis proposal is about to be rejected...Scale parameter is 0, but must be > 0!
```

This is a warning, not an error message. It is raised whenever there is an underflow or overflow in numerical computations, which are unavoidable when dealing with floating-point approximations to continuous real values. If you get this message sporadically and mostly during warm-up¹⁷ then you are probably fine. If it appears too often and/or during sampling then it is necessary to investigate the numerical stability of the model.

Finally, it is important to be aware that you might encounter some numerical issues when dealing with several parameters close to zero in pystan. This is related to how Python deals with divisions by very small numbers and, at the time of writing, there is no user-friendly work around available to help.¹⁸ If such situations are encountered then a possible approach is to use the same Stan model, but from within R. We show how this can be done in Chapter 10.

Further Reading

- Andreon, S. and B. Weaver (2015). *Bayesian Methods for the Physical Sciences: Learning from Examples in Astronomy and Physics*. Springer Series in Astrostatistics. Springer.
- Betancourt, M. (2016). “Some Bayesian modeling techniques in Stan.” www.youtube.com/watch?v=uSjsJg8fcwY (visited on 06/18/2016).
- Chattopadhyay, A. K. and T. Chattopadhyay (2014). *Statistical Methods for Astronomical Data Analysis*. Springer Series in Astrostatistics. Springer.
- Gelman, A., J. Carlin, H. Stern, D. Dunson, A. Vehtari, and D. Rubin (2013). *Bayesian Data Analysis, Third Edition*. Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis.
- Gelman, A., D. Lee, and J. Guo (2015). “Stan: a probabilistic programming language for Bayesian inference and optimization.” *J. Educational and Behavioral Statistics*.
- Ivezic, Z. et al. (2014). *Statistics, Data Mining, and Machine Learning in Astronomy: A Practical Python Guide for the Analysis of Survey Data*. EBSCO ebook academic collection. Princeton University Press.

- Korner-Nievergelt, F. et al. (2015). *Bayesian Data Analysis in Ecology Using Linear Models with R, BUGS, and Stan*. Elsevier Science.
- Kruschke, J. (2010). *Doing Bayesian Data Analysis: A Tutorial Introduction with R*. Elsevier Science.
- Lunn, D. et al. (2012). *The BUGS Book: A Practical Introduction to Bayesian Analysis*. Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis.
- McElreath, R. (2016). *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*. Chapman & Hall/CRC Texts in Statistical Science. CRC Press.
- Muenchen, R. A. and J. M. Hilbe (2010). *R for Stata Users*. Statistics and Computing. Springer.
- Teator, P. (2011). *R Cookbook*. O'Reilly Media.
- Zuur, A. F., J. M. Hilbe, and E. N. Ieno (2013). *A Beginner's Guide to GLM and GLMM with R: A Frequentist and Bayesian Perspective for Ecologists*. Highland Statistics.
-

¹ <https://cran.r-project.org/>

² <https://github.com/johnbaums/jagstools>

³ www.python.org/~guido/

⁴ “The history of Python” – <http://python-history.blogspot.com.br/2009/01/personal-history-part-1-cwi.html>

⁵ www.stsci.edu/portal/

⁶ www.astropy.org/

⁷ www.astroml.org/

⁸ [https://wiki.python.org/moin/BeginnersGuide/Download](http://wiki.python.org/moin/BeginnersGuide/Download)

⁹ If you have Python 2 to 2.7.9 or Python 3 to 3.4 downloaded from python.org then pip is already installed, but you have to upgrade it using `$ pip install -U pip`.

¹⁰ <https://pip.pypa.io/en/stable/>

¹¹ <https://sites.google.com/site/pydatalog/python/pip-for-windows>

¹² Installing `pystan` involves compiling Stan. This may take a considerable amount of time.

¹³ For a few models in Chapters 7 and 10, where we show how to call R functions from Python, you will additionally need to install R and the package `rpy2`.

¹⁴ <http://ipython.org/>

¹⁵ <http://mc-stan.org>

¹⁶ Please check the community recommendations and issues about hard priors in the article [Stan \(2016\)](#).

¹⁷ In Stan, “warm-up” is equivalent to the burn-in phase in JAGS. See the arguments in [Betancourt \(2015\)](#).

¹⁸ https://groups.google.com/forum/#topic/stan-users/hn4W_p8j3fs

3 Frequentist vs. Bayesian Methods

In this chapter we describe the frequentist-based and Bayesian approaches to statistical analysis. For most of the twentieth century the frequentist tradition dominated, but Bayesian methods are rapidly becoming mainstream in the early twenty-first century. The foremost reason for this is the growth in computer processor speed and memory. The creation of immensely more efficient sampling algorithms has also contributed to the advance of Bayesian methods. This combination has allowed for the creation of Bayesian models that can be used on standard as well as highly complex model designs.

3.1 Frequentist Statistics

Linear regression is the basis of a large part of applied statistics and has long been a mainstay of astronomical data analysis. In its most traditional description one searches for the best linear relation describing the correlation between x (the exploratory variable) and y (the response variable).

Approaching the problem through a physical–astronomical perspective, the first step is to visualize the data behavior through a scatter plot (Figure 3.1). On the basis of this first impression, the researcher concludes that there is a linear correlation between the two observed quantities and builds a model. This can be represented by

$$y_i = ax_i + b, \tag{3.1}$$

where x_i and y_i are the measured quantities,¹ with the index running through all available observations, $i = 1, \dots, n$, and $\{a, b\}$ the model parameters whose values must be determined.

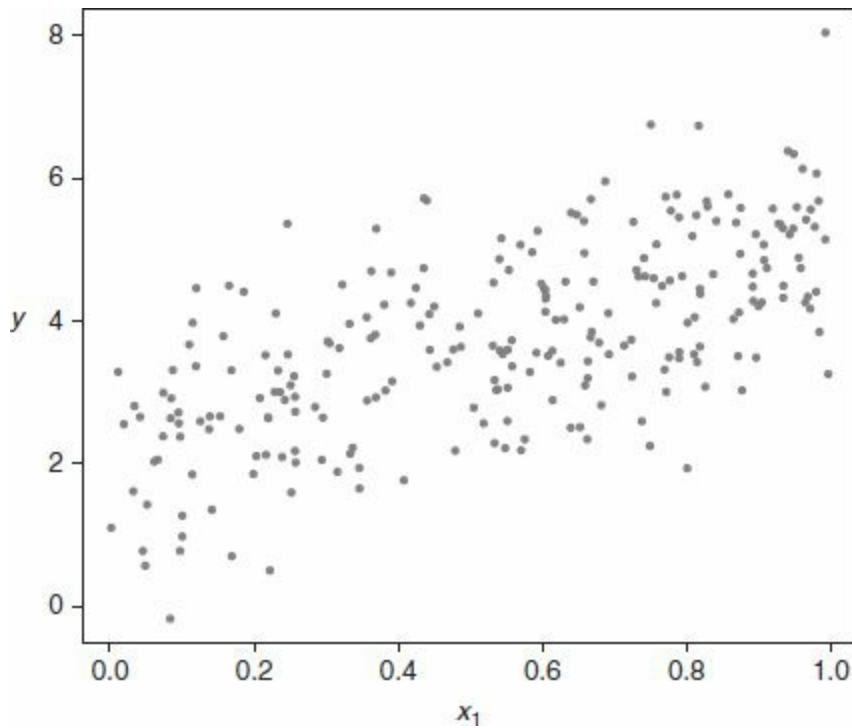


Figure 3.1 Example of toy data following a linear relation.

In the frequentist approach, the “true” values for the model parameters can be estimated by minimizing the residuals ε_i between the predicted and measured values of the response variable y for each measured value of the explanatory variable x (Figure 3.2). In other words the goal is to find values for a and b that minimize

$$\varepsilon_i = y_i - ax_i - b. \quad (3.2)$$

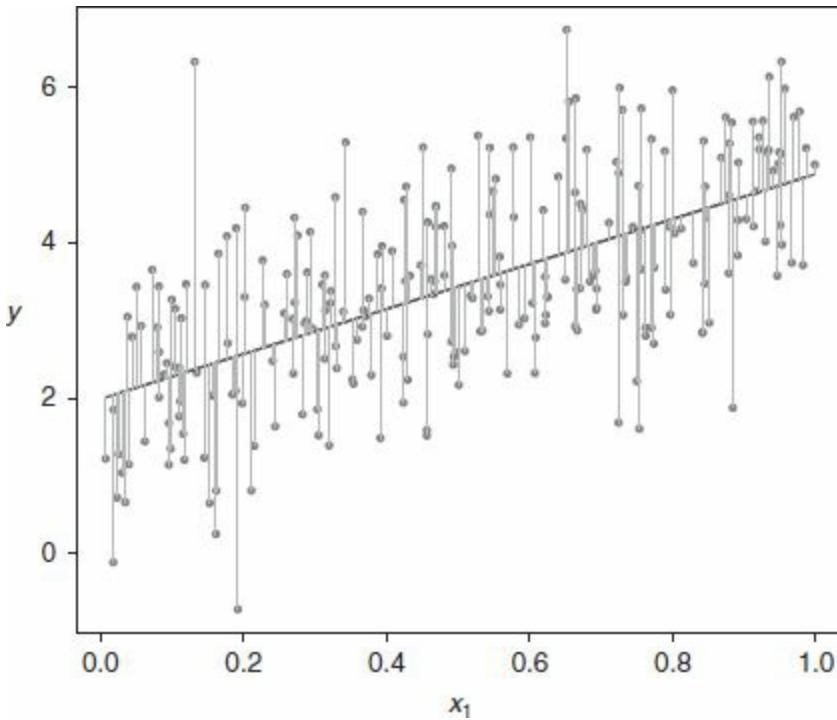


Figure 3.2 Results from linear regression applied to the toy data. The dots are synthetic observations, the solid diagonal line is the fitted regression line, and the gray vertical lines are the residuals.

Assuming that $\{\varepsilon_i\}$ is normally distributed and that observations are independent of each other, the likelihood of the data given the model is

$$L(\text{data}|a, b) = \prod_{i=1}^N \frac{1}{2\sqrt{\pi\sigma^2}} \exp\left[\frac{(y_i - ax_i - b)^2}{2\sigma^2}\right], \quad (3.3)$$

which is more commonly used in its logarithmic form,

$$\mathcal{L} \propto \frac{1}{2\sigma^2} \sum_{i=1}^N (y_i - ax_i - b)^2. \quad (3.4)$$

The maximum likelihood estimation (MLE) method aims to find the values of the parameters $\{a, b\}$ that minimize Equation 3.4. We shall discuss the likelihood in depth in subsequent sections. First, though, let us see how this simple example can be implemented.

3.1.1 Fitting a Linear Regression in R

We shall create a synthetic “toy” model to demonstrate basic linear modeling. Our simulated data set contains 250 observations having a random uniform predictor x_1 and a linear predictor x_b , which

is the sum of the intercept and the uniform predictor term x_1 times the slope. We assign an intercept value of $\alpha = 2$ and slope $\beta = 3$ and insert the linear predictor xb into a normal random number generator function, `rnorm`. In what follows, y is a normally distributed response variable internally adjusted so that the assigned slope or coefficient is produced. We use the `lm` function for linear models, encasing the function within a summary function so that results are automatically displayed on screen. We can use this format for creating a wide range of synthetic models, which we will see throughout the text. We suggest placing the R code into the R editor using [Files → new Script](#).

Code 3.1 Basic linear model in R.

```
=====
# Data
set.seed(1056)          # set seed to replicate example
nobs = 250               # number of obs in model
x1 <- runif(nobs)        # random uniform variable
alpha = 2                 # intercept
beta = 3                  # angular coefficient

xb <- alpha + beta * x1   # linear predictor, xb
y <- rnorm(nobs, xb, sd=1) # create y as adjusted random normal variate

# Fit
summary(mod <- lm(y ~ x1)) # model of the synthetic data.
=====
```

This will generate the following output on screen:

```
Call:
lm(formula = y ~ x1)
Residuals:
    Min      1Q      Median      3Q      Max 
-3.2599 -0.7708  -0.0026  0.7888  3.9575 
Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 1.9885    0.1379   14.42 <2e-16 ***
x1          2.8935    0.2381   12.15 <2e-16 *** 
Residual standard error: 1.068 on 248 degrees of freedom
Multiple R-squared:  0.3732,    Adjusted R-squared:  0.3707 
F-statistic: 147.7 on 1 and 248 DF,  p-value: < 2.2e-16
```

Note that the values we assigned to the synthetic model are close to those displayed in the model output. In order to visualize the regression line in Figure 3.2 we can type

```
ypred <- predict(mod, type="response")           # prediction from the model
plot(x1, y, pch=19, col="red")                  # plot scatter
lines(x1, ypred, col='grey40', lwd=2)            # plot regression line
segments(x1, fitted(mod), x1, y, lwd=1, col="gray70") # add the residuals
```

3.1.2 Fitting a Linear Regression in Python

The equivalent routine in Python can be written as

Code 3.2 Ordinary least squares regression in Python without formula.

```
=====
import numpy as np
import statsmodels.api as sm
from scipy.stats import uniform, norm

# Data
np.random.seed(1056)          # set seed to replicate example
nobs = 250                     # number of obs in model
```

```

x1 = uniform.rvs(size=nobs)           # random uniform variable
alpha = 2.0                           # intercept
beta = 3.0                            # slope
xb = alpha + beta * x1                # linear predictor
y = norm.rvs(loc=xb, scale=1.0, size=nobs) # create y as adjusted
                                         # random normal variate
# Fit
unity_vec = np.full((nobs,),1, np.float) # unity vector
X = np.column_stack((unity_vec, x1))     # build data matrix with intercept
results = sm.OLS(y, X).fit()

# Output
print(str(results.summary()))
=====
```

For the above, the summary shown on the screen reads

```

OLS Regression Results
=====
Dep. Variable:                  y      R-squared:         0.420
Model:                          OLS    Adj. R-squared:      0.419
Method: Least Squares          F-statistic:        3612.
Date: Sun, 24 Jan 2016          Prob (F-statistic):   0.00
Time: 17:56:36                 Log-Likelihood:     -7060.4
No. Observations:             5000      AIC:            1.412e+04
Df Residuals:                 4998      BIC:            1.414e+04
Df Model:                      1
Covariance Type:               nonrobust
=====
      coeff    std err        t      P>|t|      [95.0% Conf. Int.]
-----  

const    2.0213    0.028    71.603    0.000      1.966    2.077  

x1       2.9747    0.049    60.102    0.000      2.878    3.072
=====
```

The final intercept value is reported as `const` and the slope as `x1`. The code produces the mean (`coeff`), standard deviations (`std err`), t statistics (this can be understood as the statistical significance of the coefficient), and p -values (`P>|t|`), on the null hypothesis that the coefficient is zero, and the lower and upper bounds for the 95% confidence intervals along with other diagnostics (some of which will be discussed later on).

Notice that here we have used a different option from that presented in Code 2.3, which does not require writing the formula ($y \sim x$) explicitly. We have done this so the reader can be aware of different possibilities of implementation. In what follows we shall always use the formula version in order to facilitate the correspondence between the R and Python codes.

3.2 Basic Theory of Bayesian Modeling

Bayesian statistical analysis, and Bayesian modeling in particular, started out slowly, growing in adherents and software applications as computing power became ever more powerful. When personal computers had sufficient speed and memory to allow statisticians and other analysts to engage in meaningful Bayesian modeling, developers began to write software packages aimed at the Bayesian market. This movement began in earnest after the turn of the century. Prior to that, most analysts were forced to use sometimes very tedious analytical methods to determine the posterior distributions of model parameters. This is a generalization, of course, and statisticians worked to discover alternative algorithms to calculate posteriors more optimally.

Enhancements to Bayesian modeling also varied by discipline. Mathematically adept researchers in fields such as ecology, econometrics, and transportation were particularly interested in applying the Bayesian approach. The reason for this attitude rests on the belief that science tends to evolve on the basis of a Bayesian methodology. For example, the emphasis of incorporating outside information as it becomes available, or is recognized, into one's modeling efforts seems natural to the way in which astrophysicists actually work.

Currently there are a number of societies devoted to Bayesian methodology. Moreover, the major statistical associations typically have committees devoted to the promotion of Bayesian techniques. The number of new books on Bayesian methodology has been rapidly increasing each year. In astronomy, Bayesian methods are being employed ever more frequently for the analysis of astronomical data. It is therefore important to develop a text that describes a wide range of Bayesian models that can be of use to astronomical study. Given that many astronomers are starting to use Python, Stan, or JAGS for Bayesian modeling and R for basic statistical analysis, we describe Bayesian modeling from this perspective.

In this chapter we present a brief overview of this approach. As discussed in Chapter 1, Bayesian statistics is named after Thomas Bayes (1702–1761), a British Presbyterian minister and amateur mathematician who was interested in the notion of inverse probability, now referred to as posterior probability. The notion of posterior probability can perhaps best be understood with a simple example.

Suppose that 60% of the stellar systems in a galaxy far, far away host an Earth-like planet. It follows therefore that 40% of these stellar systems fail to have Earth-like planets. Moreover, let us also assume that every system that hosts an Earth-like planet also hosts a Jupiter-like planet, while only half the systems which fail to host Earth-like planets host Jupiter-like planets. Now, let us suppose that, as we peruse stellar systems in a Galaxy, we observe a particular system with a Jupiter-like planet. What is the probability that this system also hosts an Earth-like planet?

This is an example of inverse probability. Bayes developed a formula that can be used to determine the probability that the stellar system being examined here does indeed host an Earth-like planet. The equation used to solve this problem is called Bayes' theorem.

We can more easily calculate the probability if we list each element of the problem together with its probability. We symbolize the terms as follows:

\oplus = Earth-like planet

∇ = Jupiter-like planet

\sim = not, or not the case. This symbol is standard in symbolic logic.

The list of problem elements is therefore:

$P(\oplus) = 0.6$, the probability that the system hosts an Earth-like planet, notwithstanding (i.e., ignoring) other information;

$P(\sim \oplus) = 0.4$, the probability that a system does not have a Earth-like planet, notwithstanding other information;

$P(\text{J} \mid \sim \oplus) = 0.5$, the probability that a system hosts a Jupiter-like planet given that it does not host an Earth-like planet;

$P(\text{J} \mid \oplus) = 1$, the probability that a system hosts a Jupiter-like planet given that it also hosts an Earth-like planet;

$P(\text{J}) = 0.8$, the probability of randomly selecting a system that has a Jupiter-like planet.

The value $P(\text{J}) = 0.8$ follows from

$$P(\text{J}) = P(\text{J} + \sim \oplus) + P(\text{J} + \oplus) \quad (3.5)$$

$$= P(\text{J} \mid \sim \oplus)P(\sim \oplus) + P(\text{J} \mid \oplus)P(\oplus) = 0.5 \times 0.4 + 1 \times 0.6 = 0.8. \quad (3.6)$$

In terms of the above notation, the probability that a system contains both an Earth-like and a Jupiter-like planet is given by

$$P(\oplus + \text{J}) = P(\oplus \mid \text{J})P(\text{J}) = P(\text{J} \mid \oplus)P(\oplus). \quad (3.7)$$

We can rearrange this expression as

$$P(\oplus \mid \text{J}) = \frac{P(\text{J} \mid \oplus)P(\oplus)}{P(\text{J})} = \frac{1 \times 0.6}{0.8} = 0.75 \quad (3.8)$$

Thus, the posterior mean probability that a particular system hosts an Earth-like planet, given that it hosts a Jupiter-like planet, is approximately 75%. This statistic makes sense given the specified conditions. But it is not an outcome that can be obtained using traditional frequentist methods. Moreover, the solution is not easy to calculate without knowing Bayes' theorem. Formula 3.8 is an application of the generic expression of Bayes theorem, which is given formally as

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}, \quad (3.9)$$

where A and B are specified events. Suppose we have a model consisting of continuous variables, including the response y . When using Bayes' theorem for drawing inferences from data such as these, and after analyzing the data, one might be interested in the probability distribution of one or several parameters θ , i.e., the posterior distribution $p(\theta \mid y)$. To this end, the Bayesian theorem Equation 3.9 is reformulated for continuous parameters using probability distributions:

$$p(\theta|y) = \frac{p(y|\theta)p(\theta)}{p(y)} = \frac{L(\theta)\pi(\theta)}{\int L(\theta)\pi(\theta)d\theta}. \quad (3.10)$$

In Bayesian modeling, the second right-hand side of Equation 3.10 is formed by the model likelihood $L(\theta)$ times the prior distribution $\pi(\theta)$ divided by $\int L(\theta)\pi(\theta)d\theta$, which is the total probability of the data taking into account all possible parameter values. The denominator is also referred to as the marginal probability or the normalization constant guaranteeing that the observation probabilities sum to 1. Since the denominator is a constant, it is not normally used in determining the posterior distribution $p(\theta|y)$. The posterior distribution of a model parameter is key to understanding a Bayesian model. The Bayesian formula underlying Bayesian modeling is therefore

$$p(\theta|y) \propto L(\theta)\pi(\theta), \quad (3.11)$$

with $p(\theta|y)$ as the posterior probability of a model parameter, given the data. The mean of the posterior distribution in a Bayesian model is analogous to the coefficient of a maximum likelihood model. The standard deviation of the posterior is similar to the standard error of the coefficient and the usual 95% credible intervals are similar to the 95% confidence intervals in frequentist thinking. The similarities are, however, purely external. Their interpretations differ, as do the manner in which they are calculated.

In Bayesian modeling, the model parameters are assumed to be distributed according to some probability distribution. In frequentist statistics the parameters are considered to be constant: they are the values of the underlying probability distribution which describes the data being modeled. The goal in maximum likelihood modeling is to estimate the unknown parameter values as accurately as possible. The error in this estimation is reflected in a parameter's standard error. The parameters to be estimated are slopes informing us of the rate of change in the response variable given a one-unit change in the respective predictors. The confidence interval for an estimated parameter is obtained on the basis that the true parameter would be within the confidence interval if the experiment were repeated an infinite number of times.

In Bayesian modeling the likelihood of the predictor data is combined with outside information which the analyst knows from other sources. If this outside information (data not in the original model specification) is to be mixed with the likelihood distribution, it must be described by an appropriate probability distribution with specified parameter values, or hyperparameters. This is the prior distribution. Each model parameter is mixed with an appropriate prior distribution. The product of the likelihood distribution and prior distribution produces a posterior distribution for each parameter in the model. To be clear, this includes posteriors for each predictor in the model plus the intercept and any ancillary shape or scale parameters. For instance, in a normal Bayesian model, posterior distributions must be found for the intercept, for each predictor, and for the variance, σ^2 . If the analyst does not know of any outside information bearing on a model parameter then a *diffuse prior* is given, which maximizes the information regarding the likelihood or the data. Some statisticians call this type of prior non-informative, but all priors carry some information. We shall employ informative prior distributions in Bayesian models when we discuss real astronomical

applications toward the end of the book. Examples of diffuse priors will also be given. When we describe synthetic Bayesian models, in Chapters 4 to 8, we will nearly always use diffuse priors. This will allow the reader to see the structure of the code so that it can be used for many types of application. It also allows the reader to easily compare the input fiducial model and the output posteriors.

Typically the foremost value of interest in a Bayesian model is the posterior mean. The median, or even the mode, can also be used as the central tendency of the posterior distribution. The median is generally used when the posterior distribution is substantially skewed. Many well-fitted model predictor-parameters are normally distributed, or approximately so; consequently the mean is most often used as the standard statistic of central tendency. Standard deviations and credible intervals are based on the mean and shape of the posterior. The credible intervals are defined as the outer 0.025 (or some other value) quantiles of the posterior distribution of a parameter. When the distribution is highly skewed, the credible intervals are usually based on the *highest posterior density* (HPD) region. For 0.025 quantiles, we can say that there is a 95% probability that the credible interval contains the true posterior mean. This is the common sense interpretation of a credible interval, and is frequently confused with the meaning of the maximum likelihood confidence interval. However, this interpretation cannot be used with maximum likelihood models. A confidence interval is based on the hypothesis that if we repeat the modeling estimation a large number of times then the true coefficient of a predictor or parameter will be within the range of the interval on 95% of the time. Another way to understand the difference between confidence and credible intervals is to realize that for a confidence interval the interval itself is a random variable whereas for a credible interval the estimated posterior parameter is the random variable.

In the simplest case, an analyst can calculate the posterior distribution of a parameter as the product of the model likelihood and a prior providing specific parameter values that reflect the outside information being mixed with the likelihood distribution. Most texts on Bayesian analysis focus on this process. More often though, the calculations are too difficult to do analytically, so that a Markov chain Monte Carlo (MCMC) sampling technique must be used to calculate the posterior. For the models we discuss in this book, we use software that assumes that such a sampling occurs.

A MCMC sampling method was first developed by Los Alamos National Laboratories physicists Nicholas Metropolis and Stanislaw Ulam ([Metropolis and Ulam, 1949](#)). Their article on “The Monte Carlo method” in the *Journal of the American Statistical Association* initiated a methodology that the authors could not have anticipated. The method was further elaborated by [Metropolis et al. \(1953\)](#). It was further refined and formulated in what was to be known as the Metropolis–Hastings method by W. K. Hastings ([Hastings, 1970](#)). Gibbs sampling was later developed by Stuart and Donald Geman in 1984 in what is now one of the most cited papers in statistical and engineering literature ([Geman and Geman, 1984](#)); JAGS, the main software we use in this text, is an acronym for “Just another Gibbs sampler.” Further advances were made by [Gelfand et al. \(1990\)](#), [Gelfand and Smith \(1990\)](#), and [Tanner and Wong \(1987\)](#). The last authors made it possible to obtain posterior distributions for parameters and latent variables (unobserved variables) of complex models. Bayesian algorithms are continually being advanced and, as software becomes faster, more sophisticated algorithms are likely to be developed. Advances in sampling methods and software have been developed also within the astronomical community itself; these include importance-nested sampling ([Feroz and Hobson, 2008](#)) and an affine-invariant ensemble sampler for MCMC ([Foreman-Mackey et al., 2013](#)). Stan, which we use for many examples in this text, is one of the

latest apparent advances in Bayesian modeling; it had an initial release date of 2012.

A Markov chain, named after Russian mathematician Andrey Markov (1856–1922), steps through a series of random variables in which a given value is determined on the basis of the value of the previous element in the chain. Values previous to the last are ignored. A transition matrix is employed which regulates the range of values randomly selected during the search process. When applied to posterior distributions, MCMC runs through a large number of samples until it finally achieves the shape of the distribution of the parameter. When this occurs, we say that convergence has been achieved. Bayesian software normally come with tests of convergence as well as graphical outputs displaying the shape of the posterior distribution. We will show examples of this in the text. A simple example of how a transition matrix is constructed and how convergence is achieved can be found in [Hilbe \(2011\)](#). Finally, without going into details, in Appendix A we show a very simple application of a faster alternative to MCMC sampling known as the integrated nested Laplace approximation, which results in posteriors that are close to MCMC-based posteriors.

3.2.1 Example: Calculating a Beta Prior and Posterior Analytically

In order to better understand Bayesian priors and how they relate to the likelihood and posterior distributions we will present a simple example with artificially created data. However, since we will be dealing with only summary data, our example can be applied to a wide variety of models.

There are many different types of priors and ways to calculate them. How a particular prior affects a likelihood distribution largely depends on the distributional properties of the likelihood. Our example assumes that the data is described by a Bernoulli likelihood distribution. We construct a *beta prior* (see Equation 3.15), multiplying the likelihood and prior distributions. Although this example is aimed to show how to construct a simple prior, we shall also calculate the posterior distribution analytically. Throughout the text we have to forego analytic solutions and let the sampling algorithms of R's `MCMCpack`, `JAGS`, `pymc3`, and Stan determine the posterior distribution for us. But, until the turn of the present century, nearly all Bayesian modeling had to be done analytically.

When a prior is mixed with a likelihood function, the resulting distribution, called the posterior distribution, may have distributional properties considerably different from either of its component distributions. Such a prior is generally referred to as an informative prior. We construct priors to reflect outside information that bears on a parameter. If we have no outside information relevant to the likelihood, a prior may be fashioned that barely affects the likelihood. In this case the prior is typically referred to as non-informative, although this is a poor name for it, as mentioned earlier. At times a so-called non-informative prior may actually contribute significant information to the model. In any case, though, manufacturing a prior such that it does not change the likelihood parameters nevertheless provides relevant information about the data – it tells the statistician that we know of no outside information about the data.

When a prior is mixed with a likelihood parameter the idea is to shift the posterior mean, median, or mode to a more reasonable value than if the prior information were not taken into account. We shall focus on the distributional mean in this book, not forgetting that if the data is highly skewed then a median is likely to be preferred. The mode is also frequently used as the primary statistic of

interest for a given parameter. The mean, however, is used more than the other measures of central tendency, so it is assumed in the remainder of the book, unless specifically indicated to the contrary.

In the example that follows we break up the discussion into likelihood, prior, and posterior. It does not matter if we start with the prior instead. It matters only that the posterior is described afterwards with reference to its two components. Our example will be based on a Bernoulli likelihood and beta prior. This likelihood and prior are common when the data being evaluated is binary in nature.

Likelihood

Suppose that we set out to randomly observe 50 star clusters in a particular type of galaxy over the course of a year. We observe one cluster per observing session, determining whether the cluster has a certain characteristic X , e.g., is an open cluster. We record our observations, defining $y = 1$ if the cluster we observe has characteristic X and $y = 0$ if it does not. It is noted that the distribution of observations follows a Bernoulli PDF, with the Bernoulli parameter θ , representing the probability that a random observation of a cluster within the 50 star clusters results in the detection of a cluster with characteristic X . That is, in this study, θ is the probability of observing a cluster with characteristic X .

The Bernoulli PDF may be expressed as

$$f(y_i; \theta) = \theta^{y_i} (1 - \theta)^{1-y_i}. \quad (3.12)$$

In terms of the likelihood, the data may be described as

$$L(\theta; y_i) = \theta^{y_i} (1 - \theta)^{1-y_i} \quad (3.13)$$

where we attempt to estimate the parameter, θ , on the basis of the distribution of y_i .

When we are actually attempting to model data, the log-likelihood function is preferred to the likelihood. The mathematical properties of summing across observations, which the log-likelihood provides, are substantially more efficient than multiplying across observations, as specified when employing a likelihood function for estimation. In fact, using the likelihood rather than the log-likelihood in maximum likelihood estimation will nearly always result in convergence problems. But, for our purposes, working with the likelihood is best. We are simply demonstrating how the likelihood, describing the data, is mixed with external prior information to form a posterior distribution with specific parameter values. We shall use the mean of the posterior distribution when reporting parameter values, not forgetting that if the data are skewed then the median is probably to be preferred to the mean.

Given that we observe seven clusters with characteristic X and 43 clusters without it, we may specify the likelihood for the data we have observed as

$$L(\pi) = \pi^7(1 - \pi)^{43}, \quad (3.14)$$

where π is the symbol used in many texts to represent the predicted mean value of the model response term, y . For a Bernoulli distribution, which is the function used in binary logistic regression, π is the probability that $y = 1$, whereas $1 - \pi$ is the probability that $y = 0$. The value of π ranges from 0 to 1, whereas y has values of only 0 or 1. Note that the symbol μ is commonly used in place of π to represent the predicted mean. In fact, we shall be using μ throughout most of the text. However, we shall use π here for our example of a Bayesian Bernoulli and beta predicted probability. It may be helpful to get used to the fact that these symbols are used interchangeably in the literature.

In Bayesian modeling we assume that there is likely outside knowledge, independent of the model data, which can provide extra information about the subject of our model. Every parameter in the model – the predictor parameters, the intercept, and any ancillary dispersion, variance, or scale parameters – has a corresponding prior. If there is no outside information then we multiply the likelihood by a diffuse prior, which is also referred to as a reference prior, a flat prior, a uniform prior, or even a non-informative prior. The parameter estimates are then nearly identical to the maximum likelihood estimated parameters.

Prior

Let us return for a moment to the open or globular cluster example we mentioned before. According to previous studies we know that 22% to 26% of star clusters, in the particular galaxy type we are studying, have the characteristic X (are open clusters). We are using the observed data to specify our likelihood function, but wish to use the information from previous studies as an informative prior. That is, we wish to combine the information from our study with information from previous studies to arrive at a posterior distribution.

We start to construct a prior by determining its mean, which is 24%, or half of 22% and 26%. Given the rather narrow range in the previous observations we might consider the prior as heavily informative, but it does depend on the number of previous studies made, how similar the other galaxies are to our study, and our confidence in their accuracy. We do not have this information, though.

A reasonable prior for a Bernoulli likelihood parameter π is the beta distribution (Section 5.2.4). This is a family of continuous probability distribution functions defined on the interval $[0, 1]$ and hence ideal for modeling probabilities. The beta distribution may be expressed as

$$f(\pi; \alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \pi^{\alpha-1} (1 - \pi)^{\beta-1}. \quad (3.15)$$

The factor involving gamma functions is referred to as a normalization constant and is included to ensure that the product of observations equals 1. Notice that this factor does not include the

parameter π and, as such, it is typically excluded in the calculation of the posterior until the final steps, so that we write

$$p(\pi, \alpha, \beta) \propto \pi^{\alpha-1} (1-\pi)^{\beta-1}; \quad (3.16)$$

the symbol for proportionality, \propto , appears owing to the exclusion of the normalization constant. As already mentioned, the parameters of prior distributions are generally referred to as hyperparameters, in this case α and β .

Our posterior will therefore be determined by the product of a Bernoulli likelihood (Equation 3.14) and a beta prior over the parameter π (Equation 3.16):

$$P(\pi | \text{data}) = P(\pi | \alpha, \beta) \propto L(\pi) p(\pi; \alpha, \beta). \quad (3.17)$$

The influence of the prior in shaping the posterior distribution is highly dependent on the sample size n . If we have an overwhelmingly large data set at hand then the data itself, through the likelihood, will dictate the shape of the posterior distribution. This means that there is so much information in the current data that external or prior knowledge is statistically irrelevant. However, if the data set is small (in comparison with our prior knowledge and taking into account the complexity of the model), the prior will lead to the construction of the posterior. We shall illustrate this effect in our stellar cluster example.

As we mentioned above, we know from previous experience that, on average, 24% of the stellar clusters we are investigating are open clusters. Comparing Equations 3.13 and 3.16 we see that the same interpretation as assigned to the exponents of the Bernoulli distribution can be used for those of the beta distribution (i.e. the distributions are conjugate). In other words, $\alpha - 1$ can be thought of as the number of open clusters (i.e. where characteristic X was observed) and $\beta - 1$ as the number of globular clusters (i.e. without characteristic X). Considering our initial sample of $n = 50$ clusters, this means that

$$\alpha - 1 \text{ is given by } 0.24 \times 50 = 12$$

and

$$\beta - 1 \text{ is given by } 0.76 \times 50 = 38.$$

Thus, the parameters of our prior are $\alpha = 13$ and $\beta = 39$ and the prior itself is described as

$$p(\pi; \alpha, \beta) = \text{Beta}(\pi; 13, 39) \propto \pi^{13-1} (1-\pi)^{39-1} = \pi^{12} (1-\pi)^{38} \quad (3.18)$$

A plot of this prior, displayed in Figure 3.3, can be generated in R with the following code:

```

> x <- seq(0.005, 0.995, length=100)
> y <- dbeta(x, 13, 39)
> plot(x,y, type="l")
> abline(v=0.24)

```

We have placed a vertical line at the mean, 0.24, to emphasize that it coincides with the peak of the distribution.

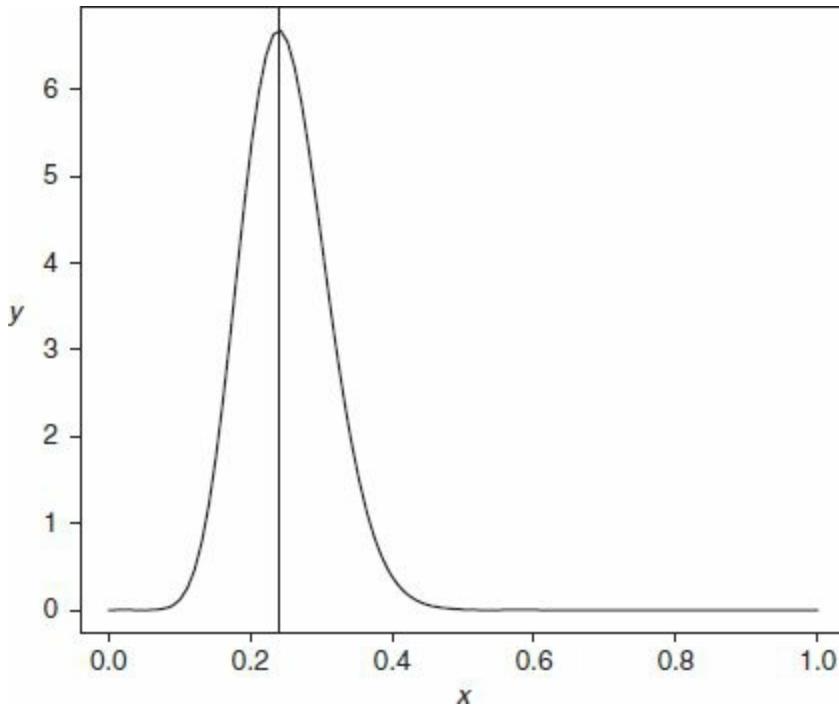


Figure 3.3 Beta prior p for $\alpha = 13$ and $\beta = 39$. The vertical line indicates the mean at $x = 0.24$.

Posterior

The posterior distribution is calculated as the product of the likelihood and the prior. Given the simple form of the Bernoulli and beta distributions, the calculation of the posterior is straightforward:

$$\begin{aligned}
 P(\pi | \alpha, \beta, n) &\propto \pi^y (1 - \pi)^{n-y} \pi^{\alpha-1} (1 - \pi)^{\beta-1} = \pi^{\alpha+y-1} (1 - \pi)^{\beta+n-y-1} \\
 &\propto \text{Beta}(\alpha + y, \beta + n - y).
 \end{aligned} \tag{3.19}$$

Using the parameter values calculated before,

$$P(\pi) \propto \text{Beta}(\pi; 13 + 7, 39 + 50 - 7) = \text{Beta}(\pi; 20, 82), \tag{3.20}$$

and the complete posterior distribution is

$$P(\pi) = \frac{\Gamma(102)}{\Gamma(20)\Gamma(82)}\pi^{20-1}(1-\pi)^{82-1}, \quad (3.21)$$

whose mean is given by

$$\pi_{\text{mean}} = \frac{\alpha}{\alpha + \beta} = \frac{20}{20 + 82} = 0.196. \quad (3.22)$$

Using a similar routine in R, we obtain the graphical representation shown in Figure 3.4:

```
> x <- seq(0.005, 0.995, length=100)
> y <- dbeta(x, 20, 82)
> plot(x, y, type="l")
> abline(v=.196)
```

Notice that the posterior distribution has shifted over to have a mean of 19.6%.

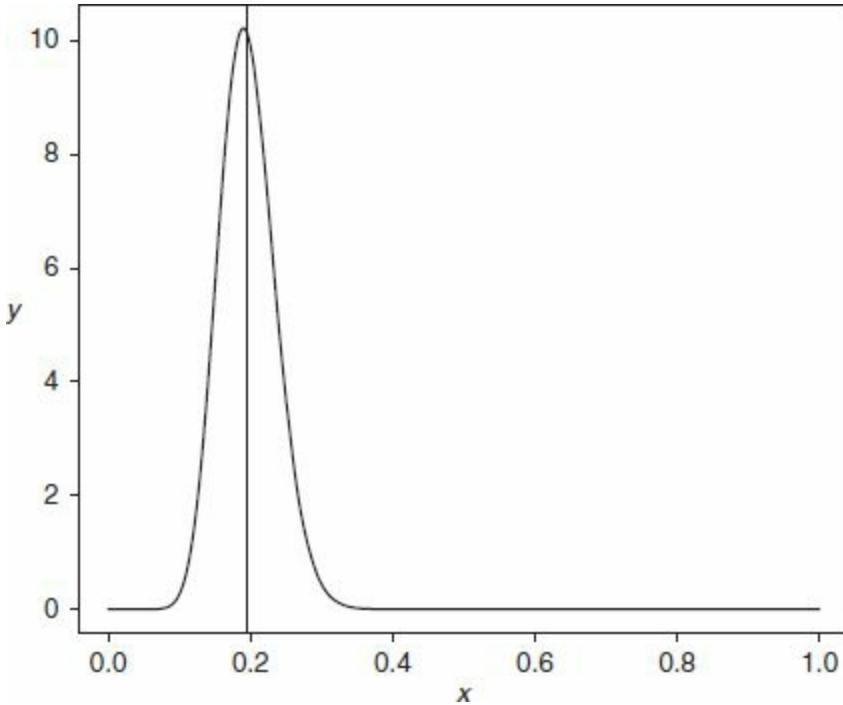


Figure 3.4 Beta posterior with $\alpha = 20$ and $\beta = 82$. The vertical line highlights the mean at $x = 0.196$.

The `triplot` function in the `LearnBayes` R package, which can be downloaded from CRAN, produces a plot of the likelihood, prior, and posterior, given the likelihood and prior parameters, as in Figure 3.5:

```
> library(LearnBayes)
> triplot(prior=c(13,39), data=c(7,43))
```

Note how the posterior (dashed line) summarizes the behavior of the likelihood (solid line) and the prior (dotted line) distributions.

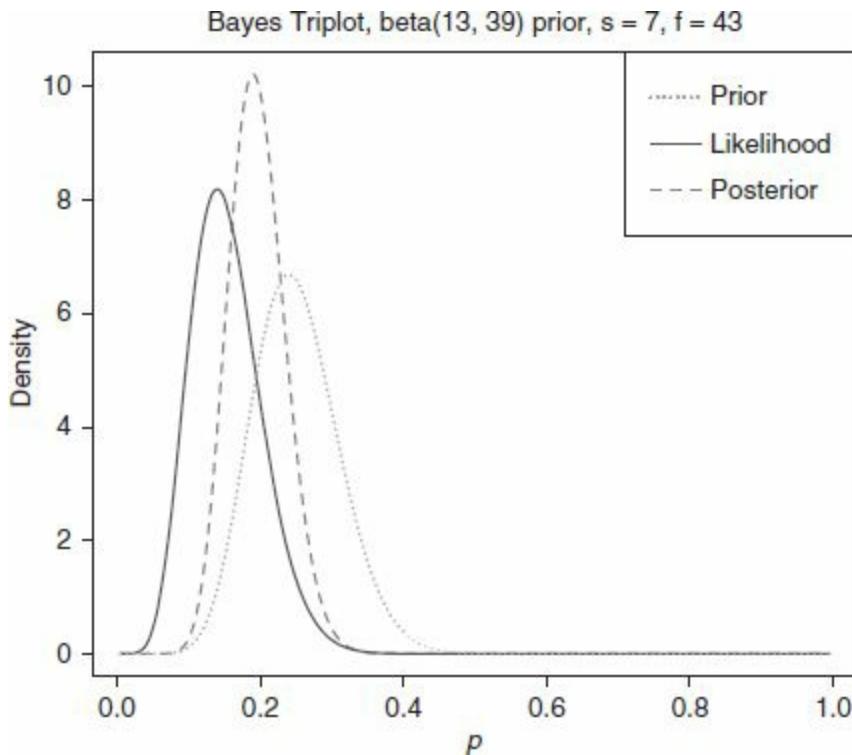


Figure 3.5 Beta prior distribution (dotted line), likelihood (solid line), and posterior distribution (dashed line) over π with sample size $n = 50$, calculated with the `tripplot` R function.

We can see from Figure 3.5 that the prior was influential and shifted the posterior mean to the right. The earlier plots of the prior and posterior are identical to the plots of the prior and posterior in this figure. The nice feature of the `tripplot` function is that the likelihood is also displayed.

Let us take a look at how the influence of the prior in the likelihood changes if we increase the number of data points drastically. Consider the same prior, a beta distribution with 0.24 mean, and a sample size of $n = 1000$ star clusters rather than 50, with the proportion between the classes as in our initial data set (7 out of 50, or 14% of the clusters in the sample are open clusters); then

number of open clusters is $0.14 \times 1000 = 140$

and

number of globular clusters is $0.86 \times 1000 = 860$.

Note that we still have 14% of the new observed star clusters as open clusters, but now with a

much larger data set to support this new evidence. The distributions shown in Figure 3.6 can be generated by typing

```
> triplot(prior=c(13, 39), data=c(140, 860))
```

in the R terminal.

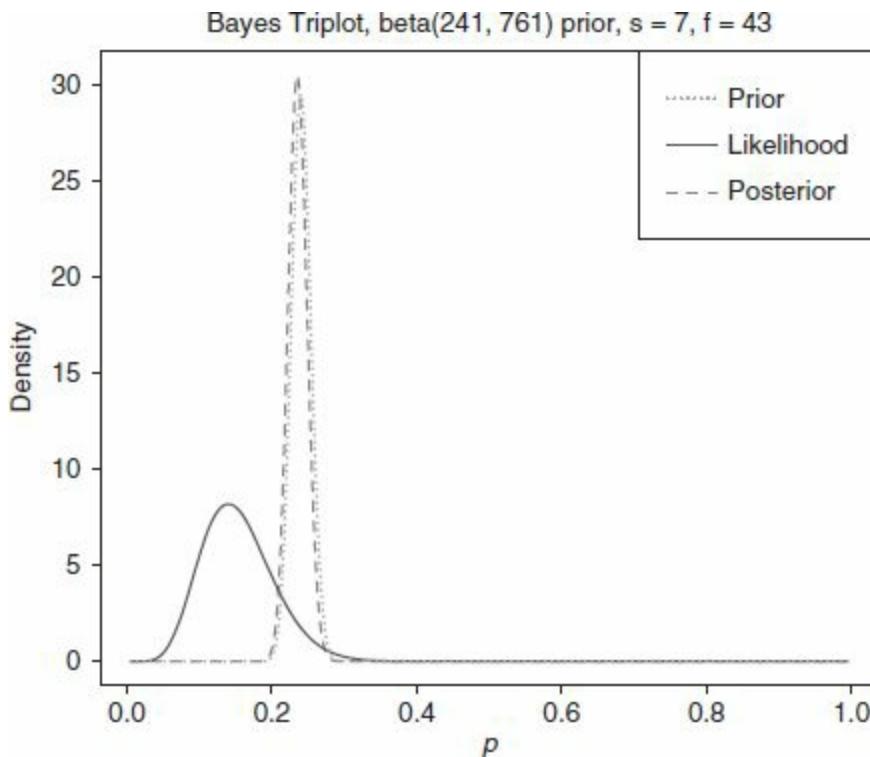


Figure 3.6 Beta prior distribution (dotted line), likelihood (solid line), and posterior distribution (dashed line) over π , with sample size $n = 1000$

It is clear that the likelihood and posterior are nearly the same. Given a large sample size, the emphasis is on the data. We are telling the model that the prior is not very influential given the overwhelming new evidence.

Finally we employ a sample size of 10, which is five times smaller than our initial data set, in order to quantify the influence of the prior in this data situation where

number of open clusters is $0.14 \times 10 = 1.4$

and

number of globular clusters is $0.86 \times 10 = 8.6$

A figure of all three distributions with $n = 10$ is displayed, as in Figure 3.7, by typing

```
> triplot(prior=c(13,39), data=c(1.4,8.6))
```

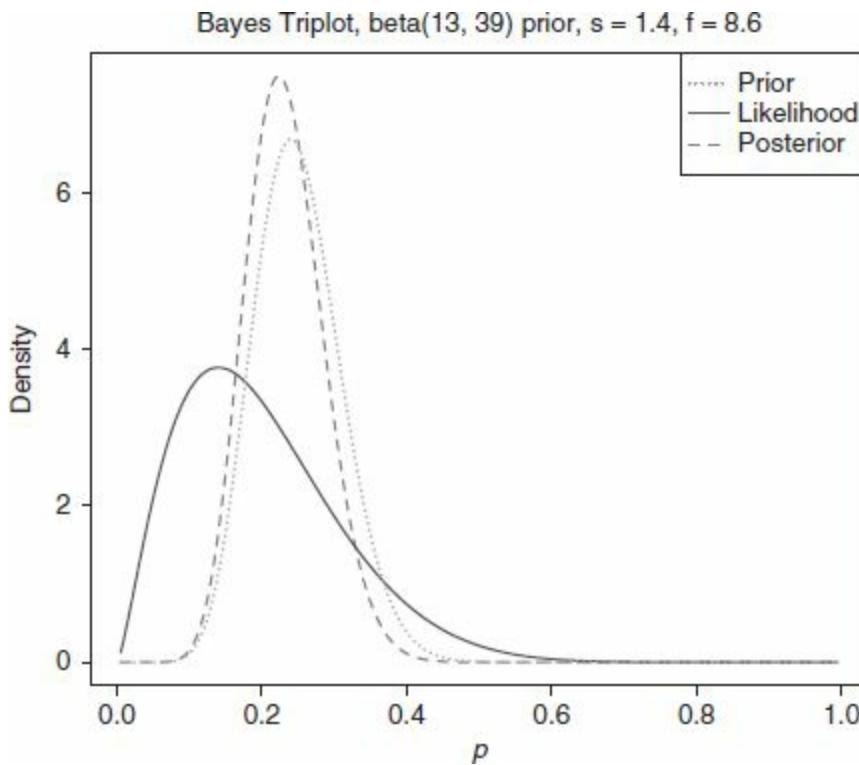


Figure 3.7 Beta prior distribution (dotted line), likelihood (solid line), and posterior distribution (dashed line) over π with sample size $n = 10$

Note that now the prior is highly influential, resulting in the near identity of the prior and posterior. Given a low sample size, the outside information coming to bear on the model is much more important than the study data itself.

The above example is but one of many that can be given. We have walked through this example to demonstrate the importance of the prior relative to new information present in the data set. Other commonly used priors are the normal, gamma, inverse gamma, Cauchy, half-Cauchy, and χ^2 . Of course there are a number of diffuse priors as well – the uniform or flat prior, and standard priors with hyperparameters that wash out meaningful information across the range of the likelihood – the so-called non-informative priors. These priors will be discussed in the context of their use throughout the text.

3.2.2 Fitting a Simple Bayesian Normal Model using R

When applying Bayesian estimation techniques we have the option of either writing the MCMC code ourselves or using an R function such as `mcmcregress` from the `MCMCpack` package. The code below demonstrates how this can be done using the latter option.

Code 3.3 Bayesian normal linear model in R.

```
=====
library(MCMCpack)

# Data
nobs = 5000           # number of obs in model
x1 <- runif(nobs)     # random uniform variable

beta0 = 2.0            # intercept
beta1 = 3.0            # angular coefficient

xb <- beta0 + beta1 * x1      # linear predictor
y <- rnorm(nobs, xb, sd=1)

# Fit
posteriors <- MCMCregress(y ~ x1, thin=1, seed=1056, burnin=1000,
                           mcmc=10000, verbose=1)

# Output
summary(posteriors)
=====
```

The above code will generate the following diagnostic information on screen:

```
Iterations = 1001:11000
Thinning interval = 1
Number of chains = 1
Sample size per chain = 10000

1. Empirical mean and standard deviation for each variable,
   plus standard error (SE) of the mean:
      Mean        SD    Naive SE  Time-series SE
(Intercept) 1.985  0.03740  0.0003740  0.0003740
x1          3.032  0.04972  0.0004972  0.0004972
sigma2       1.026  0.02028  0.0002028  0.0002060

2. Quantiles for each variable:
   2.5%    25%    50%    75%   97.5%
(Intercept) 1.9110  1.960  1.985  2.009  2.058
x1          2.9341  2.998  3.032  3.066  3.131
sigma2       0.9874  1.012  1.026  1.040  1.067
```

To visualize the posteriors (Figure 3.8) one can just type

```
> plot(posteriors)
```

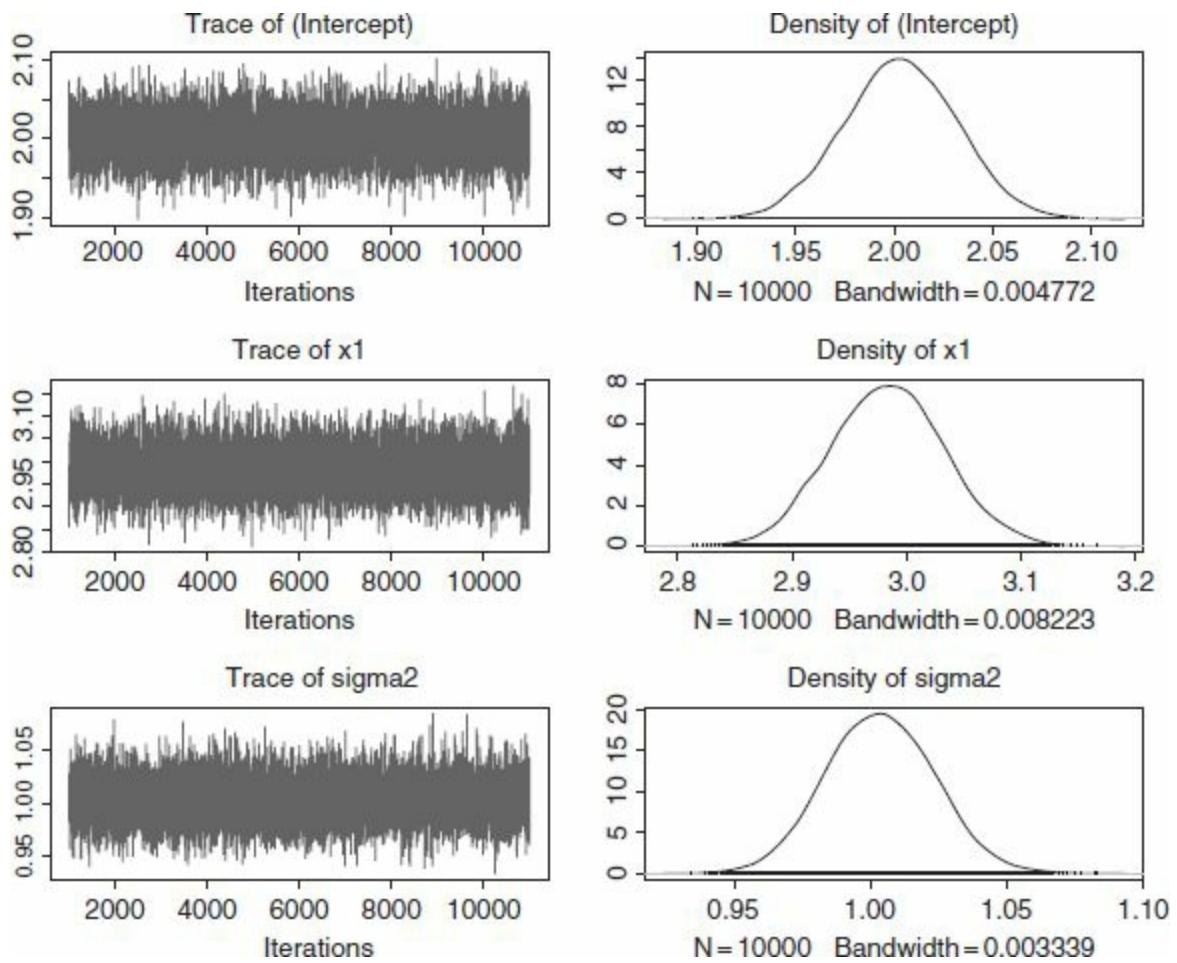


Figure 3.8 Default posterior diagnostic plots from MCMCregress.

3.2.3 Fitting a Simple Bayesian Normal Model using Python

In order to demonstrate how a linear regression can be done in Python using `pymc3` we will use the toy data generated in Section 3.1.2. Thus, in the snippet below we assume the same `Data` code block as in Code 3.2.

Code 3.4 Bayesian normal linear model in Python.

```
=====
from pymc3 import Model, sample, summary, traceplot
from pymc3.glm import glm
import pylab as plt
import pandas

# Fit
df = pandas.DataFrame({'x1': x1, 'y': y}) # rewrite data

with Model() as model_glm:
    glm('y ~ x1', df)
    trace = sample(5000)

# Output
summary(trace)

# Show graphical output
traceplot(trace)
plt.show()
```

```
=====
```

This code will return a summary like this:

```
Intercept:  
  Mean      SD     MC Error    95% HPD interval  
-----  
 1.985     0.052    0.002    [1.934, 2.045]  
Posterior quantiles:  
  2.5       25       50       75       97.5  
|-----|=====|=====|-----|  
1.930     1.967    1.986    2.006    2.043  
  
x1:  
  Mean      SD     MC Error    95% HPD interval  
-----  
 2.995     0.079    0.002    [2.897, 3.092]  
Posterior quantiles:  
  2.5       25       50       75       97.5  
|-----|=====|=====|-----|  
2.897     2.965    2.997    3.031    3.092  
  
sd_log:  
  Mean      SD     MC Error    95% HPD interval  
-----  
-0.009     0.013    0.001    [-0.028, 0.012]  
Posterior quantiles:  
  2.5       25       50       75       97.5  
|-----|=====|=====|-----|  
-0.028    -0.016   -0.009   -0.002    0.012  
  
sd:  
  Mean      SD     MC Error    95% HPD interval  
-----  
 0.992     0.013    0.001    [0.972, 1.012]  
Posterior quantiles:  
  2.5       25       50       75       97.5  
|-----|=====|=====|-----|  
0.972     0.984    0.991    0.998    1.012  
=====
```

The plots for the chains (MCMC sampling sessions) and the posterior plots (the Python equivalent to Figure 3.8) are shown in Figure 3.9.

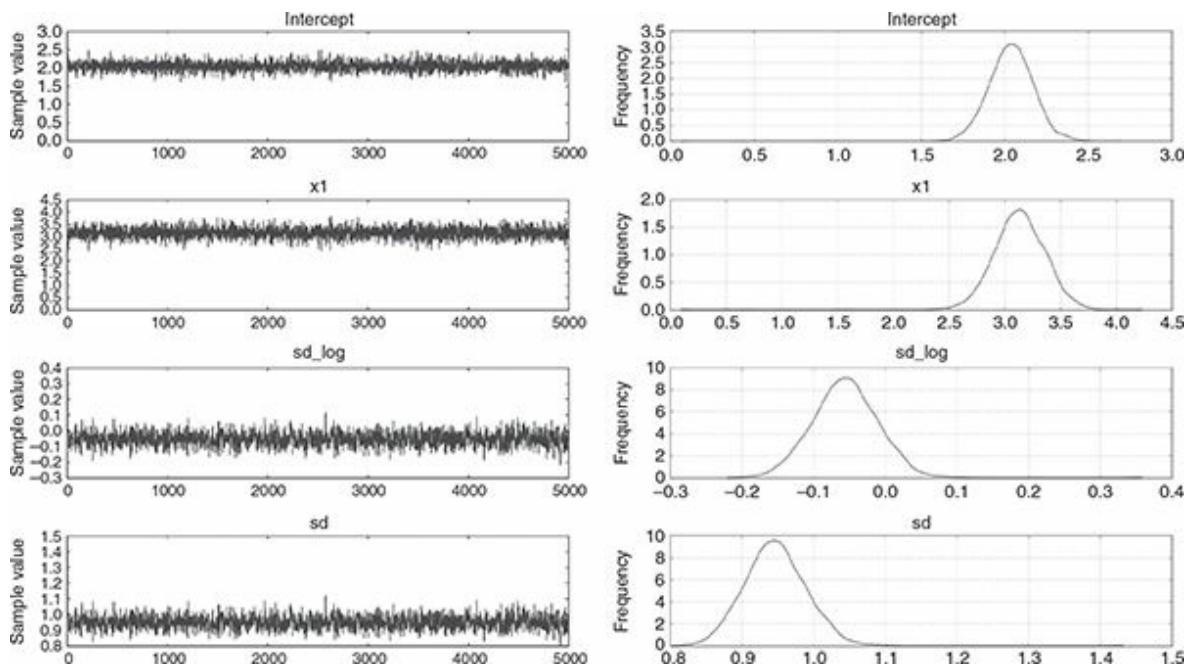


Figure 3.9 Output plots for the chains (left) and posterior (right) from Code 3.4.

3.3 Selecting Between Frequentist and Bayesian Modeling

A key reason to prefer a Bayesian model over one from the frequentist tradition is to bring information into the model that comes from outside the specific model data. That is, these models are based on data. Maximum likelihood methods attempt to find the best estimates of the true parameter values of the probability distribution that generates the data being modeled. The data are given; however, most modeling situations, in particular scientific modeling, are not based on data that is unconnected to previous study information or to the prior beliefs of those modeling the data. When modeling astrophysical data we want to account for any information we may have about the data that is relevant to understanding the parameter values being determined. The difficult aspect of doing this is that such prior information needs to be converted to a probability function with parameter values appropriate for the information to be included in the model. When we mix the likelihood with a prior distribution, we are effectively changing the magnitude and/or direction of movement of the likelihood parameters. The result is a posterior distribution. In Chapter 10 we will show a number of examples of how informative priors affect posterior distributions when we discuss applications of Bayesian models to real astronomical data.

In some instances converting prior information to a probability function is rather simple, but it is typically challenging. For these latter types of situation most statisticians subject the data to an MCMC-type sampling algorithm that converges to the mean (or median) of the parameters being estimated. On the basis of the mean of the distribution and the shape of the parameter, standard deviation and credible interval values may be calculated. In Bayesian modeling separate parameter distributions are calculated for each explanatory predictor in the model as well as for the intercept and for each scale or ancillary parameter specified for the model in question. For a normal model, separate parameters are solved for each predictor, the intercept, and the variance parameter, usually referred to as sigma-squared. The Bayesian logistic model provides parameters for each predictor

and the intercept. It has no scale parameter. Likewise, a Bayesian Poisson model provides parameter means for predictors and the intercept. The negative binomial model, however, also has a dispersion parameter which must be estimated. The dispersion parameter is not strictly speaking a scale parameter but, rather, is an ancillary parameter which adjusts for Poisson overdispersion. We shall describe these relationships when addressing count models later in the text.

A common criticism of Bayesian modeling relates to the fact that the prior information brought into a model is basically subjective. It is for those modeling the data to decide which prior information to bring into the model, and how that data is cast into a probability function with specific parameter values. This criticism has been leveled against Bayesian methodology from the first time it was used as a model. We believe that the criticism is not truly valid unless the prior information is not relevant to the modeling question. However, it is vital for those defining prior information to present it clearly in their study report or journal article. After all, when frequentist statisticians model data, it is their decision on which predictors to include in the model and, more importantly, which to exclude. They also transform predictor variables by logging, taking the square, inverting, and so forth. Frequentists decide whether to factor continuous predictors into various levels or smooth them using a cubic spline, or a lesser smoother, etc. Thus decisions are made by those modeling a given data situation on the basis of their experience as well as the goals of the study. Moreover, when appropriate, new information can be brought into a maximum likelihood model by adding a new variable related to the new information.

However, in Bayesian modeling the prior information is specific to each parameter, providing a much more powerful way of modeling. Moreover, Bayesian modeling is much more in tune with how scientific studies are carried out and with how science advances. Remember that, when selecting so called non-informative or diffuse priors on parameters in a Bayesian model, the modeling results are usually close to the parameter values produced when using maximum likelihood estimation. The model is then based on the likelihood, or log-likelihood, and not on outside prior information. There is no *a priori* reason why almost all statistical models should not be based on Bayesian methods. If a researcher has no outside information to bear on a model that he or she is developing, a `normal(0, 0.00001)` or equivalent prior may be used to wash out any prior information² and only the model data is used in the sampling. Of course this involves a new way of looking at statistical modeling, but it may become common practice as computing power becomes ever faster and sampling algorithms more efficient.

As we shall discuss later in the book, how out-of-sample predictions are made using a Bayesian model differs considerably from how such predictions are made using maximum likelihood modeling. We will demonstrate examples of Bayesian prediction in the context of modeling real data in Chapter 10.

Finally, it is generally well known that most researchers using maximum likelihood estimation interpret confidence intervals to mean that there is a 95% probability (or whatever probability level is desired) that the true parameter value is located within the range of the interval. However, this is not how maximum likelihood confidence intervals are to be understood, as discussed earlier in this chapter. If a researcher wishes to be able to interpret a model's confidence interval in that manner, then he or she should use a Bayesian model. Doing so allows the researcher to obtain a credible interval for each parameter which can be interpreted in such a manner. However, if the priors used

in a Bayesian model are as diffuse or as non-informative as possible then the credible intervals will be nearly identical to the confidence intervals produced by employing maximum likelihood estimation on the same data. Of course the log-likelihood or likelihood functions used for describing the data must be the same. An observation might be made that, since confidence intervals are very close in value to the credible intervals for a “non-informative” Bayesian model, the Bayesian interpretation of credible intervals can be applied to the same frequency-based maximum likelihood model. But, although the values of the intervals may be the same, their respective derivations, and therefore justifications, differ.

Further Reading

- Cowles, M. K. (2013). *Applied Bayesian Statistics: With R and Open BUGS Examples*. Springer Texts in Statistics. Springer.
- Feigelson, E. D. and G. J. Babu (2012a). *Modern Statistical Methods for Astronomy: With R Applications*. Cambridge University Press.
- Hilbe, J. M. and A. P. Robinson (2013). *Methods of Statistical Model Estimation*. EBL-Schweitzer. CRC Press.
- Korner-Nievergelt, F., T. Roth, S. von Felten, J. Guélat, B. Almasi, and P. Korner-Nievergelt (2015). *Bayesian Data Analysis in Ecology Using Linear Models with R, BUGS, and Stan*. Elsevier Science.
- McElreath, R. (2016). *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*. Chapman & Hall/CRC Texts in Statistical Science. CRC Press.
-

¹ At this point we assume there are no uncertainties associated with the observations.

² Although this is common practice, and will be used in many instances in this volume, it might lead to numerical problems. Frequently, using a weakly informative prior instead is advisable (see the recommendations from the Stan community in [Stan, 2016](#)).

4 Normal Linear Models

4.1 The Gaussian or Normal Model

The first Bayesian model we address is the normal or Gaussian model. We shall examine the normal probability distribution function (PDF), as well as its likelihood and log-likelihood functions. All the Bayesian models we discuss in this volume are based on specific likelihoods, which in turn are simple re-parameterizations of an underlying PDF. For most readers, this section will be no more than review material since it is the most commonly used model employed by analysts and researchers when modeling physical and astrophysical data.

As discussed in Chapter 3, a PDF links the outcome of an experiment described by a random variable with its probability of occurrence. The normal or Gaussian PDF may be expressed as

$$f(y; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(y-\mu)^2/2\sigma^2}, \quad (4.1)$$

where y represents the experimental measurements of the random variable y , μ is its mean, and σ^2 is the scale parameter, or variance. The square root of the variance, σ , is called the standard deviation. The Gaussian PDF has the well-known format of a bell curve, with the mean parameter determining the location of its maximum in the real number domain \mathbb{R} and the standard deviation controlling its shape (Figure 4.1 illustrates how these parameters affect the shape of the distribution). Notice the function arguments on the left-hand side of Equation 4.1. The probability function, indicated as $f(\cdot)$, tells us that the data y is generated on the basis of the parameters μ and σ . Any symbol on the right-hand side of Equation 4.1 that is not y or a parameter is regarded as a constant. This will be the case for all the PDFs we discuss.

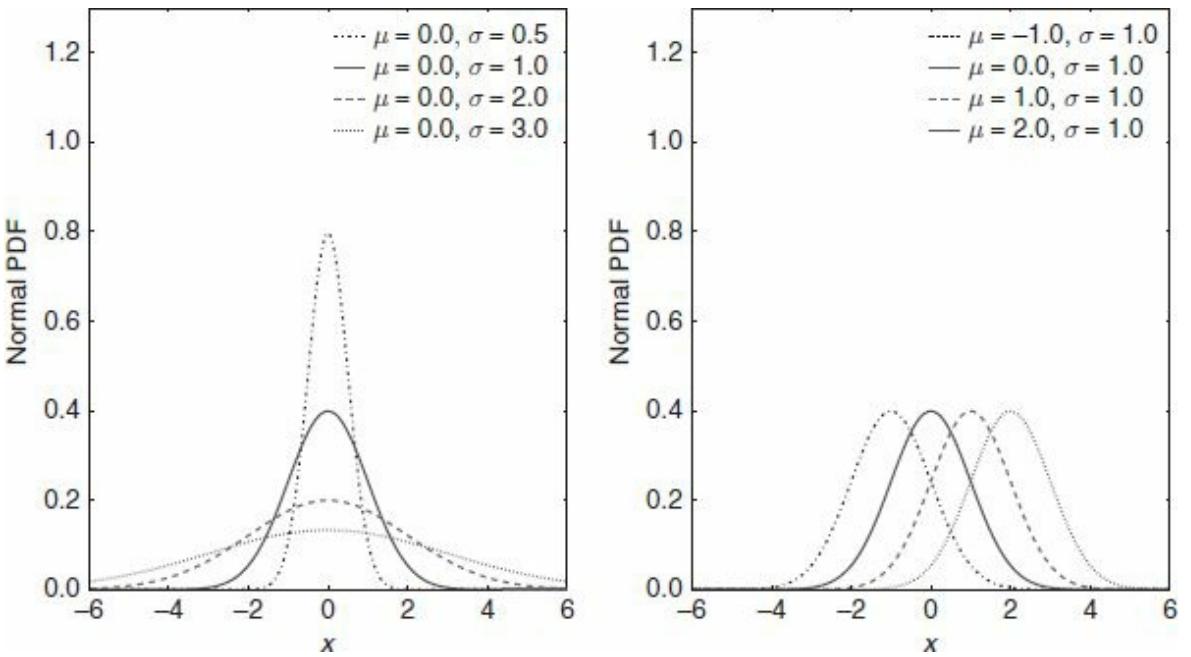


Figure 4.1 Left: Set of gaussian or normal probability distribution functions centered at the origin, $\mu = 0$, with different values for the standard deviation σ . Right: Set of gaussian or normal probability distribution functions centered in different locations with standard deviation $\sigma = 1.0$

A statistical model, whether it is based on a frequentist or Bayesian methodology, is structured to determine the values of parameters based on the given data. This is just the reverse of how a PDF is understood. The function that reverses the relationship of what is to be generated or understood in a PDF is called a **likelihood** function. The likelihood function determines which parameter values make the data being modeled most likely – hence the name likelihood. It is similar to the PDF except that the left-hand side of the equation now appears as

$$L(\mu, \sigma^2; y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(y-\mu)^2/2\sigma^2}, \quad (4.2)$$

indicating that the mean and variance parameters are to be determined on the basis of the data.

Again, in statistical modeling we are interested in determining estimated parameter values. It should also be mentioned that both the probability and likelihood functions apply to all the observations in a model. The likelihood of a model is the product of the likelihood values for each observation. Many texts indicate this relationship with a product sign, \prod , placed in front of the term on the right-hand side of Equation 4.1. Multiplying individual likelihood values across observations can easily lead to overflow, however. As a result, statisticians log the likelihood and use this as the basis of model estimation. The log-likelihood is determined from the sum of individual observation log-likelihood values. Statisticians are not interested in individual log-likelihood values, but the estimation algorithm uses them to determine the model's overall log-likelihood. The log-likelihood function \mathcal{L} for the normal model may be expressed as follows:

$$\mathcal{L}(\mu, \sigma^2; y) = \sum_{i=1}^N \left[-\frac{(y_i - \mu_i)^2}{2\sigma^2} - \frac{1}{2} \ln(2\pi\sigma^2) \right]. \quad (4.3)$$

The normal model is used to estimate the mean and variance parameters from continuous data with a range of $-\infty$ to $+\infty$. A key distributional assumption of the normal or Gaussian distribution is that the variance is constant for all values of the mean. This is an important assumption that is often neglected when models are selected for specific data situations. Also note that the normal model allows for the possibility that the continuous variable being modeled has both negative and positive values. If the data being modeled can only have positive values, or if the values are discrete or are counts, using a normal model violates the distributional assumptions upon which the model is based.

The Gaussian, or normal, model is defined by the probability and log-likelihood distributions underlying the data to be modeled, specifically the behavior of the noise associated with the response variable. In the case of a normal linear model with a single predictor, the response variable *noise* is normally distributed and the relationship between the response variable y and the explanatory variable x traditionally takes the form

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon, \quad \varepsilon \sim \text{Normal}(0, \sigma^2). \quad (4.4)$$

Figure 4.2 illustrates this behavior, showing a systematic component (the linear relationship between x and y) and the normal PDF driving the stochastic element of the model (the errors in y). In Equation 4.4, the intercept and slope of the explanatory variable x are represented by β_0 and β_1 , respectively, and ε is the error in predicting the response variable y . The index i runs through all the available observations. We can drop the error term by indicating \hat{y} as an estimated fitted value with its own standard error:

$$\hat{y}_i = \beta_0 + \beta_1 x_i. \quad (4.5)$$

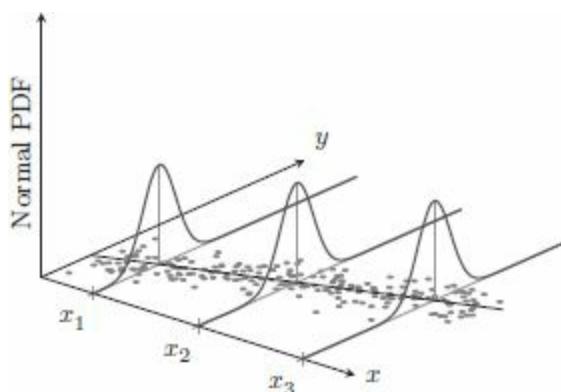


Figure 4.2 Illustration of points normally distributed around a linear relation $y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$, with $\varepsilon \sim \text{Normal}(0, \sigma^2)$

In the case of the normal linear model, \hat{y} is the fitted value and is also symbolized by μ or, more technically, $\hat{\mu}$. Statisticians use the hat over a symbol to represent an estimate (be aware that this notation is not used when the true value of the parameter is known). However, we will not employ this convention. It will be straightforward to use the context in which variables are being described in order to distinguish between these two meanings.

For the normal linear model the two values are identical, $\hat{y} = \mu$. This is not the case for models not based on the normal or Gaussian distribution, e.g., logistic, Poisson, or gamma regression models.

4.1.1 Bayesian Synthetic Normal Model in R using JAGS

Below we provide fully working JAGS code for estimating the posterior distributions of a synthetic normal model. Note that the format follows the schemata provided in Section 2.3 for a JAGS model. We do not call this a regression model, although there are authors who do. We regard a regression as a routine that is based on an ordinary least squares (OLS), maximum likelihood, iteratively re-weighted least squares (a subset of maximum likelihood), or quadrature algorithm. These are frequentist methods in which coefficients or slopes are estimated for each model predictor and parameter.

In Bayesian modeling we attempt to develop a posterior distribution for each parameter in the model. The parameters consist of the intercept, each predictor, and the scale or other heterogeneity parameters, e.g., the variance for normal models. A posterior distribution is developed for each parameter by multiplying the model log-likelihood by a prior distribution, which may differ for each parameter. A mean, standard deviation, and credible interval for the posterior distribution are calculated on the basis of some variety of MCMC sampling method. The mean of the posterior distribution of a predictor parameter is analogical to a regression coefficient, the standard deviation is analogical to the regression standard error, and the credible interval is analogical to a confidence interval. These three Bayesian statistics are not interpreted in the same manner as their regression analogs. We discussed these relationships in the last chapter, but it is important to keep them in mind when engaged in Bayesian modeling.

Code for a JAGS synthetic normal model is displayed below. Following the code and statistical output we present a walk-through guide to the algorithm.

Code 4.1 Normal linear model in R using JAGS.

```
=====
require(R2jags)

set.seed(1056)          # set seed to replicate example
nobs = 5000              # number of observations in model
x1 <- runif(nobs)        # random uniform variable

beta0 = 2.0               # intercept
beta1 = 3.0               # slope or coefficient
xb <- beta0 + beta1 * x1 # linear predictor, xb
y <- rnorm(nobs, xb, sd=1) # create y as adjusted random normal variate

# Construct data dictionary
X <- model.matrix(~ 1 + x1)
```

```

K <- ncol(X)
model.data <- list(Y = y,      # response variable
                     X = X,      # predictors
                     K = K,       # number of predictors including the intercept
                     N = nobs     # sample size
                     )
# Model set up
NORM <- "model{
  # Diffuse normal priors for predictors
  for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001) }

  # Uniform prior for standard deviation
  tau <- pow(sigma, -2)          # precision
  sigma ~ dunif(0, 100)          # standard deviation

  # Likelihood function
  for (i in 1:N){
    Y[i]~dnorm(mu[i],tau)
    mu[i] <- eta[i]
    eta[i] <- inprod(beta[], X[i,])
  }
}

# Initial values
inits <- function () {
  list(beta = rnorm(K, 0, 0.01))
}

# Parameters to be displayed
params <- c("beta", "sigma")

# MCMC
normfit <- jags(data = model.data,
                  inits = inits,
                  parameters = params,
                  model = textConnection(NORM),
                  n.chains = 3,
                  n.iter = 15000,
                  n.thin = 1,
                  n.burnin = 10000)

print(normfit, intervals = c(0.025, 0.975), digits = 2)
=====
mu.vect sd.vect   2.5%   97.5% Rhat n.eff
beta[1]     1.99    0.03    1.94    2.05    1 15000
beta[2]     3.01    0.05    2.91    3.10    1 15000
sigma      1.00    0.01    0.98    1.02    1  8300
deviance  14175.68    2.45 14172.91 14182.03    1 15000
pD = 3. - and DIC = 14173.7

```

The source code below, CH_Figures-R, comes from a section of code that is made available to readers of [Zuur, Hilbe, and Ieno \(2013\)](#) from the publisher's website.

Plot the chains to assess mixing (Figure 4.3):

```

source("CH-Figures.R")
out <- normfit$BUGSoutput
MyBUGSChains(out,c(uNames("beta",K),"sigma"))

```

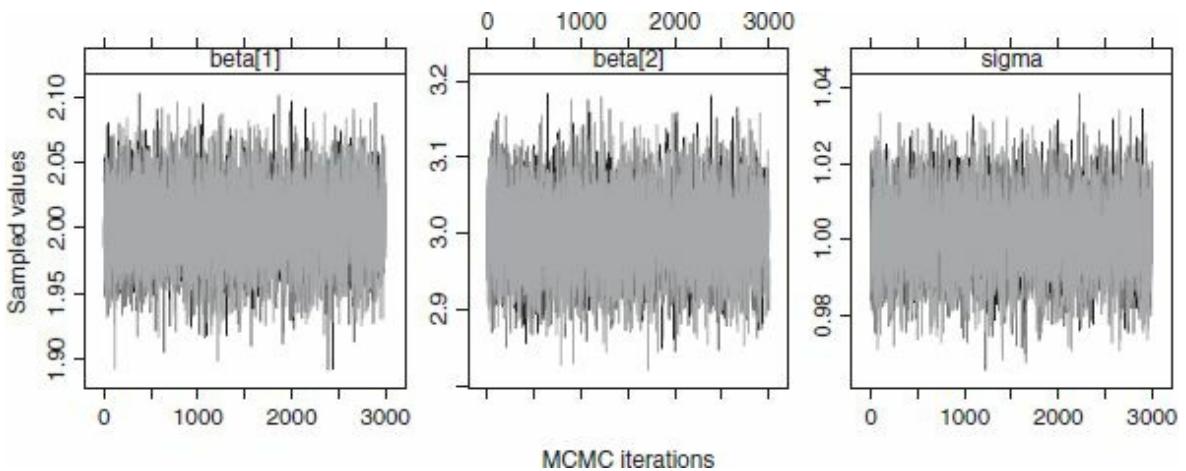


Figure 4.3 MCMC chains for the two regression parameters β_1 and β_2 and for the standard deviation σ for the normal model.

Display the histograms of the model parameters (Figure 4.4):

```
source("CH-Figures.R")
out <- normfit$BUGSoutput
MyBUGSHist(out,c(uNames("beta"),K),"sigma"))
```

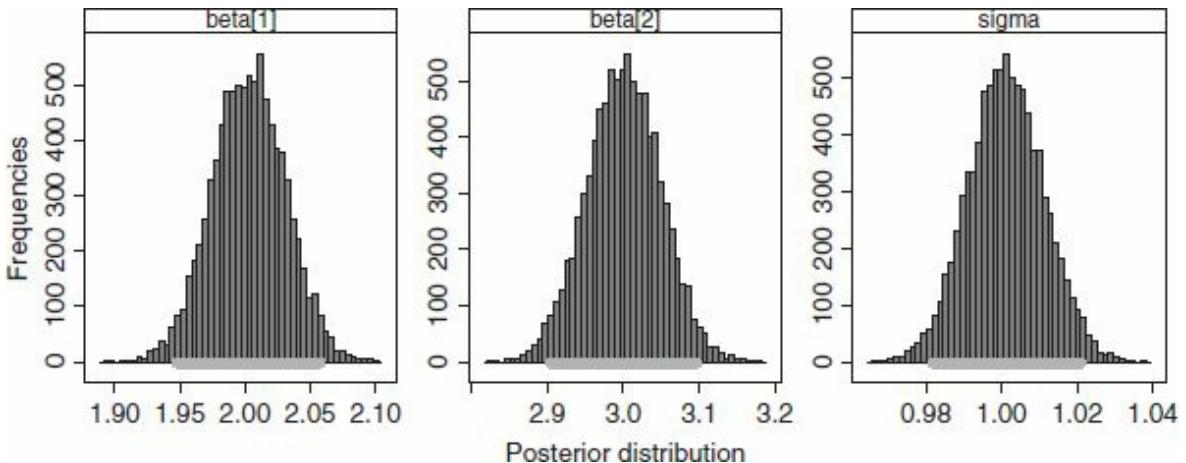


Figure 4.4 Histograms of the MCMC iterations for each parameter. The thick line at the base of each histogram represents the 95% credible interval. Note that no 95% credible interval contains 0.

The algorithm begins by making certain that the `r2jags` library is loaded into the memory. If it is not then an error message will appear when one attempts to run the model. We set the seed at 1056; in fact we could have set it at almost any value. The idea is that the same results will be obtained if we remodel the data using the same seed. We also set the number of observations in the synthetic model at 5000. A single predictor consisting of random uniform values is defined for the model and given the name `x1`. The following line specifies the linear predictor, `xb`, which is the sum of the intercept and each term in the model. A term is understood as the predictor value times its associated coefficient (or estimated mean). We have only a single term here. Finally, we use R's (pseudo)random normal number generator, `rnorm()`, with three arguments to calculate a random

response y . We specify that the synthetic model has intercept value 2 and slope or coefficient 3. When we run a linear OLS model of y on x_1 , we expect the values of the intercept and coefficient to be close to 2 and 3 respectively. We defined sigma to be 1, but sigma is not estimated using R's `g1m` function. It is estimated using R's `lm` function as the “residual mean squared error.” It can also be estimated using our JAGS function for a normal model. The value estimated is 0.99 ± 0.1 .

The Bayesian code begins by constructing a directory or listing of the components needed for developing a Bayesian model. Recall that the code will provide posterior distributions for the intercept, for the x_1 slope, and for the variance.

The term x consists of a matrix of 5000 observations and two rows, one for the intercept, 1, and another for x_1 ; `model.data` is a list containing y , the response, x , the data matrix, κ , the number of columns in the matrix, and n , the number of observations in the matrix. We will have other models where we define b_0 and σ^2 in the list as the mean and variance of the priors used in the model. If we are using so-called non-informative priors then they are not needed but, if we do want to use them here, we could specify `b0 = rep(0, κ)` meaning that there are κ priors each with value 0. The variance σ^2 can be defined as `diag(0.00001, κ)`, meaning that each prior has a very wide variance (as the inverse of the precision) and thus does not contribute additional information to the model. Again, our model here is simple and does not require knowing or adjusting the mean and variance of the priors.

The next grouping consists of a block where the priors and log-likelihood are defined. For the intercept and x_1 we assign what is called a non-informative prior. We do this when we have no outside information about x_1 , which is certainly the case here given its synthetic nature. Bayesian statisticians have demonstrated that no prior is truly non-informative if run on real data. We learn something about the data by specifying that we have no outside information about it.

Recall that outside information (the prior) must be converted to a probability distribution and multiplied with the log-likelihood distribution of the model. The idea here is that we want the log-likelihood, or distribution defining the data, to be left as much as possible as it is. By specifying a uniform distribution across the range of parameter values, we are saying that nothing is influencing the data. There can be problems at times with using the uniform prior, though, since it may be improper, so most Bayesians prefer to use the normal distribution with mean 0 and a very small precision value e.g., 0.0001. JAGS uses a precision statistic in place of the variance or even the standard deviation as the scale parameter for the normal distribution. The precision, though, is simply the inverse of the variance. A precision of 0.0001 is the same as a variance of 10 000. This generally guarantees that the variance is spread so wide that it is equal for all values in the range of the response.

The second block in the `NORM` model relates to defining priors for the variance σ^2 . Sigma is the standard deviation, which is given a uniform distribution running from 0 to 100. The precision tau (τ) is then given as the inverse of σ^2 , i.e. `pow(sigma, -2)`.

The third block defines the normal probability with the betas and the means and the precision as the inverse variance. As a synthetic model, this defines the random normal values adjusted by the

mean and variance. The line `mu[i] <- eta[i]` assigns the values of the linear predictor of the model, obtained from `inprod(beta[], x[i,])`, to `mu`, which is commonly regarded as the term indicating the fitted or predicted value. For a normal model `eta = mu`. This will not be the case for other models we discuss in the book.

It should be noted that, when using JAGS, the order in which we place the code directives makes no difference. In the code lines above it would seem to make sense to calculate the linear predictor, `eta`, first and then assign it to `mu`. The software is actually doing this, but it is not apparent from viewing the code.

The next block of code provides initial values for each parameter in the model. The initial values for `beta` are specified as 0 mean and 0.01 precision. These values are put into a list called `beta`. Following the initial values, the code specifies the parameters to be estimated, which are the `beta` or mean values and `sigma`, the standard deviation.

The final block of code submits the information above into the MCMC sampling algorithm, which is in fact a variety of Gibbs sampling. The details of how the sampling is to occur are specified in the argument list. The model algorithm samples the data 15 000 times (`m.iter`); the first 10 000 are used as burn-in values. These values are not kept. However, the next 5000 are kept and define the shape of the posterior distributions for each parameter in the model.

The final line in the code prints the model results to the screen. There are a variety of ways of displaying this information, some of which are quite complicated. Post-estimation code can be given for the model diagnostics, which are vital to proper modeling. We shall discuss diagnostics in the context of examples.

Recall that we mentioned that “non-informative” priors are not completely non-informative. Some information is entailed in the priors even if we have no outside information to use for adjusting the model parameters. Moreover, a prior that has little impact on one type of likelihood may in fact have a rather substantial impact on another type of likelihood. However, the non-informative priors used in this book are aimed to truly minimize outside information from influencing or adjusting the given model data.

Since we employed non-informative priors on the above synthetic Bayesian normal model, we should expect that the posterior means are close to the values we set for each parameter. In the top R code used to define the data, we specified values of 2 for the intercept β_0 , 3 for the slope β_1 , and 1 for the standard deviation σ . The results of our model are 2.06 for the intercept, 2.92 for the slope, and 0.99 for the standard deviation. This is a simple model, so the MCMC sampling algorithm does not require a very large number of iterations in order to converge. When the data consist of a large number of factor variables, and a large number of observations, or when we are modeling mixed and multilevel models, convergence may require a huge number of sampling iterations. Moreover, it may be necessary to skip one or more iterations when sampling, a process called thinning. This helps ameliorate, if not eliminate, autocorrelation in the posterior distribution being generated.

Another tactic when having problems with convergence is to enlarge the default burn-in number.

When engaging a sampling algorithm such as MCMC the initial sample is very likely to be far different from the final posterior distribution. Convergence as well as appropriate posterior values are obtained if the initial sampling values are excluded from the posterior distribution. Having a large number of burn-in samples before accepting the remainder of the sample values generally results in superior estimates. If we requested more iterations (`n.iter = 50000`) and a greater burn-in number (`n.burnin = 30000`) it is likely that the mean, standard deviation, and credible interval would be closer to the values we defined for the model. Experimentation and trial-and-error are common when working with Bayesian models. Remember also that we have used a synthetic model here; real data is much more messy.

Finally, whenever reporting results from a Bayesian model, it is important to be clear about the elements enclosed in a given implementation. In this context, the Gaussian normal model studied in this section can be reported as

$$\begin{aligned}
 Y_i &\sim \text{Normal}(\mu_i, \sigma^2) \\
 g(\mu_i) &= \eta_i \\
 \eta_i &\equiv \mathbf{x}_i^T \boldsymbol{\beta} = \beta_1 + \beta_2 x_1 \\
 \beta_1 &\sim \text{Normal}(0, 10^4) \\
 \beta_2 &\sim \text{Normal}(0, 10^4) \\
 \sigma^2 &\sim \text{Uniform}(0, 10^2) \\
 i &= 1, \dots, N
 \end{aligned} \tag{4.6}$$

where N is the number of observations. Notice that this is almost a direct mathematical translation from the JAGS implementation shown in Code 4.1. Although in this particular example the description is trivial, this type of representation can be extremely helpful as models increase in complexity, as we will show in the following chapters.

4.1.2 Bayesian Synthetic Normal Model in R using JAGS and the Zero Trick

For complex non-linear models it may be difficult to construct the correct log-likelihood and associated statistics. Analysts commonly use what is termed the *zero trick* when setting up a Bayesian model. We shall use this method often throughout the book since it makes estimation easier. In some cases it is helpful to display the codes for both standard and zero-trick approaches.

The zero trick¹ is based on the probability of zero counts in a Poisson distribution with a given mean. You may come across references to a one-trick model as well, which is based on the probability of obtaining the value 1 from a Bernoulli or binary logistic distribution. The zero-trick approach is much more common than the one-trick approach, so we focus on it.

The code below converts a standard JAGS model to a model using the zero trick for estimation of the posterior distributions of the model parameters. We use the same data and same model set-up. All that differs is that the zero trick is used. It is instructive to use this method since the reader can easily observe the differences and how to set up the zero trick for his or her own purposes.

The amendments occur in the `model.data` function, where we add `zeros = rep(0,obs)` to the list. In the likelihood block we have added `c <- 1000` and two lines beginning with `zero`. We give `c` the value 1000 and add it to the likelihood to ensure positive values. Otherwise, all is the same.

Code 4.2 Normal linear model in R using JAGS and the zero trick.

```
=====
require(R2jags)
set.seed(1056)                      # set seed to replicate example
nobs = 5000                          # number of obs in model
x1 <- runif(nobs)                   # predictor, random uniform variable

beta0 = 2.0                           # intercept
beta1 = 3.0                           # predictor
xb <- beta0 + beta1 * x1            # linear predictor, xb
y <- rnorm(nobs, xb, sd=1)          # y as an adjusted random normal
                                     variate

# Model setup
X <- model.matrix(~ 1 + x1)
K <- ncol(X)
model.data <- list(Y = y,           # Response variable
                    X = X,           # Predictors
                    K = K,           # Number of predictors including
                                     the intercept
                    N = nobs,         # Sample size
                    Zeros = rep(0, nobs) # Zero trick
)
NORM0 <- "
model{
    # Diffuse normal priors for predictors
    for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001) }

    # Uniform prior for standard deviation
    tau <- pow(sigma, -2)      # precision
    sigma ~ dunif(0, 100)      # standard deviation

# Likelihood function
C <- 10000
for (i in 1:N){
    Zeros[i] ~ dpois(Zeros.mean[i])
    Zeros.mean[i] <- -L[i] + C
    l1[i] <- -0.5 * log(2*3.1416) - 0.5 * log(sigma)
    l2[i] <- -0.5 * pow(Y[i] - mu[i],2)/sigma
    L[i] <- l1[i] + l2[i]
    mu[i] <- eta[i]
    eta[i] <- inprod(beta[], X[i,])
}
}

inits <- function () {
    list(
        beta = rnorm(K, 0, 0.01))
}
params <- c("beta", "sigma")

norm0fit <- jags(data = model.data,
                  inits = inits,
                  parameters = params,
                  model = textConnection(NORM0),
                  n.chains = 3,
                  n.iter = 15000,
                  n.thin = 1,
                  n.burnin = 10000)

print(norm0fit,intervals=c(0.025, 0.975), digits=2)
```

```
=====
          mu.vect    sd.vect      2.5%     97.5%   Rhat   n.eff
beta[1]        1.99      0.03     1.94      2.05     1  10000
beta[2]        3.01      0.05     2.91      3.11     1  13000
sigma         1.00      0.02     0.96      1.04     1  15000
deviance    100014175.71    2.50  100014172.92  100014182.20     1       1
pD = 3.1 and DIC = 100014178.8
```

The values are the same as in the model not using the zero trick. We have displayed both models to assure you that the zero trick works. We shall use it frequently for other models discussed in this volume.

4.1.3 Bayesian Synthetic Normal Model in Python using Stan

In what follows we use the synthetic data generated in Section 3.1.2 and stored in the vectors x_1 and y . Recall that data must be written as a dictionary so that it can be used as input for a Stan model.

Code 4.3 Normal linear model in Python using Stan.

```
=====
import pystan
import numpy as np
from scipy.stats import uniform
import statsmodels.api as sm

# Data
np.random.seed(1056)                      # set seed to replicate example
nobs = 5000                                 # number of obs in model
x1 = uniform.rvs(size=nobs)                 # random uniform variable

x1.transpose()                             # create response matrix
X = sm.add_constant(x1)                   # add intercept
beta = [2.0, 3.0]                          # create vector of parameters

xb = np.dot(X, beta)                      # linear predictor
y = np.random.normal(loc=xb, scale=1.0, size=nobs) # create y as adjusted
                                                # random normal variate

# Fit
toy_data = {}                                # build data dictionary
toy_data['nobs'] = nobs                     # sample size
toy_data['x'] = x1                           # explanatory variable
toy_data['y'] = y                            # response variable

stan_code = """
data {
    int<lower=0> nobs;
    vector[nobs] x;
    vector[nobs] y;
}
parameters {
    real beta0;
    real beta1;
    real<lower=0> sigma;
}
model {
    vector[nobs] mu;

    mu = beta0 + beta1 * x;
    y ~ normal(mu, sigma);      # Likelihood function
}"""

fit = pystan.stan(model_code=stan_code, data=toy_data, iter=5000,
                   chains=3, verbose=False)

# Output
nlines = 8                                     # number of lines on screen output
output = str(fit).split('\n')
```

```

for item in output[:nlines]:
    print(item)
=====
The summary dumped to the screen will have the form
      mean   se_mean     sd  2.5%   25%   50%   75%  97.5%  n_eff  Rhat
beta0  1.99    7.7e-4   0.03  1.93  1.97  1.99  2.01  2.04  1349.0  1.0
beta1   3.0    1.4e-3   0.05   2.9  2.96   3.0  3.03   3.1  1345.0  1.0
sigma   0.99    2.4e-4  9.9e-3  0.97  0.98  0.99   1.0  1.01  1683.0  1.0

```

The posterior distribution and MCMC chains can be visualized using

Code 4.4 Plotting posteriors from Code 4.3.

```

=====
import pylab as plt

# Plot posterior and chain
fit.plot(['beta0', 'beta1', 'sigma'])
plt.tight_layout()
plt.show()
=====
```

4.1.4 Bayesian Synthetic Normal Model using Stan with a Customized Likelihood

In the example above we used the built-in normal distribution in Stan by employing the symbol \sim in the `model` code block when defining the likelihood. Stan also offers the possibility to implement custom probability distributions without the necessity to use the zero trick. As an example, if you wish to write your own normal distribution in the above code, it is enough to replace the `model` block by

Code 4.5 Modifications to be applied to Code 4.3 in order to use a customized likelihood.

```

=====
model {vector[nobs] mu;
  real temp_const;
  vector[nobs] loglike;
  mu = beta0 + beta1 * x;

  temp_const = log(1.0/(sigma * sqrt(2 * pi()))); # multiplicative constant
  for (i in 1:nobs) {
    # loglikelihood
    loglike[i] = temp_const - pow((y[i] - mu[i]), 2) / (2 * pow(sigma, 2));
  }
  target += loglike;
}
=====
```

All other parts of the code remain the same. This feature is redundant at this point but, as explained in Section 4.1.2, it greatly enlarges the range of models that we are able to address.

4.2 Multivariate Normal Model

The model can be easily generalized for more predictors when the response variable y is a linear combination of a set of explanatory variables x . In this case, Equation 4.4 takes the form

$$y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \cdots + \beta_j x_{ji} + \varepsilon, \quad (4.7)$$

where each term of $\beta_i x_j$ depicts a predictor x_j and its associated slope or coefficient β_i . In addition, ε represents the error in predicting the response variable on basis of the explanatory variables, i runs through all available observations in the model, and j represents the number of predictors in the model.

The mathematical representation can be easily extended to

$$\begin{aligned}
 Y_i &\sim \text{Normal}(\mu_i, \sigma) \\
 g(\mu_i) &= \eta_i \\
 \eta_i &\equiv \mathbf{x}_i^T \boldsymbol{\beta} = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_N x_{iN} \\
 \beta_j &\sim \text{Normal}(0, 10^4) \\
 \sigma &\sim \text{Uniform}(0, 10^2) \\
 i &= 1, \dots, N \\
 j &= 1, \dots, J
 \end{aligned} \tag{4.8}$$

4.2.1 Multivariate Linear Regression in R using JAGS

Code 4.6 Multivariate normal linear model in R using JAGS.

```

=====
require(R2jags)
set.seed(1056)                      # set seed to replicate example
nobs = 5000                           # number of obs in model
x1 <- runif(nobs)                     # random uniform variable
x2 <- runif(nobs)                     # random uniform variable

beta1 = 2.0                            # intercept
beta2 = 3.0                            # 1st coefficient
beta3 = -2.5                           # 2nd coefficient

xb <- beta1 + beta2*x1 + beta3*x2   # linear predictor
y <- rnorm(nobs, xb, sd=1)           # create y as adjusted random normal
                                         # variate

# Model setup
X <- model.matrix(~ 1 + x1+x2)
K <- ncol(X)
model.data <- list(Y = y,            # response variable
                    X = X,          # predictors
                    K = K,          # number of predictors including the intercept
                    N = nobs)       # sample size
)

NORM <- "
model{
  # Diffuse normal priors for predictors
  for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001) }

  # Uniform prior for standard deviation
  tau <- pow(sigma, -2)                # precision
  sigma ~ dunif(0, 100)                 # standard deviation

  # Likelihood function
  for (i in 1:N){
    Y[i]~dnorm(mu[i],tau)
    mu[i] <- eta[i]
    eta[i] <- inprod(beta[], X[i,])
  }
}
```

```

        }
    }"
inits <- function () {
  list (
    beta = rnorm(K, 0, 0.01))
}
params <- c ("beta", "sigma")

norm0fit <- jags (data = model.data,
  inits = inits,
  parameters = params,
  model = textConnection (NORM),
  n.chains = 3,
  n.iter = 3,
  n.iter = 15000,
  n.thin = 1,
  n.burnin = 10000)
print (norm0fit, intervals=c(0.025, 0.975), digits=2)
=====
      mu.vect sd.vect    2.5%    97.5%   Rhat n.eff
beta[1]     2.016   0.038    1.941    2.089 1.001  7500
beta[2]     3.031   0.049    2.933    3.126 1.001  7500
beta[3]    -2.552   0.048   -2.648   -2.457 1.001  7500
sigma       1.013   0.010    0.993    1.033 1.001  7500
deviance 14317.084   2.805 14313.580 14324.312 1.001  7500
pD = 3.9 and DIC = 14321.0
```

4.2.2 Multivariate Linear Regression in Python using Stan

In order to implement a multivariate model in Stan we need to change only a few items in the code shown in Section 4.1.3. Considering the case with two explanatory variables, we need to introduce `x2` into the data dictionary as well as into the Stan code. This increases the number of lines displayed in the output.

Although we are dealing here with only two explanatory variables, the same reasoning may be applied to a higher dimensional case with a finite number of predictors. In order to avoid the explicit definition of variables and data vectors, we show in the code below a vectorized version of the Stan code displayed in Section 4.1.3.

In what follows, all the slope coefficients are stored as elements of a vector and all data are given as a matrix. Such a compact notation facilitates the application of this model to a high-dimensional problem.

Code 4.7 Multivariate normal linear model in Python using Stan.

```
=====
import numpy as np
import statsmodels.api as sm
import pystan

from scipy.stats import uniform, norm

# Data
np.random.seed(1056)                      # set seed to replicate example
nobs = 5000                                  # number of obs in model
x1 = uniform.rvs(size=nobs)                  # random uniform variable
x2 = uniform.rvs(size=nobs)                  # second explanatory

X = np.column_stack((x1,x2))                # create response matrix
X = sm.add_constant(X)                      # add intercept
beta = [2.0, 3.0, -2.5]                     # create vector of parameters

xb = np.dot(X, beta)                         # linear predictor, xb
y = np.random.normal(loc=xb, scale=1.0, size=nobs) # create y as adjusted
                                                # random normal variate

# Fit
toy_data = {}
toy_data['y'] = y                           # response variable
toy_data['x'] = X                           # predictors
toy_data['k'] = toy_data['x'].shape[1]        # number of predictors including
                                              # intercept
toy_data['nobs'] = nobs                     # sample size

# Stan code
stan_code = """
data {
    int<lower=1> k;
    int<lower=0> nobs;
    matrix[nobs, k] x;
    vector[nobs] y;
}
parameters {
    matrix[k,1] beta;
    real<lower=0> sigma;
}
transformed parameters{
    matrix[nobs,1] mu;
    vector[nobs] mu2;

    mu = x * beta;                          # normal distribution
    mu2 = to_vector(mu);                   # does not take matrices as input
}
model {
    for (i in 1:k){                      # diffuse normal priors for predictors
        beta[i] ~ normal(0.0, 100);
    }
    sigma ~ uniform(0, 100);            # uniform prior for standard deviation
    y ~ normal(mu2, sigma);           # likelihood function
}
"""

# Run mcmc
fit = pystan.stan(model_code=stan_code, data=toy_data, iter=5000, chains=3,
                    verbose=False, n_jobs=3)

# Output
nlines = 9                                     # number of lines to appear on screen
output = str(fit).split('\n')
for item in output[:nlines]:
    print(item)
=====
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
beta[0,0]	2.02	9.1e-4	0.04	1.94	1.99	2.02	2.04	2.09	1645.0	1.0
beta[1,0]	2.97	1.2e-3	0.05	2.88	2.94	2.98	3.01	3.07	1712.0	1.0
beta[2,0]	-2.49	1.2e-3	0.05	-2.58	-2.52	-2.49	-2.46	-2.4	1683.0	1.0
sigma	0.99	2.3e-4	9.9e-3	0.97	0.99	0.99	1.0	1.01	1922.0	1.0

Notice that in the block of transformed parameters it is necessary to define two new parameters: one to store the result of multiplying the data and parameter matrices, which returns a matrix object of dimensions $\{\text{nobs}, 1\}$, and another to transform this result into a vector object. This is necessary since the normal sampler accepts only scalars or vectors as a mean. The fitting and plotting routines are the same as those presented in Section 4.1.3.

4.3 Bayesian Errors-in-Measurements Modeling

The standard methodology for regression modeling often assumes that the variables are measured without errors or that the variance of the uncertainty is unknown. Most quantities in astronomy arise from measurements which are taken with some error. When the measurement error is large relative to the quantity being measured, it can be important to introduce an explicit model of errors in variables. A Bayesian approach to errors-in-variables models is to treat the true quantities being measured as missing data (Richardson and Gilks, 1993). This requires a model of how the measurements are derived from the true values. Let us assume a normal linear model where the true values of the predictor, x_i , and the response, y_i , variables are not known, but only the observed quantities x_i^{obs} and y_i^{obs} . If the measurement uncertainties σ_x and σ_y can be estimated, the observed values can be modeled as function of the true values plus a measurement noise. A common approach is to assume the measurement error as normal with known deviation: $\varepsilon_x \sim \text{Normal}(0, \sigma_x)$ and $\varepsilon_y \sim \text{Normal}(0, \sigma_y)$.

A synthetic normal model with measurement errors in both the x and y variables may be created in R using the code given below.

4.3.1 Generating Data with Errors using R

Code 4.8 Synthetic normal data in R with errors in measurements.

```
=====
# Data
set.seed(1056)          # set seed to replicate example
nobs = 1000              # number of obs in model
sdobsx <- 1.25
truex <- rnorm(nobs, 0, 2.5)    # normal variable
errx <- rnorm(nobs, 0, sdobsx)
obsx <- truex + errx
beta1 <- -4
beta2 <- 7
sdyy <- 1.25
sdobsy <- 2.5
erry <- rnorm(nobs, 0, sdobsy)
truey <- rnorm(nobs, beta1 + beta2*truex, sdyy)
obsy <- truey + erry
=====
```

This code has only one predictor, x_1 , with assigned coefficient 7 and intercept -4 . First, we try to make an inference about the parameters ignoring the presence of errors.

The JAGS code for a Bayesian normal model, ignoring errors, is given below.

4.3.2 Build Model ignoring Errors in R using JAGS

Code 4.9 Normal linear model in R using JAGS and ignoring errors in measurements.

```
=====
Require (R2 jags)
K <- 2
model.data <- list(obsy = obsy,
                     obsx = obsx,
                     K = K,
                     N = nobs)

NORM <-" model{
  # diffuse normal priors for predictors
  for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001) }

  # uniform prior for standard deviation
  tau <- pow(sigma, -2)      # precision
  sigma ~ dunif(0, 100)       # standard deviation

  # likelihood function
  for (i in 1:N){
    obsy[i]~dnorm(mu[i],tau)
    mu[i]  <- eta[i]
    eta[i] <- beta[1]+beta[2]*obsx[i]
  }
}"
# Initial value
inits <- function () {
  list(
    beta = rnorm(K, 0, 0.01))
}
# Parameters to display and save
params <- c("beta", "sigma")

normefit <- jags(data = model.data,
                  inits = inits,
                  parameters = params,
                  model = textConnection(NORM),
                  n.chains = 3,
                  n.iter = 10000,
                  n.thin = 1,
                  n.burnin = 5000)
print(normefit,intervals=c(0.025, 0.975), digits=3)
=====
```

	mu.vect	sd.vect	2.5%	7.5%	Rhat	n.eff
beta[1]	-4.075	0.271	-4.620	-3.549	1.001	25000
beta[2]	5.637	0.093	5.457	5.520	1.001	15000
sigma	8.554	0.192	8.182	8.944	1.001	15000
deviance	7128.678	2.470	7125.898	7135.215	1.002	2500

pD = 3.0 and DIC = 7131.7

4.3.3 Build Model including Errors in R using JAGS

Now we take errors into account:

Code 4.10 Normal linear model in R using JAGS and including errors in variables.

```
=====
Require (R2 jags)
model.data <- list(obsy = obsy,
                     obsx = obsx,
                     K = K,
                     errx = errx,
                     erry = erry,
                     N = nobs)

NORM_err <-" model{
  # Diffuse normal priors for predictors
  for (i in 1:K) { beta[i] ~ dnorm(0, 1e-3) }

  # Uniform prior for standard deviation
  tauy <- pow(sigma, -2)      # precision
  sigma ~ dunif(0, 100)        # diffuse prior for standard deviation

  # Diffuse normal priors for true x
```

```

for (i in 1:N){
  x[i]~dnorm(0,1e-3)
}

# Likelihood
for (i in 1:N){
  obsy[i]~dnorm(y[i],pow(erry[i],-2))
  y[i]~dnorm(mu[i],tauy)
  obsx[i]~dnorm(x[i],pow(errx[i],-2))
  mu[i]<-beta[1]+beta[2]*x[i]
}

}"
}

# Initial values
inits <- function () {
  list(
    beta = rnorm(K, 0, 0.01))
}

# Parameter to display and save
params <- c("beta", "sigma")

evfit <- jags(data = model.data,
               inits = inits,
               parameters = params,
               model = textConnection(NORM_err),
               n.chains = 3,
               n.iter = 5000,
               n.thin = 1,
               n.burnin = 2500)
print(evfit,intervals=c(0.025, 0.975), digits=3)
=====
      mu.vect    sd.vect      2.5%     97.5%   Rhat  n.eff
beta[1]    -4.065    0.138    -4.337    -3.797  1.002  1600
beta[2]     6.756    0.056     6.645     6.862  1.014   150
sigma       1.310    0.176     0.970     1.666  1.050    49
deviance   5509.445   49.959   5412.902   5608.426  1.003   910
pD = 1245.4 and DIC = 6754.8

```

The inference with the errors-in-variables model is much closer to the assigned values; in particular, ignoring the errors largely overestimates the intrinsic scatter of the data. Figure 4.5 enables a comparison of the two models.

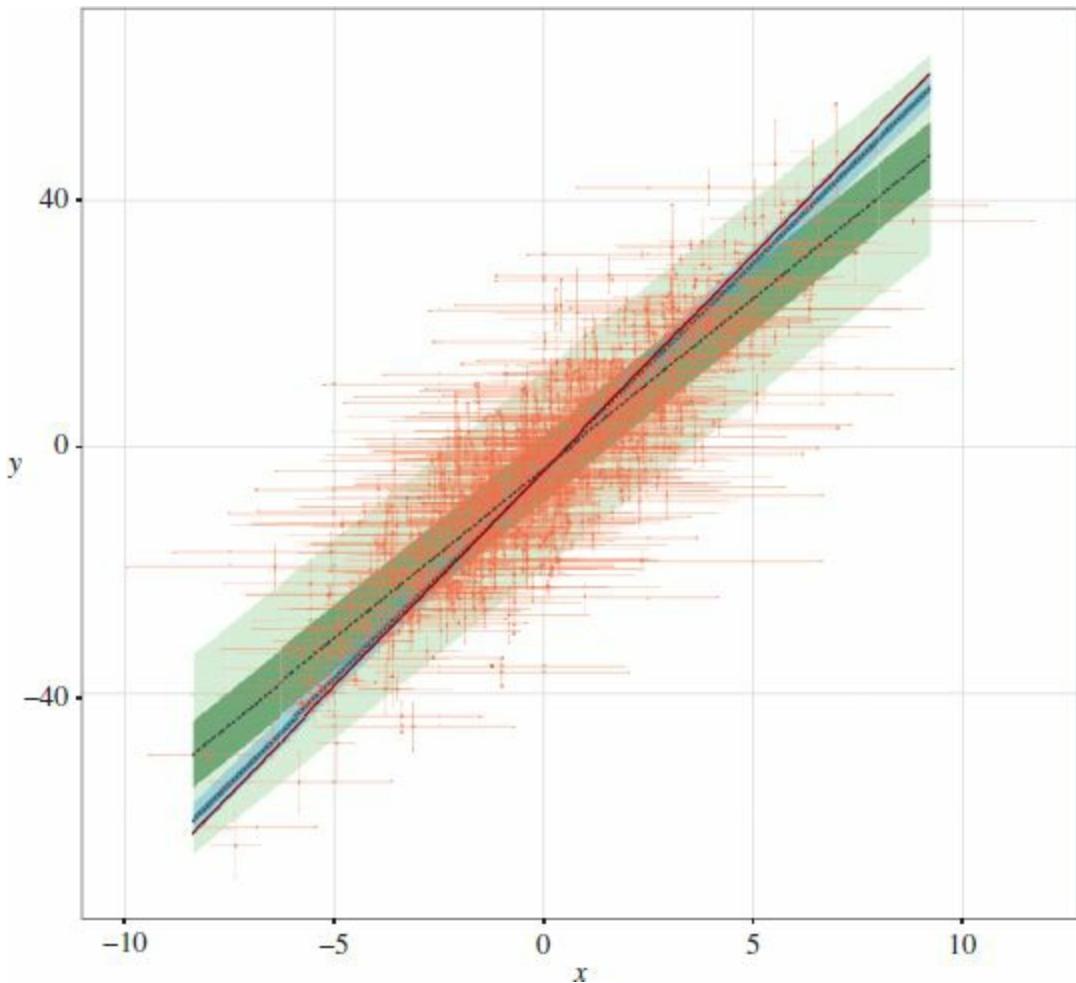


Figure 4.5 Visualization of different elements of a Bayesian normal linear model that includes errors in variables. The dots and corresponding error bars represent the data. The dashed line and surrounding shaded bands show the mean, 50% (darker), and 95% (lighter) credible intervals when the errors are ignored. The dotted line and surrounding shaded bands show the mean, 50% (darker), and 95% (lighter) credible intervals obtained when the errors are taken into account (note that these shaded bands are very narrow). The solid line (i.e., the line with the steepest slope) shows the fiducial model used to generate the data.

4.3.4 Bayesian Errors-in-Measurements Modeling in Python using Stan

Code 4.11 Normal linear model in Python using Stan and including errors in variables.

```
=====
import numpy as np
import statsmodels.api as sm
import pystan

from scipy.stats import norm

# Data
np.random.seed(1056)                      # set seed to replicate example
nobs = 1000                                 # number of obs in model
sdobsx = 1.25
truex = norm.rvs(0, 2.5, size=nobs)          # normal variable
errx = norm.rvs(0, sdobsx, size=nobs)        # errors
obsx = truex + errx                         # observed

beta0 = -4
beta1 = 7
sdy = 1.25
sdobsy = 2.5
```

```

erry = norm.rvs(0, sdobsy, size=nobs)
truey = norm.rvs(beta0 + beta1*truex, sdy, size=nobs)
obsy = truey + erry

# Fit
toy_data = {}                                # build data dictionary
toy_data['N'] = nobs                         # sample size
toy_data['obsx'] = obsx                      # explanatory variable
toy_data['errx'] = errx                      # uncertainty in explanatory variable
toy_data['obsy'] = obsy                      # response variable
toy_data['erry'] = erry                      # uncertainty in response variable
toy_data['xmean'] = np.repeat(0, nobs) # initial guess for true x position

# Stan code
stan_code = """
data {
    int<lower=0> N;
    vector[N] obsx;
    vector[N] obsy;
    vector[N] errx;
    vector[N] erry;
    vector[N] xmean;
}
transformed data{
    vector[N] varx;
    vector[N] vary;
    for (i in 1:N){
        varx[i] = fabs(errx[i]);
        vary[i] = fabs(erry[i]);
    }
}
parameters {
    real beta0;
    real beta1;
    real<lower=0> sigma;
    vector[N] x;
    vector[N] y;
}
transformed parameters{
    vector[N] mu;
    for (i in 1:N){
        mu[i] = beta0 + beta1 * x[i];
    }
}
model{
    beta0 ~ normal(0.0, 100);      # diffuse normal priors for predictors
    beta1 ~ normal(0.0, 100);

    sigma ~ uniform(0.0, 100);     # uniform prior for standard deviation

    x ~ normal(xmean, 100);
    obsx ~ normal(x, varx);
    y ~ normal(mu, sigma);
    obsy ~ normal(y, vary);
}
"""

# Run mcmc
fit = pystan.stan(model_code=stan_code, data=toy_data, iter=5000, chains=3,
                    n_jobs=3, warmup=2500, verbose=False, thin=1)

# Output
nlines = 8                                     # number of lines in screen output

output = str(fit).split('\n')
for item in output[:nlines]:
    print(item)
=====
Output on screen:
      mean se_mean    sd  2.5%   25%   50%   75%  97.5%   n_eff   Rhat
beta0 -3.73  3.2e-3  0.14 -4.02 -3.83 -3.73 -3.63 -3.45  2049.0  1.0
beta1  6.68  2.4e-3  0.06  6.56  6.64  6.68  6.72  6.8   632.0  1.01
sigma  1.69  0.01  0.19  1.33  1.56  1.69  1.82  2.07  173.0  1.03

```

Further Reading

- Feigelson, E. D. and G. J. Babu (1992). “Linear regression in astronomy. II”. *ApJ* 397, 55–67. DOI: [10.1086/171766](https://doi.org/10.1086/171766)
- Gelman, A., J. Carlin, and H. Stern (2013). *Bayesian Data Analysis*, Third Edition. Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis.
- Isobe, T. *et al.* (1990). “Linear regression in astronomy.” *ApJ* 364, 104–113. DOI: [10.1086/169390](https://doi.org/10.1086/169390)
- Kelly, B. C. (2007). “Some aspects of measurement error in linear re-gression of astronomical data.” *ApJ* 665, 1489–1506. DOI: [10.1086/519947](https://doi.org/10.1086/519947). arXiv: [0705.2774](https://arxiv.org/abs/0705.2774)
- Zuur, A. F., J. M. Hilbe, and E. N. Ieno (2013). *A Beginner’s Guide GLM and GLMM with R: A Frequentist and Bayesian Perspective for Ecologists*. Highland Statistics.
-

¹ The zero trick is a way to include a given customized likelihood, $L(\theta)$, not available in JAGS. It uses the fact that for a Poisson distribution, the probability of observing a zero is $e^{-\mu}$. So, if we observe a data point with a value of zero, a Poisson distribution with mean $\log(L(\theta))$ yields the probability $L(\theta)$. But because $\log(L(\theta))$ must always be a positive value (a requirement of the Poission distribution), a large constant needs to be added to the likelihood to ensure that it is always positive.

5 GLMs Part I – Continuous and Binomial Models

5.1 Brief Overview of Generalized Linear Models

The normal model, also referred to as linear regression, was discussed in Chapter 4. The characteristic assumption of the normal model is that the data being modeled is normally distributed. A normal or Gaussian distribution has two parameters, a location or mean parameter μ and a variance or scale parameter σ^2 . In most frequency-based linear models the variance is not estimated but is regarded as having a fixed or constant value. A full maximum likelihood estimation of a normal model, however, does estimate both the mean and variance parameters. We found that both parameters are estimated in a Bayesian normal model. It is important to remember that analysts employing a Bayesian model nearly always estimate all the parameters associated with the probability function considered to underlie a model.

Generalized linear models (GLMs) are fundamental to much contemporary statistical modeling. The normal or linear model examined in the previous chapter is also a GLM and rests at the foundation of this class of models. However, the true power of GLMs rests in the extensions that can be drawn from it. The results are binary, binomial or proportional, multinomial, count, gamma, and inverse Gaussian models, mixtures of these, and panel models for the above to account for clustering, nesting, time series, survival, and longitudinal effects. Other models can be developed from within this framework, so it is worth taking time to outline what is involved in GLM modeling and to see how it can be extended. Bayesian models can be derived from these distributions, and combinations of distributions, in order to obtain a posterior distribution that most appropriately represents the data being modeled. Despite the ubiquitous implementation of GLMs in general statistical applications, there have been only a handful of astronomical studies applying GLM techniques such as logistic regression (e.g. [de Souza et al., 2015a, 2016; Lansbury et al., 2014; Raichoor and Andreon, 2014](#)), Poisson regression (e.g. [Andreon and Hurn, 2010](#)), gamma regression ([Elliott et al., 2015](#)), and negative binomial (NB) regression ([de Souza et al., 2015b](#)).

The generalized linear model approach was developed by statisticians [John Nelder and Robert Wedderburn in 1972](#) while working together at the Rothamsted Experimental Station in the UK. Two years later they developed a software application for estimating GLMs called Generalized Linear Interactive Models (GLIM), which was used by statisticians worldwide until it was discontinued in 1994. By this time large commercial statistical packages were beginning to have GLM procedures as part of their official packages. The earliest of these procedures were authored

by Nelder for GenStat in the late 1970s and by Trevor Hastie (then at AT&T) for S in the mid 1980s; this was later incorporated into S-Plus and then later into R. The first author of this book authored a full GLM procedure in Stata, including the negative binomial, in late 1992 and then with Berwin Turlach did the same for XploRe economic statistical software in 1993. Gordon Johnston of SAS created the SAS/STAT Genmod procedure for GLM models using SAS in 1994. Now nearly all statistical packages have GLM capabilities.

The traditional or basic GLM algorithm provides a unified way to estimate the mean parameter of models belonging to the single-parameter exponential family of distributions. The random response variable, Y_i , $i = 1, 2, \dots, n$, may be represented as

$$\begin{aligned} Y_i &\sim f(\mu_i, a(\phi)V(\mu_i)), \\ g(\mu_i) &= \eta_i, \\ \eta_i &\equiv \mathbf{x}_i^T \boldsymbol{\beta} = \beta_0 + \beta_1 x_1 + \dots + \beta_p x_p. \end{aligned} \tag{5.1}$$

In Equation 5.1, f denotes a response variable distribution from the exponential family (EF), μ_i is the response variable mean, ϕ is the EF dispersion parameter in the dispersion function $a(\cdot)$, $V(\mu_i)$ is the variance function, η_i is the linear predictor, $\mathbf{x}_i = \{x_{i1}, x_{i2}, \dots, x_{ip}\}$ is the vector of the explanatory variables (covariates or predictors), $\boldsymbol{\beta} = \{\beta_1, \beta_2, \dots, \beta_p\}$ is the vector of the covariates' coefficients, and $g(\cdot)$ is the link function, which connects the mean to the predictor. If $V(\mu_i)$ is a constant for all μ_i then the mean and variance of the response are independent, which allows the use of a Gaussian response variable. If the response is Gaussian then the linear predictor $\eta = g(\mu) = \mu$. This relationship is referred to as the identity link and is characteristic of the normal or Gaussian model. Combinations of distributions and link functions define the various members of the GLM family of models. Hence, GLMs incorporate linear regression as a subset, taking the form

$$\begin{aligned} Y_i &\sim Normal(\mu_i, \sigma^2), \\ \mu_i &= \beta_0 + \beta_1 x_1 + \dots + \beta_p x_p. \end{aligned} \tag{5.2}$$

The key feature of traditional GLM methodology is an iteratively reweighted least squares (IRLS) algorithm, which linearizes the relationship between the linear predictor and fitted value of the model being estimated. The IRLS algorithm is a simple yet robust method of estimating the mean parameter of this class of models. The advantage of using GLM methodology is that it is simple to construct and nearly always results in an analyst being able to obtain correct parameter estimates and related model statistics.

Generalized linear models were invented at a time when computing processing power was minimal compared with today. Using a GLM function for modeling usually meant that the analyst did not have to struggle to find appropriate starting or initial values, and if the model was specified correctly then convergence was almost always achieved. When the analyst used a then standard

non-linear regression to model binary or count data, for example, convergence frequently failed. Maximum likelihood modeling was tricky, to be sure, and usually took much longer to come to a solution than a GLM. Actually, GLM methodology is a variety of maximum likelihood modeling but is a simplification of it applied to models belonging to the exponential family of distributions. Contemporary maximum likelihood estimation algorithms are much more efficient than those constructed before the 1990s and usually do not have the problems experienced in earlier times. However, GLM methodology was, and still is, an efficient way to model certain data situations and is at the foundation of a number of more advanced models.

In order for a model to be a GLM, it must meet the following criteria:

1. The response term y is conditionally distributed as a single-parameter member of the exponential family of distributions.
2. The model has a monotonic and differentiable link function that linearizes the relationship of the linear predictor, η or $x\beta$, and the fitted value, μ . Statisticians usually symbolize the link function as $g(\mu)$, although other symbolizations exist.
3. An inverse link function $g^{-1}(\eta)$ exists that defines the fitted value.
4. Traditional GLMs use IRLS as their method of estimation. More recent implementations use a full maximum likelihood algorithm.

It should be noted that the use of a full maximum likelihood estimation algorithm allows estimation of a scale or ancillary parameter if the otherwise GLM model has such a second parameter. This is the case for models with a continuous variable and for the negative binomial count model.

The unique property of the exponential family of probability distributions is that the link, mean, and variance functions of the GLM distributions can easily be determined from the distribution – if it is cast in exponential-family form. The formula for the log-likelihood of the exponential family is given as

$$\mathcal{L} = \sum_{i=1}^n \left\{ \frac{y_i \theta_i - b(\theta_i)}{\alpha(\phi)} + c(y_i, \phi) \right\} \quad (5.3)$$

with y as the response variable to be modeled, θ , the link function, $b(\theta)$ the cumulant, and ϕ the scale parameter. The first derivative of the cumulant, with respect to θ , defines the mean of the distribution. The second derivative with respect to θ defines the distributional variance. The PDF normalization term $c()$ ensures that the distribution sums to 1. Note that the above equation provides for the estimation of both the mean and scale parameters. The scale parameter is applicable only for continuous response models and is generally ignored when one is modeling traditional GLMs. It is set to 1 for all GLM count models, including such models as logistic and probit regression, Poisson, and negative binomial regression. As we discuss later, though, the negative binomial has a second dispersion parameter, but it is not a GLM scale parameter so is dealt with in a different manner (it is

entered into the GLM algorithm as a constant). When estimated as single-parameter models, such as is the case with R's `glm` function, the interior terms of the above log-likelihood equation reduce to, for a single observation,

$$y\theta - b(\theta) + c(y). \quad (5.4)$$

For example, consider the Poisson PDF,

$$f(y; \mu) = \frac{e^{-\mu} (\mu)^y}{y!}. \quad (5.5)$$

Formatted into exponential-family form, the log-likelihood of the Poisson PDF with subscripts appears as,

$$\mathcal{L} = \sum_{i=1}^n \{y_i \ln(\mu_i) - \mu_i - \ln(y_i)\}. \quad (5.6)$$

The value of θ is then $\ln(\mu)$ or $\log \mu$, which is the Poisson link function. The first derivative of μ with respect to $\ln(\mu)$, is μ , as is the second derivative. The means and variances of the Poisson distribution and model are therefore identical, both being μ . In terms of the linear predictor $x\beta$, the mean and variance of the Poisson model is $\exp(x\beta)$. All the other GLM models share the above properties, which makes it rather easy for an analyst to switch between models.

It should also be mentioned that the canonical or natural link function is derived directly from the distribution, as is the case for the Poisson distribution above. However, one may change link functions within the same distribution. We may therefore have lognormal models or log-gamma models. In the note to Table 5.1 below, it is mentioned that the traditional negative binomial model used in statistical software is in fact a log-linked negative binomial. The link that is used is not derived directly from the negative binomial PDF (see the book [Hilbe, 2011](#)).

Table 5.1 The traditional GLM families with their variance and link functions.

Discrete distributions	Variance function	Link function
Bernoulli	$\mu(1 - \mu)$	$\log(\mu/(1 - \mu))$
Binomial	$\mu(1 - \mu/m)$	$\log(\mu/(m - \mu))$
Poisson	μ	$\log(\mu)$
Geometric	$\mu(1 + \mu)$	$\log(\mu/(1 + \mu))^{*}$
Negative binomial	$\mu(1 + \alpha\mu)$	$\log(\alpha\mu/(1 + \alpha\mu))^{*}$
Continuous distributions	Variance function	Link function
Gaussian or normal	$1 - \sigma^2$	μ

Gamma	μ^2	μ^2/v	$1/\mu$
Inverse Gaussian	μ^3	μ^3/v	$1/\mu^2$

* The geometric model is the negative binomial with $\alpha = 1$. It is not usually estimated as a model in its own right but is considered as a type of negative binomial. The geometric and negative binomial models are nearly always parameterized with a log link, like the Poisson. In using the same link as for the Poisson, the negative binomial can be used to adjust for overdispersion or extra correlation in otherwise Poisson data. The link functions displayed in the table are the canonical links. Lastly, the negative binomial has been parameterized here with a direct relation between the dispersion parameter and the mean. There is a good reason for this, which we discuss in the section on negative binomial models.

The variance functions in Table 5.1 show two columns for continuous response models. The second column displays the scale parameter for a two-parameter GLM. The scale parameter is defined as 1 for discrete response models, e.g., binomial and count models. Note that v is sometimes regarded as a shape parameter with the scale defined as $\phi = 1/v$.

As we shall find later in this chapter, the Bernoulli distribution is a subset of the binomial distribution. Both are used for models with binary responses, e.g. 0, 1. The m in the binomial variance and link functions indicates the binomial denominator. For the Bernoulli distribution $m = 1$. The geometric distribution is identical to the negative binomial with a dispersion parameter equal to 1. The Poisson can also be regarded as a negative binomial with a dispersion parameter equal to 0. These relationships can easily be observed by inspecting the variance functions in Table 5.1.

When estimating a negative binomial model within a traditional GLM program, a value for the dispersion parameter α is entered into the estimation algorithm as a constant. However, most contemporary GLM functions estimate α in a subroutine and then put it back into the regular GLM algorithm. The results are identical to a full maximum likelihood estimation (MLE) of the model.

Notice, as was mentioned before, the normal variance function is set at 1 and is not estimated when using a traditional GLM algorithm. When GLMs are estimated using MLE it is possible to estimate the variance (normal) and scale parameters for the gamma and inverse Gaussian models. Technically this is an extension to the basic GLM algorithm.

Recall our earlier discussion of the probability distribution function, the parameter values of which are assumed to generate the observations we are modeling. The equation used for estimating the distributional parameters underlying the model data is called the likelihood function. It is an equation which tells us how likely our model data is, given specific parameter values for the probability characterizing our model. The log of the likelihood function is used in the estimation process for frequency-based models as well as for most Bayesian models.

In order to better visualize how the IRLS algorithm works for a GLM, we shall construct a simple model using R. We shall provide values for an x continuous predictor and a binary variable y , which is the variable being modeled, the response variable. Since y is a binary variable, we model it using a logistic regression. To replicate the example below, place the code in the R editor, select it, and run. The output below displays the results of the simple GLM that we created using R's `glm` function. The top code is an example of IRLS. Note that it is a linear model re-weighted by the variance at each iteration.

Code 5.1 GLM logistic regression in R.

```
=====
# Data
x <- c(13,10,15,9,18,22,29,13,17,11,27,21,16,14,18,8)
y <- c(1,1,1,0,0,1,0,1,0,0,0,1,1,0,0)

# Fit
mu <- (y + 0.5)/2                      # initialize mu
eta <- log(mu/(1-mu))                   # initialize eta with the Bernoulli link
for (i in 1:8) {
  w <- mu*(1-mu)                         # variance function
  z <- eta + (y - mu)/(mu*(1-mu))       # working response
  mod <- lm(z ~ x, weights=w)            # weighted regression
  eta <- modsfit                          # linear predictor
  mu <- 1/(1+exp(-eta))                 # fitted value
  cat(i, coef(mod), "\n")
}

# Output
summary(mod)

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.28606   1.65744   0.776   0.451
x           -0.07915   0.09701  -0.816   0.428

# Logistic regression using glm package
import statsmodels.formula.api as smf

mylogit <- glm(y ~ x, family=binomial)
summary(mylogit)
=====

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) 1.28606   1.55294   0.828   0.408
x           -0.07915   0.09089  -0.871   0.384
```

The `glm` logistic model coefficients are identical to the coefficients of our synthetic model. The standard errors differ a bit but there are very few observations in the model, so we do not expect to have near identical standard errors and hence p-values. Remember that the z-value is the ratio of the coefficient and the standard error. It is easy to develop an R model that will produce equivalent standard errors for both the regular `glm` function and a synthetic model. But we need not be concerned with that here. See the book [Hilbe and Robinson \(2013\)](#) for a full explanation of how to program GLM as well as of Bayesian modeling.

We have provided a brief overview of traditional GLM methodology, understanding that it is squarely within the frequentist tradition of statistics. However, we will be discussing the Bayesian logistic model as well as the Bayesian normal model and others, as Bayesian alternatives to standard GLM models. We have earlier discussed the advantages of Bayesian modeling, but it may be helpful to observe the contrast.

The equivalent Python code for the example above concerning weighted regression may be written as:

Code 5.2 GLM logistic regression in Python.

```
=====
import numpy as np
import statsmodels.api as sm

# Data
x = np.array([13, 10, 15, 9, 18, 22, 29, 13, 17, 11, 27, 21, 16, 14, 18, 8])
y = np.array([1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0])
```

```

X = np.transpose(x)
X = sm.add_constant(X) # add intercept
# Fit
mu = (y + 0.5) / 2 # initialize mu
eta = np.log(mu/(1 - mu)) # initialize eta with the Bernoulli link
for i in range(8):
    w = mu * (1 - mu) # variance function
    z = eta + (y - mu)/(mu * (1 - mu)) # working response
    mod = sm.WLS(z, X, weights=w).fit() # weighted regression
    eta = mod.predict() # linear predictor
    mu = 1/(1 + np.exp(-eta)) # fitted value using inverse link function
print (mod.summary())
=====
      coeff    std err        t     P>|t|   [95.0% Conf. Int.]
-----
const    1.2861    1.657     0.776    0.451    -2.269    4.841
x1      -0.0792    0.097    -0.816    0.428    -0.287    0.129
=====
# Write data as dictionary
mydata = {}
mydata['x'] = x
mydata['y'] = y

# fit using glm package
import statsmodels.formula.api as smf

mylogit = smf.glm(formula='y ~ x', data=mydata, family=sm.families.Binomial())
res.summary()
res = mylogit.fit()
=====
      coeff    std err        z     P>|z|   [95.0% Conf. Int.]
-----
Intercept 1.2861    1.553     0.828    0.408    -1.758    4.330
x         -0.0792    0.091    -0.871    0.384    -0.257    0.099
=====
```

5.2 Bayesian Continuous Response Models

We have discussed the normal model in Chapter 4. It is the paradigm continuous response model, and it stands at the foundation of all other statistical models. With respect to generalized linear models, the normal model is an identity-linked model. The linear predictor, η or $x\beta$, and the fitted value, μ , are identical. No transformation of the linear predictor needs to take place in order to determine the fit or predicted value. The linear regression estimation process is therefore not iterative, whereas estimation using other GLMs is iterative. Bayesian modeling, however, still recognizes log-likelihood distributions as well as link functions that can be used within the Bayesian modeling code to convert, for instance, a Gaussian or normal model to a lognormal model or a Bernoulli logistic model to a Bernoulli probit model. Several important terms originating in GLM and maximum likelihood estimation are also used in Bayesian modeling, although their interpretations can differ.

We shall discuss four continuous response models in this section, two of which we will later demonstrate with astrophysical applications. Of course, we shall then focus completely on Bayesian methods for the estimation of model parameters. The models addressed in this section are the lognormal, gamma, inverse Gaussian, and beta models. The gamma and inverse Gaussian distributions are traditional GLM family members. The two-parameter lognormal and beta are not, but some authors have classified them as extended GLMs. A lognormal model may be created using GLM software by using a log link with the Gaussian family instead of the default identity link. However, only the mean parameter is estimated, not the scale. Frequentist two-parameter lognormal

software is difficult to locate, but we shall find it easy to construct using Bayesian techniques. In this section we create synthetic lognormal, log-gamma, log-inverse-Gaussian, and beta data, modeling each using a Bayesian model. Using the log parameterization for the gamma and inverse Gaussian distributions has better applications for modeling astronomical data than does using the canonical forms. We discuss this in the relevant subsections.

It should be mentioned that the lognormal, log-gamma, and log-inverse-Gaussian models are appropriate for modeling positive real numbers. Unlike in the normal model, in which it is assumed that the variance is constant across all observations, the lognormal variance is proportional to the square of the mean. There is no restriction for the log-gamma and log-inverse-Gaussian models, though. In comparison with the normal and lognormal models, the shape or scale parameters expand the range of modeling space for positive real values. The beta distribution is an important model, which is used for proportional data with values between 0 and 1, i.e., $\{0 < x < 1\}$.

5.2.1 Bayesian Lognormal Model

The lognormal model is commonly used on positive continuous data for which the values of the response variable are real. Figure 5.1 shows examples of the lognormal PDF for different parameter values and Figure 5.2 shows an example of lognormal-distributed data. Note the skewness as the data approaches zero on the y -axis. It is preferable to use a lognormal model rather than logging the response and modeling the result as a normal or linear model. A lognormal model can be created from the Gaussian or normal log-likelihood by logging the single y term in the normal distribution. Recalling the normal PDF and log-likelihood from Chapter 4, the lognormal PDF, which is sometimes referred to as the Galton distribution, can be expressed as

$$f(y; \mu, \sigma^2) = \frac{1}{y\sigma\sqrt{2\pi}} e^{-(\ln y - \mu)^2/2\sigma^2}, \quad (5.7)$$

while the associated log-likelihood assumes the form

$$\mathcal{L}(\mu, \sigma^2; y) = \sum_{i=1}^n \left\{ -\frac{(\ln y_i - \mu_i)^2}{2\sigma^2} - \frac{1}{2} \ln(2\pi\sigma^2 y_i^2) \right\}. \quad (5.8)$$

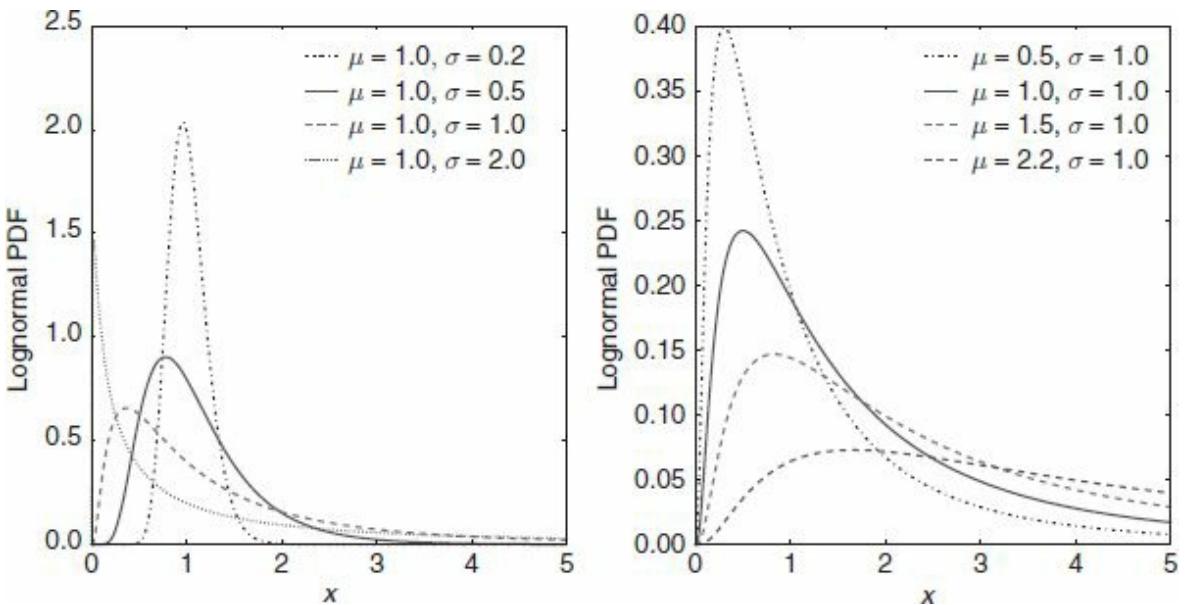


Figure 5.1 Left: Set of lognormal probability distribution functions with different values for the scatter parameter σ , centered at $\mu = 1.0$. Right: Set of lognormal probability distribution functions with different values for μ , and scatter parameter $\sigma = 1.0$

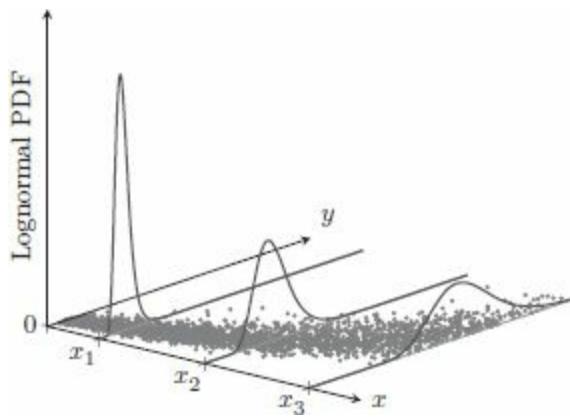


Figure 5.2 Illustration of lognormal-distributed data.

The lognormal model is based on a lognormal PDF, which log-transforms the response term y within the distribution itself. The actual data being modeled are left in their original form. The key is that the data being modeled are positive real values. Values equal to zero or negative values are not appropriate for the lognormal model. Because the normal distribution assumes the possibility of negative and zero values in addition to positive values, using a normal model on data that can only be positive violates the distributional assumptions underlying the normal model. To reiterate, it is statistically preferable to model positive continuous data using a lognormal model rather than a log- y -transformed normal model. A foremost advantage of using the lognormal is that the variance can vary with the mean. However, the constraint is that the ratio of the mean and the standard deviation in a lognormal model is constant. For a normal model, the variance is assumed to be constant throughout the entire range of model observations. Unfortunately the lognormal model has not always been available in standard commercial statistical software, so many researchers instead

have log-transformed y and estimated parameters using a normal model. This is no longer a problem, though.

Novice researchers get into trouble when they model count data using a lognormal model, or worse, when they log the response and model the data as normal. Count data should always be modeled using a so-termed count model, e.g., the Poisson, negative binomial, generalized Poisson, and similar models. Count models are especially designed to account for the distributional assumptions underlying count data. Count data is discrete, and should always be modeled using an appropriate count model ([O'Hara and Kotze, 2010](#)). With this caveat in mind, let us discuss the nature of lognormal models, providing examples using synthetic data. A synthetic lognormal model may be created in R using the following code.

Code 5.3 Synthetic lognormal data and model generated in R.

```
=====
require(gamlss)
# Data
set.seed(1056)                      # set seed to replicate example
nobs = 5000                          # number of observations in model
x1 <- runif(nobs)                   # random uniform variable
xb <- 2 + 3*x1                      # linear predictor, xb
y <- rlnorm(nobs, xb, sdlog=1)      # create y as random lognormal variate
summary(mylnm <- gamlss(y ~ x1, family=LOGNO))
=====
   Estimate Std.Error t value Pr(>|t|)
(Intercept) 1.99350    0.02816    70.78 <2e-16 ***
x1          3.00663    0.04868    61.76 <2e-16 ***
Sigma link function: log
Sigma coefficients:
(Intercept) -0.001669  0.010000   -0.167       0.867
```

This code has only one predictor, x_1 , with assigned value 3 and intercept 2. This provides us with the data set which we will use for developing a Bayesian lognormal model. Note that the `rlnorm` pseudo-random number generator creates lognormal values adjusted by the linear predictor that we specified. This data may be modeled using R's `glm` function with a normal family and log link or by using the maximum likelihood lognormal model found in the `gamlss` package. If an analyst wishes to model the data using a frequentist-based model, the `gamlss` model is preferred since it estimates a variance parameter in addition to coefficients. The `glm` function does not. Of course, our purpose is to model the data using a Bayesian model. Again, the y variable is lognormally distributed, as adjusted by the terms of the linear predictor and the log link function.

Keep in mind that, when constructing a synthetic model such as the above, the values placed in the predictors are coefficients, or slopes. That is what is being estimated when we are using a maximum-likelihood-based model. However, when we use the synthetic data for a Bayesian model, we are estimating posterior parameter means, not specifically coefficients. If diffuse or minimal-information priors (unfortunately usually referred to as non-informative priors, as mentioned earlier) are used with the Bayesian model, the results of the Bayesian model will usually be very close to the maximum likelihood parameter estimates. We have discussed why this is the case. It is a good test for determining whether our Bayesian code is correct. For real data applications, though, it is preferred to use meaningful priors if they are available. We shall continually discuss this subject as we progress through the book.

Again, it is important to emphasize that we are constructing Bayesian code with largely minimal-information priors to model synthetic data with preset parameter values, in order to demonstrate the appropriateness of the code. This procedure also allows us to describe the model generically, which makes it easier for the reader to use for his or her own applications. We will also demonstrate that the R program creating the lognormal function is correct and that the response y is in fact lognormally distributed. The output demonstrates that the synthetic data is structured as we specified.

For some distributions there exists R software that provides the analyst with the opportunity to employ a Bayesian model on the data. Specifically, the R package `MCMCpack` can be used to develop several different Bayesian models. When this is a possibility in this chapter, we shall create synthetic data, modeling it using the appropriate R function as well as our JAGS code. For some of the hierarchical models discussed in Chapter 8, the R `MCMCpack` package can be used to calculate posteriors as well as diagnostics. We will use the R package when possible for R models. Unfortunately, the `MCMCpack` package does not support Bayesian lognormal models or any of the continuous response models discussed in this section. We therefore shall proceed directly to the creation of a Bayesian lognormal model using JAGS, from within the R environment.

Lognormal Model in R using JAGS

We now shall run a JAGS lognormal algorithm on the data in Code 5.3, expecting to have values of the predictor parameters close to those we specified, i.e., an intercept parameter close to 2 and a coefficient parameter for x_1 close to 3. Unless we set a seed value, each time we run the code below the results will differ a bit. Estimation of parameters is obtained by sampling. Note the closeness of the JAGS and `gamlss` results.

Code 5.4 Lognormal model in R using JAGS.

```
=====
require(R2jags)
X <- model.matrix(~ 1 + x1)
K <- ncol(X)
model_data <- list(Y = y, X = X, K = K, N = nobs,
                     Zeros = rep(0, nobs))
LNORM <-
model{
  # Diffuse normal priors for predictors
  for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001) }

  # Uniform prior for standard deviation
  tau <- pow(sigma, -2)      # precision
  sigma ~ dunif(0, 100)       # standard deviation

  # Likelihood
  for (i in 1:N){
    Y[i] ~ dlnorm(mu[i],tau)
    mu[i] <- eta[i]
    eta[i] <- inprod(beta[], X[i,])
  }
}

inits <- function () { list(beta = rnorm(K, 0, 0.01)) }
params <- c("beta", "sigma")
LN <- jags(data = model_data,
            inits = inits,
            parameters = params,
            model = textConnection(LNORM),
            n.chains = 3,
            n.iter = 5000,
            n.thin = 1,
```

```

n.burnin = 2500)
print(LN, intervals=c(0.025, 0.975), digits=3)
=====
```

It should be noted that the time taken for a sampling run can be given with the `system.time` function. Type `?system.time` for help with this capability.

	mu.vect	sd.vect	2.5%	97.5%	Rhat	n.eff
beta[1]	1.992	0.029	1.934	2.048	1.009	260
beta[2]	3.009	0.051	2.911	3.111	1.007	360
sigma	0.999	0.010	0.979	1.019	1.001	7500
deviance	49162.668	2.570	49159.806	49169.332	1.000	1

To display the histograms (Figure 5.3) and trace plots (Figure 5.4) of the model parameters one can use:

```

source("CH-Figures.R")
out <- LN$BUGSoutput
MyBUGSHist(out,c(uNames("beta",K),"sigma"))
```

and

```
MyBUGSChains(out,c(uNames("beta",K),"sigma"))
```

Note that the `R2jags` `traceplot` function can also be used to produce trace plots. Type `traceplot(LN)` for this example.

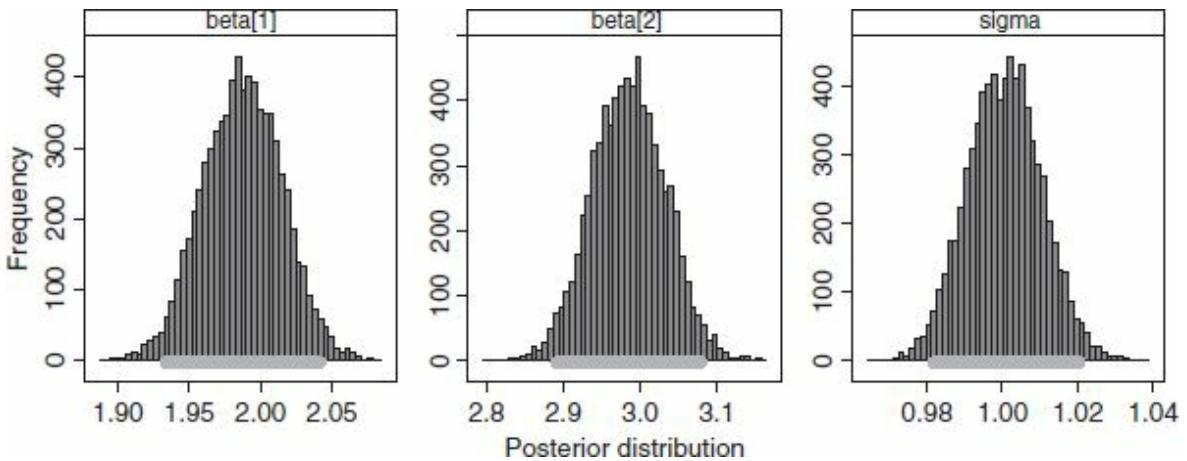


Figure 5.3 Histogram of the MCMC iterations for each parameter. The thick lines at the bases of the histograms represent the 95% credible intervals.

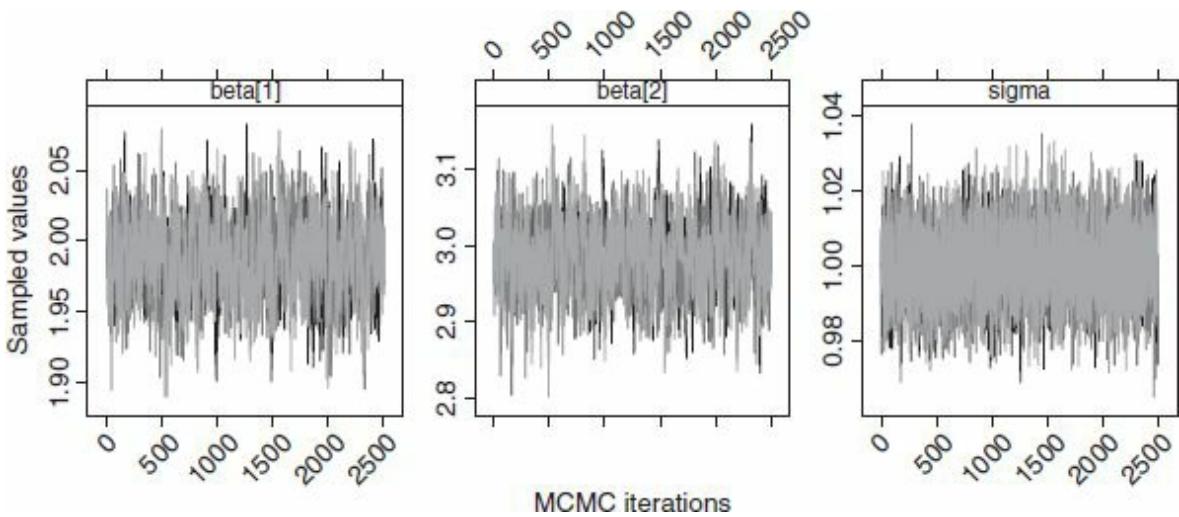


Figure 5.4 MCMC chains for the two parameters, β_1 and β_2 , and the standard deviation σ for the lognormal model.

If the data involves further parameters and observations then you will probably need to increase the burn-in and number of iterations. If there is also substantial autocorrelation in the data, we suggest thinning it, i.e., arranging for the sampling algorithm in JAGS to accept only every other sample (`n.thin=2`), or every third sample (`n.thin=3`), etc. The greater the thinning value, the longer the algorithm will take to converge to a solution, but the solution is likely to be more appropriate for the data being modeled. Note that the variance parameter of the lognormal model increases in proportion to the square of the mean, unlike for the normal model where the variance is constant throughout the range of values in the model. Figure 5.5 shows the fitted model in comparison with the synthetic data, together with 50% and 95% prediction intervals. The interpretation of the 95% prediction intervals is straightforward: there is a 95% probability that a new observation will fall within this region.

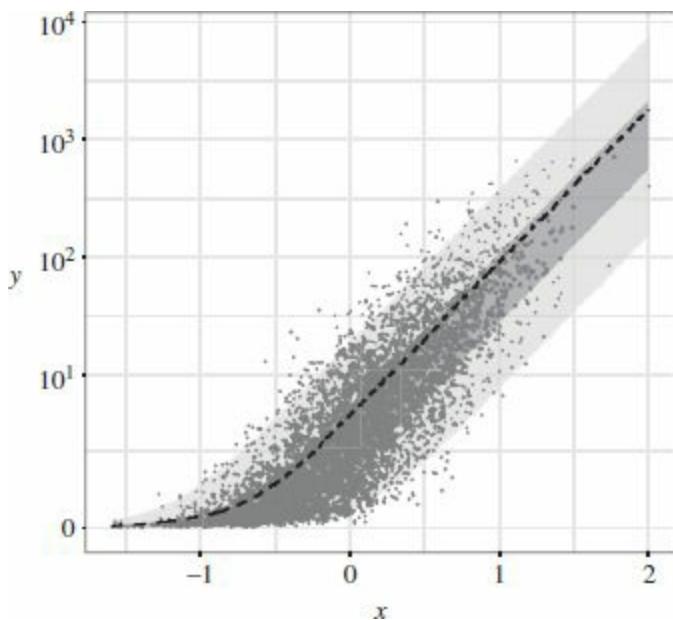


Figure 5.5 Visualization of the lognormal model. The dashed line and darker (lighter) shaded areas show the fitted model and 50% (95%) credible intervals, respectively. The dots represent the synthetic data points.

Lognormal Models in Python using Stan

In the following, we show how to construct a generic lognormal Bayesian model from Python, using Stan.

Code 5.5 Lognormal model in Python using Stan.

```
=====
import numpy as np
from scipy.stats import uniform, lognorm
import pystan

# Data
np.random.seed(1056)                                # set seed to replicate example
nobs = 5000                                         # number of obs in model
x1 = uniform.rvs(size=nobs)                         # random uniform variable

beta0 = 2.0                                           # intercept
beta1 = 3.0                                           # linear coefficient
sigma = 1.0                                           # dispersion

xb = beta0 + beta1 * x1                             # linear predictor
exb = np.exp(xb)
y = lognorm.rvs(sigma, scale=exb, size=nobs)        # create y as adjusted
                                                    # random normal variate

# Fit
mydata = {}
mydata['N'] = nobs
mydata['x1'] = x1
mydata['y'] = y

stan_lognormal = """
data{
    int<lower=0> N;
    vector[N] x1;
    vector[N] y;
}
parameters{
    real beta0;
    real beta1;
    real<lower=0> sigma;
}
transformed parameters{
    vector[N] mu;
    for (i in 1:N) mu[i] = beta0 + beta1 * x1[i];
}
model{
    y ~ lognormal(mu, sigma);
}
"""

# Fit
fit = pystan.stan(model_code=stan_lognormal, data=mydata, iter=5000, chains=3,
                    verbose=False, n_jobs=3)

# Output
nlines = 8                                         # number of lines in output
output = str(fit).split('\n')
for item in output[:nlines]:
    print(item)

=====
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
beta0	1.99	7.7e-4	0.03	1.93	1.97	1.99	2.01	2.04	1390.0	1.0
beta1	3.0	1.3e-3	0.05	2.9	2.96	3.0	3.03	3.09	1408.0	1.0
sigma	0.99	2.4e-4	9.9e-3	0.97	0.98	0.99	1.0	1.01	1678.0	1.0

The results of the Python model are very close to those for the JAGS model, as we expect. Both models properly estimate the means of the posterior distributions for each parameter in the synthetic model we specified at the outset of the section. Of course, if we provide an informative prior for any parameter then the mean value for the calculated posterior distribution will change. By providing diffuse priors the posteriors are based on the log-likelihood function, which means that they are based on the model data and not on any information external to it. As a consequence the Bayesian results are similar to those obtained using maximum likelihood estimation.

Finally, it is important to highlight that there is a difference between the lognormal parameterization in JAGS and Stan such that their dispersion parameters are related by

$$\sigma_{\text{Stan}} = \frac{1}{\sqrt{\tau_{\text{JAGS}}}}. \quad (5.9)$$

The results are consistent between Codes 5.4 and 5.5 because for a unitary dispersion $\tau_{\text{JAGS}} = \sigma_{\text{Stan}}$.

5.2.2 Bayesian Gamma Models

Two varieties of gamma model are typically used by analysts, whether in the domains of social science or in the physical sciences. These two are the canonical or inverse model and the log-gamma model. The canonical model is also referred to as the reciprocal or inverse gamma model. Both the log and inverse parameterizations are defined by the link function employed in the log-likelihood. We may add here that the identity link is sometimes used with the gamma family, but it is rare. The base probability distribution function for the gamma distribution is given by

$$f(y; \mu, \phi) = \frac{1}{y\Gamma(1/\phi)} \left(\frac{y}{\mu\phi}\right)^{1/\phi} e^{-y/\mu\phi}. \quad (5.10)$$

The canonical gamma log-likelihood, which is also sometimes referred to as the inverse gamma model, can be expressed as

$$\mathcal{L}(\mu, \phi; y) = \sum_{i=1}^n \left\{ \frac{1}{\phi} \ln \left(\frac{y_i}{\phi\mu_i} \right) - \left(\frac{y_i}{\phi\mu_i} \right) - \ln(y_i) - \ln \Gamma \left(\frac{1}{\phi} \right) \right\}. \quad (5.11)$$

Gamma models are used for positive-only continuous data, like the lognormal model. The link function of the canonical gamma model, however, dictates that the model is to be used when there is an indirect relationship between the fit and the linear predictor. For a simple economic example, suppose that the data to be modeled is set up to have a direct relationship with the mean miles-per-gallon used by automobiles. Using a canonical gamma model provides an interpretation of the

predictors based on the mean gallons-per-mile – the inverse of the manner in which the data was established. The same inverse relationship obtains with respect to any other application as well.

Owing to the distributional interpretation of the canonical gamma model, most analysts employ a log-gamma model for evaluating positive real data based on a gamma distribution. The value in modeling positive real data using a log-gamma model rather than a normal model is that the shape parameter varies with the mean, allowing substantially greater flexibility to the range of modeling. For the lognormal model the variance increases with the square of the mean, so it is limited to that relationship. However, with the log-gamma model, the analyst has two flexible parameters with which to model positive real data. Moreover the mean and fitted value are directly related, unlike in the canonical gamma. Figure 5.6 shows examples of the gamma PDF for different parameter values and Figure 5.7 shows an example of gamma-distributed data.

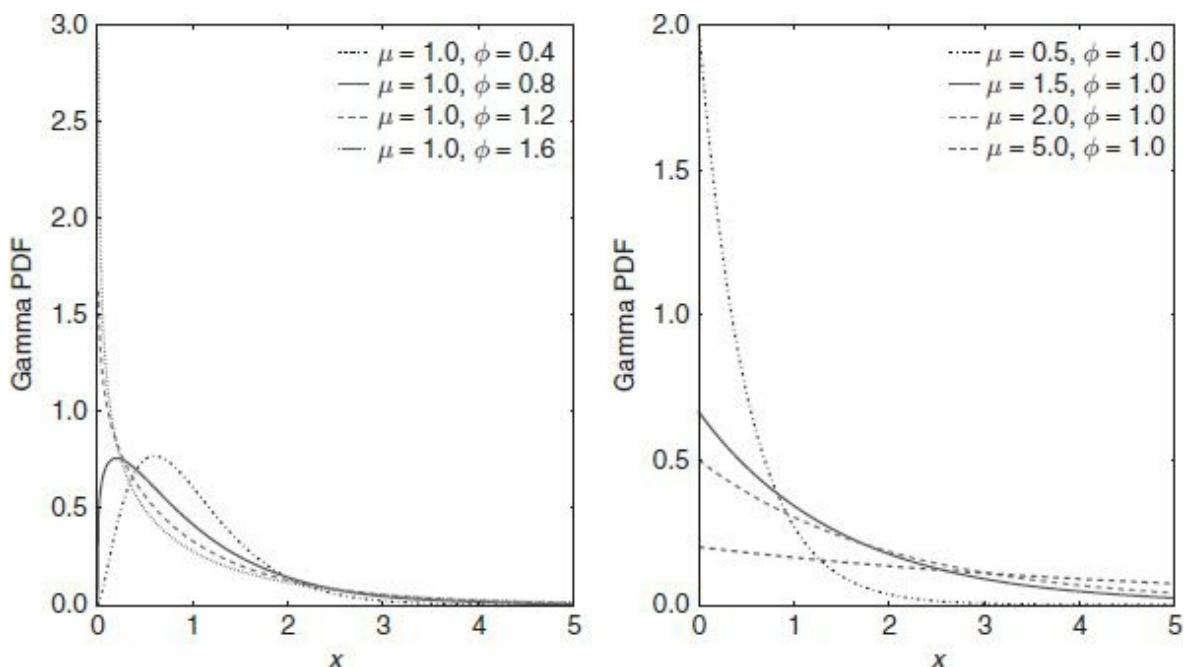


Figure 5.6 Left: Set of gamma probability distribution functions with different values for the parameter ϕ ; $\mu = 1$. Right: Set of gamma probability distribution functions with different values for μ ; $\phi = 1.0$.

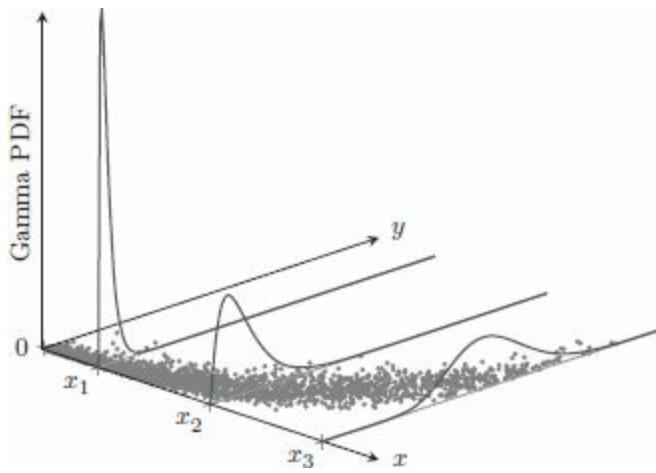


Figure 5.7 Illustration of gamma-distributed data.

The form that, in JAGS, can be used as the log-likelihood for a gamma Bayesian model is as follows:

```
mu <- exp(xb) # xb = the linear predictor
LL <- (1/phi)* log(y/(phi*mu)) - (y/(phi*mu)) - log(y) - lngamma(1/phi)
```

Notice that it is the same as for the inverse or canonical linked gamma, except that `mu` is defined as `exp(xb)`.

Code to create synthetic log-gamma data with specific values for the predictor parameters is based on specifying terms in a linear predictor and submitting it to the `rgamma` function. The logic of the `rgamma` function is

```
rgamma(n, shape, rate, scale = 1/rate)
```

We provide a rate value of `exp(xb)` or a scale of `1/exp(xb)`, where `xb` is the linear predictor. Thesynthetic data below specifies values for the intercept coefficient parameters, and shape parameter `r`.

Code 5.6 Log-gamma synthetic data generated in R.

```
=====
set.seed(33559)
nobs <- 1000
r <- 20          # shape
beta1 <- 1
beta2 <- 0.66
beta3 <- -1.25
x1 <- runif(nobs)
x2 <- runif(nobs)
xb <- beta1 + beta2*x1+beta3*x2
exb <- exp(xb)
py <- rgamma(nobs,shape = r, rate= r/exb)
LG <- data.frame(py, x1, x2)
=====
```

Log-Gamma Models in R using JAGS

The JAGS code for a Bayesian log-gamma model given the data in `LG` is given below.

Code 5.7 Log-gamma model in R using JAGS.

```
=====
library(R2jags)

X <- model.matrix(~ x1 + x2, data = LG)           # number of columns
K <- ncol(X)

model.data <- list(Y = LG$py,                      # response
                    X = X,                         # covariates
                    N = nrow(LG),                  # sample size
                    b0 = rep(0,K),
                    B0 = diag(0.0001, K))

sink("LGAMMA.txt")
cat("
model{
  # Diffuse priors for model betas
  beta ~ dmnorm(b0[], B0[,])

  # Diffuse prior for shape parameter
  r ~ dgamma(0.01, 0.01)

  # Likelihood
  for (i in 1:N){
    Y[i] ~ dgamma(r, lambda[i])
    lambda[i] <- r / mu[i]
    log(mu[i]) <- eta[i]
    eta[i] <- inprod(beta[], X[i,])
  }
}

", fill = TRUE)
sink()

inits <- function () {
  list(
    beta = rnorm(K,0,0.01),
    r = 1 )
}
params <- c("beta", "r")

# JAGS MCMC
LGAM <- jags(data      = model.data,
               inits     = inits,
               parameters = params,
               model.file = "LGAMMA.txt",
               n.thin    = 1,
               n.chains  = 3,
               n.burnin  = 3000,
               n.iter    = 5000)
print(LGAM, intervals=c(0.025, 0.975), digits=3)
=====

      mu.vect   sd.vect    2.5%   97.5%   Rhat   n.eff
beta[1]     1.007    0.020    0.969    1.047   1.009    260
beta[2]     0.662    0.025    0.613    0.711   1.007    430
beta[3]    -1.253    0.027   -1.302   -1.200   1.012    200
r          19.660    0.889   17.992   21.448   1.001   7500
deviance   1216.043   2.819  1212.406  1222.851   1.003   2400

pD = 4.0 and DIC = 1220.0
```

Even with only 2500 samples used to determine the parameter values, the results are close to what we specified. The above generic example of a log-gamma model in JAGS can be expanded to handle a wide variety of astronomical data situations as well as applications from almost any other discipline.

Log-Gamma Models in Python using Stan

Code 5.8 Log-gamma model in Python using Stan.

```

output = str(fit).split('\n')
for item in output[:nlines]:
    print(item)
=====
      mean se_mean   sd  2.5%   25%   50%   75% 97.5% n_eff Rhat
beta0  1.05  1.4e-3 0.04  0.97  1.02  1.05  1.07  1.12 647.0  1.0
beta1  0.6   1.8e-3 0.05  0.51  0.57  0.6   0.63  0.69 663.0  1.0
beta2 -1.28  1.8e-3 0.05 -1.38 -1.31 -1.28 -1.25 -1.19 692.0  1.0
r     1.83  1.6e-3 0.04  1.75  1.8   1.83  1.86  1.91 733.0  1.0

```

5.2.3 Bayesian Inverse Gaussian Models

The inverse Gaussian distribution and inverse Gaussian regression are seldom used by analysts for research, although they have been used with success in transportation. However, simulation studies ([Hilbe, 2014](#)) have demonstrated that this distribution is best for modeling data in which the greatest density of values is in the very low numbers. The inverse Gaussian distribution is a continuous distribution, having two parameters, given by

$$f(y; \mu, \lambda) = \sqrt{\frac{\lambda}{2\pi y^3}} e^{-\lambda(y-\mu)^2/(2\mu^2y)}. \quad (5.12)$$

The log-likelihood of the canonical inverse Gaussian distribution, with inverse quadratic link $(1/\mu^2)$, is

$$\mathcal{L}(\mu, \sigma^2; y) = \sum_{i=1}^n \left\{ \frac{1}{2} \left(\frac{\lambda(y_i - \mu_i)^2}{\mu^2 y_i} \right) - \frac{1}{2} \ln(\pi y_i^3) + \frac{1}{2} \ln(\lambda) \right\}. \quad (5.13)$$

The inverse Gaussian evaluator is similar to the gamma and others that we have discussed. The difference is in how the log-likelihood is defined. In pseudocode the log-likelihood for a canonical inverse Gaussian model is given as follows:

```

mu <- 1/sqrt(xb)
LL <- -.5*(y-mu)^2/(sig2*(mu^2)*y) -.5*log(pi*y^3*sig2)

```

Figure 5.8 shows different shapes of inverse Gaussian PDFs for various different values of μ and σ .

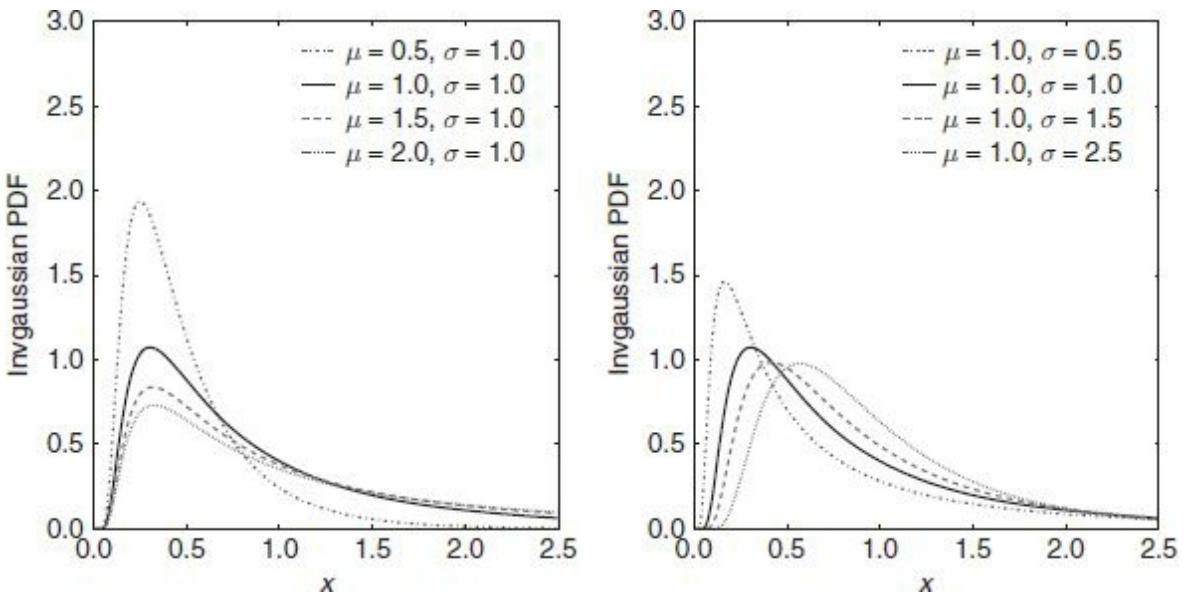


Figure 5.8 Left: Set of inverse Gaussian probability distribution functions with different values for the parameter μ ; $\sigma = 1$. Right: Set of inverse Gaussian probability distribution functions with different values for σ ; $\mu = 1.0$.

Log Inverse Gaussian in R using JAGS

One of the main reasons for using a log-gamma model in preference to a canonical gamma model is ease of interpretation. This is even more the case when the log-gamma model is compared with the canonical inverse Gaussian model. The relationship of the data, or linear predictor, and the fit is inverse quadratic. Interpretation is difficult. With the log-inverse-Gaussian model, though, the fitted value is the exponentiation of the linear predictor. There is an inverse Gaussian random number generator in the R `statmod` package on CRAN. We will now use it to create a synthetic log-inverse-Gaussian model.

We will run a JAGS model from within R, using `R2jags` afterwards.

Code 5.9 Log-inverse-Gaussian data.

```
=====
require(statmod)
set.seed(1056)          # set seed to replicate example
nobs = 3000              # number of obs in model
x1 <- runif(nobs)        # random uniform variable
xb <- 1 + 0.5*x1
mu <- exp(xb)
y <- rinvgauss(nobs, mu, 20)    # create y as adjusted random inverse-gaussian
variate
=====
```

The synthetic inverse Gaussian data is now in the `y` variable, as adjusted by `x1`, which is a random uniform variate. The JAGS code below, without truly informative priors, should result in a Bayesian model with roughly the same values for the parameters. Notice should be taken of the prior put on the variance. We use a half-Cauchy(25) distribution, which appears to be optimal for a diffuse prior on the variance function. JAGS does not have a built-in Cauchy or half-Cauchy function, but such a function can be obtained by dividing the normal distributions, $N(0, 625)$ by

$N(0, 1)$. The absolute value of this ratio follows the half-Cauchy(25) distribution. Given that JAGS employs the precision rather than the variance in its normal distribution function, the numerator will appear as `dnorm(0, 0.0016)`, where the precision is 1/625. In the code below we call the half-Cauchy distribution lambda, which is the diffuse prior we put on the inverse Gaussian variance (see [Zuur, Hilbe, and Ieno, 2013](#), for an extended explanation). Other diffuse priors put on the variance are `dgamma(0.001, 0.001)`, which represents a gamma prior distribution with mean 1 and variance 1000 and `dunif(0.001, 10)`. Checking the mixing and comparative summary statistics such as DIC and pD when using different priors is wise if you suspect convergence problems. Even if you do not, selecting an alternative prior may result in a better-fitted model. These sorts of sensitivity tests should be performed as a matter of routine.

Code 5.10 Log-inverse-Gaussian model in R using JAGS.

```
=====
require(R2jags)
X <- model.matrix(~ 1 + x1)
K <- ncol(X)

model.data <- list(Y = y,                      # response
                    X = X,                      # covariates
                    N = nobs,                   # sample size
                    K = K,                      # number of betas
                    Zeros = rep(0, nobs))
sink("IGGLM.txt")

cat("
model{
  # Diffuse normal priors betas
  for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001)}

  # Prior for lambda parameter of inverse Gaussian distribution
  num ~ dnorm(0, 0.0016)          # <---half-Cauchy(25)
  denom ~ dnorm(0, 1)             # <---half-Cauchy(25)
  lambda <- abs(num / denom)     # <---half-Cauchy(25)

  # Likelihood
  C <- 10000
  for (i in 1:N){
    Zeros[i] ~ dpois(Zeros.mean[i])
    Zeros.mean[i] <- -L[i] + C

    l1[i] <- 0.5 * (log(lambda) - log(2 * 3.141593 * Y[i]^3))
    l2[i] <- -lambda * (Y[i] - mu[i])^2 / (2 * mu[i]^2 * Y[i])
    L[i] <- l1[i] + l2[i]

    log(mu[i]) <- inprod(beta[], X[i,])
  }
  ",fill = TRUE)
sink()

inits <- function () {
  list(
    beta  = rnorm(ncol(X), 0, 0.1),
    num   = rnorm(1, 0, 25),
    denom = rnorm(1, 0, 1)  )
}

params <- c("beta", "lambda")

IVG <- jags(data = model.data,
             inits = inits,
             parameters = params,
             model = "IGGLM.txt",
             n.thin = 1,
             n.chains = 3,
             n.burnin = 2500,
             n.iter = 5000)
print(IVG, intervals=c(0.025, 0.975), digits=3)
=====

      mu.vect  sd.vect      2.5%      97.5%   Rhat  n.eff
beta[1]      0.974  0.025      0.926     1.023  1.006    410
beta[2]      0.539  0.045      0.451     0.629  1.005    480
```

```

lambda      19.846  0.885    18.168    21.609  1.001    7500
deviance   20003356.068  2.485  20003353.316  20003362.564  1.000      1
pD = 3.1 and DIC = 20003359.2

```

These values are nearly identical to those we specified in generating the model data. Note that the variance parameter was specified in the `rinvgauss` function to be 20. The JAGS result is 19.85 ± 0.885 . The log-inverse-Gaussian model provides even more flexibility in the modeling space than the log-gamma model. It is most appropriate for modeling data that is heavily peaked at lower values, with a long right skew of data.

Log Inverse Gaussian in Python using Stan

Here we face for the first time a situation where the desired distribution is not available within Stan. Thus we need to explicitly write the log-likelihood function as in Section 4.1.4.

Code 5.11 Inverse Gaussian model in Python using Stan.

```

=====
import numpy as np
import statsmodels.api as sm
import pystan
from scipy.stats import uniform, invgauss

# Data
np.random.seed(1056)                      # set seed to replicate example
nobs = 1000                                 # number of obs in model
x1 = uniform.rvs(size=nobs)                 # random uniform variable

beta0 = 1.0
beta1 = 0.5
l1 = 20

xb = beta0 + beta1 * x1                     # linear predictor
exb = np.exp(xb)

y = invgauss.rvs(exb/l1, scale=l1)          # create response variable

# Fit
stan_data = {}                                # build data dictionary
stan_data['Y'] = y                            # response variable
stan_data['x1'] = x1                          # explanatory variable
stan_data['N'] = nobs                         # sample size

# Stan code
stan_code = """
data{
    int<lower=0> N;
    vector[N] Y;
    vector[N] x1;
}
parameters{
    real beta0;
    real beta1;
    real<lower=0> lambda;
}
transformed parameters{
    vector[N] exb;
    vector[N] xb;

    for (i in 1:N) xb[i] = beta0 + beta1 * x1[i];
    for (i in 1:N) exb[i] = exp(xb[i]);
}
model{
    real l1;
    real l2;
    vector[N] loglike;

    beta0 ~ normal(0, 100);
    beta1 ~ normal(0, 100);
}
```

```

lambda ~ uniform(0.0001, 100);

for (i in 1:N){
  l1 = 0.5 * (log(lambda) - log(2 * pi() * pow(Y[i], 3)));
  l2 = -lambda*pow(Y[i] - exb[i], 2)/(2 * pow(exb[i], 2) * Y[i]);
  loglike[i] = l1 + l2;
}
target += loglike;
}

fit = pystan.stan(model_code=stan_code, data=stan_data, iter=5000, chains=3,
                   warmup=2500, n_jobs=3)

# Output
nlines = 8 # number of lines in screen output

output = str(fit).split('\n')
for item in output[:nlines]:
  print(item)
=====

      mean   se_mean     sd    2.5%    25%    50%    75%   97.5%   n_eff   Rhat
beta0    0.99   6.7e-4   0.03    0.94    0.97    0.99    1.01    1.04   1470.0   1.0
beta1    0.52   1.2e-3   0.05    0.43    0.49    0.52    0.55    0.61   1504.0   1.0
lambda   20.26   0.02    0.89   18.54   19.65   20.24   20.86   22.05   1623.0   1.0

```

Note the closeness of the parameter means, standard deviation, and credible intervals to the fiducial values and JAGS output.

5.2.4 Bayesian Beta Model

The beta model is used to model a response variable that is formatted as a proportion x between the values of 0 and 1. That is, the range of values for a beta model is $0 < x < 1$.

The beta PDF can be parameterized either with two parameters, a and b , or in terms of the mean, μ . The standard beta PDF is given as

$$f(y; a, b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} y^{a-1} (1-y)^{b-1}, \quad 0 < y < 1. \quad (5.14)$$

The mean and variance of the beta distribution are defined as

$$E(y) = \frac{a}{a+b} = \mu, \quad V(y) = \frac{ab}{(a+b)^2(a+b+1)}, \quad (5.15)$$

and the log-likelihood function is given as

$$\mathcal{L}(a, b; y) = \sum_{i=1}^n \{\log \Gamma(a+b) - \log \Gamma(a) - \log \Gamma(b) + (a-1) \log y_i + (b-1) \log(1-y_i)\}. \quad (5.16)$$

We show in Figure 5.9 examples of the beta PDF for different values of the a and b parameters,

and in Figure 5.10 an illustration of beta-distributed data.

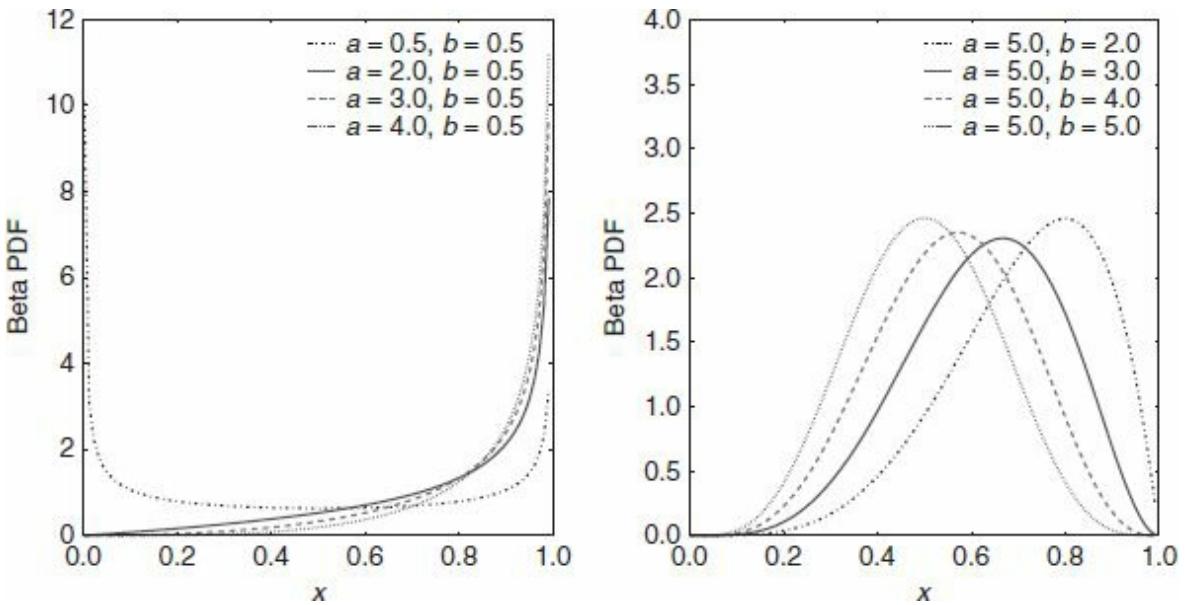


Figure 5.9 Left: Set of beta probability distribution functions with different values for the parameter a ; $b = 0.5$. Right: Set of beta probability distribution functions with different values for b ; $a = 5.0$.

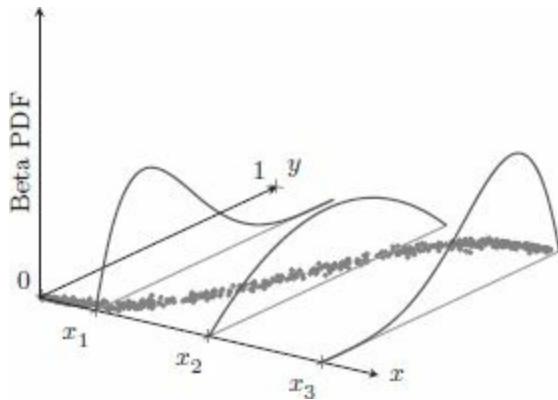


Figure 5.10 Beta-distributed data.

We can create a synthetic beta model in R using the `rbeta` pseudo-random number generator.

Code 5.12 Synthetic beta-distributed data generated in R.

```
=====
set.seed(1056)          # set seed to replicate example
nobs <- 1000            # number of obs in model
x1 <- runif(nobs)       # random normal variable
xb <- 0.3+1.5*x1       # linear predictor
exb <- exp(-xb)         # prob of 0
p <- exb/(1+exb)        # prob of 1
theta <- 15
y <- rbeta(nobs,theta*(1-p),theta*p)
```

As an example of synthetic beta data, we executed the code in the above table. To determine the range of values in y , we used a summary function and checked a histogram (Figure 5.11) of the beta-distributed variable:

```
> summary(y)
  Min. 1st Qu. Median   Mean 3rd Qu.   Max.
0.2620  0.6494  0.7526  0.7346  0.8299  0.9856
> hist(y)
```

Note that with the median slightly greater than the mean we expect that the distribution will show a slight left skew.

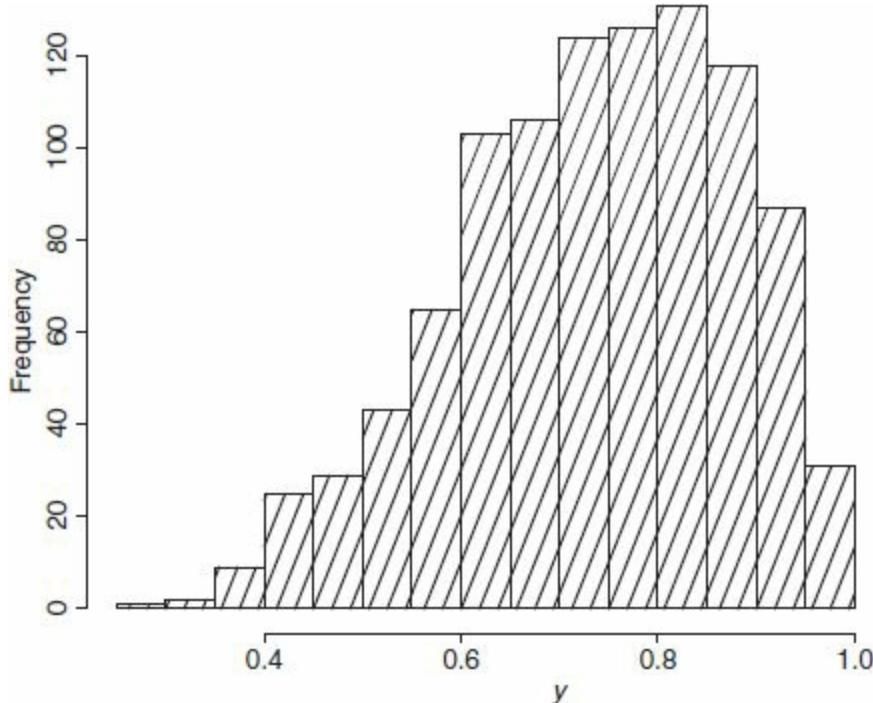


Figure 5.11 Histogram of the beta-distributed data y generated in the code snippet 5.12.

The beta distribution has a wide variety of shapes, as seen in Figure 5.9, which makes it an ideal prior distribution with binomial models. We shall discuss the beta prior in some detail when we address priors later in the book.

The model is estimated using the default logit link for the mean and the identity link for the scale parameter, ϕ . At times you will find that a beta model has a log link for the scale parameter. The coefficient parameters of the model are identical whether an identity or log link is used for the scale; only the ϕ statistic changes.

Beta Model in R using JAGS

A Bayesian beta model using JAGS is shown below. We employ the identity link for the scale, which

is called `theta` for this model. The same beta synthetic data is used for estimation.

Code 5.13 Beta model in R using JAGS.

```
=====
require(R2jags)

# Data
# define model parameters
set.seed(1056)          # set seed to replicate example
nobs<-1000               # number of obs in model
x1 <- runif(nobs)         # random normal variable

# generate toy data
xb <- 0.3+1.5*x1
exb <- exp(-xb)
p <- exb/(1+exb)
theta <- 15
y <- rbeta(nobs,theta*(1-p),theta*p)

# construct data dictionary
X <- model.matrix(~1+x1)
K <- ncol(X)
model.data <- list(Y = y,           # response variable
                     X = X,           # predictor matrix
                     K = K,           # number of predictors including the
                                     # intercept
                     N = nobs,        # sample size
                     )
Beta<-"model{
  # Diffuse normal priors for predictors
  for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001) }

  # Gamma prior for precision parameter
  theta~dgamma(0.01,0.01)

  # Likelihood function
  for(i in 1:N){
    Y[i] ~ dbeta(shape1[i],shape2[i])
    shape1[i]<-theta*pi[i]
    shape2[i]<-theta*(1-pi[i])
    logit(pi[i]) <- eta[i]
    eta[i]<-inprod(beta[],X[i,])
  }
}"
# A function to generate initial values for mcmc
inits <- function () { list(beta = rnorm(ncol(X), 0, 0.1)) }

# Parameters to be displayed in output
params <- c("beta","theta")

BETAFit<- jags(data      = model.data ,
                 inits     = inits,
                 parameters = params,
                 model      = textConnection(Beta),
                 n.thin    = 1,
                 n.chains   = 3,
                 n.burnin   = 2500,
                 n.iter     = 5000)
print(BETAFit,justify="left", digits=2)
=====

  mu.vect sd.vect 2.5%   25%   50%   75% 97.5% Rhat n.eff
beta[1]     0.32   0.03   0.26   0.30   0.32   0.35   0.39   1   1000
beta[2]     1.50   0.06   1.38   1.46   1.50   1.54   1.62   1   1200
theta       15.64  0.68  14.30  15.18  15.64  16.09  16.99   1   2600
deviance -1782.36  2.45 -1785.13 -1784.13 -1783.00 -1781.31 -1775.95   1   2500

pD = 3.0 and DIC = -1779.4
```

The output provides the parameter values that we expect. A maximum likelihood beta model can be estimated on this data using the `betareg` function found in the `betareg` package on CRAN.

```
require(betareg)
betareg(y~x1)
```

```

Call:
betareg(formula = y ~ x1)

Coefficients (mean model with logit link):
(Intercept)      x1
  0.2716     1.5442

Phi coefficients (precision model with identity link):
(phi)
15.21

```

The parameter values calculated using JAGS very closely approximate the coefficients, intercept, and scale parameter values of the `betareg` function, thus confirming the JAGS model. Of course, we deliberately did not add informative priors to the Bayesian model, as this would alter the parameters. How much they would differ depends on the number of observations in the data as well as the values of the hyperparameters defining the priors. Remember that the real power of Bayesian modeling rests in its ability to have informative priors that can be used to adjust the model on the basis of information external to the original model data.

Figure 5.12 provides a graphic of fitted `beta` values and prediction intervals for the synthetic beta model from Code 5.13.

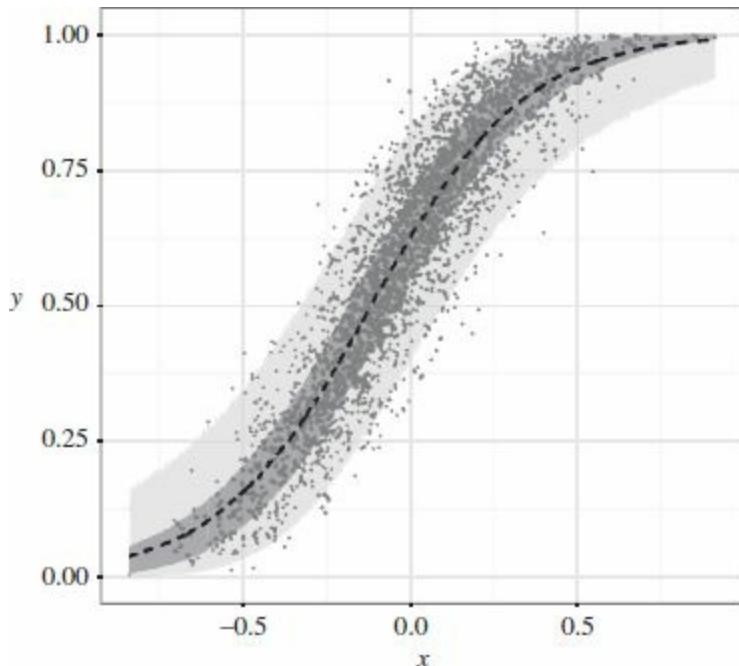


Figure 5.12 Visualization of synthetic data generated from a beta distribution. The dashed line and shaded areas show the fitted curve and the 50% and 95% prediction intervals. The dots correspond to the synthetic data.

Beta Model in Python using Stan

In the following we show how to implement the model described in Code 5.13 in Python using Stan:

Code 5.14 Beta model in Python using Stan.

```
=====
import numpy as np
import statsmodels.api as sm
import pystan

from scipy.stats import uniform
from scipy.stats import beta as beta_dist

# Data
np.random.seed(1056)                      # set seed to replicate example
nobs = 2000                                  # number of obs in model
x1 = uniform.rvs(size=nobs)                  # random uniform variable

beta0 = 0.3
beta1 = 1.5
xb = beta0 + beta1 * x1
exb = np.exp(-xb)
p = exb / (1 + exb)
theta = 15

y = beta_dist.rvs(theta * (1 - p), theta * p)  # create y as adjusted

# Fit
mydata = {}                                    # build data dictionary
mydata['N'] = nobs                           # sample size
mydata['x1'] = x1                            # predictors
mydata['y'] = y                             # response variable

stan_code = """
data{
    int<lower=0> N;
    vector[N] x1;
    vector<lower=0, upper=1>[N] y;
}
parameters{
    real beta0;
    real beta1;
    real<lower=0> theta;
}
model{
    vector[N] eta;
    vector[N] p;
    vector[N] shape1;
    vector[N] shape2;

    for (i in 1:N){
        eta[i] = beta0 + beta1 * x1[i];
        p[i] = inv_logit(eta[i]);
        shape1[i] = theta * p[i];
        shape2[i] = theta * (1 - p[i]);
    }
    y ~ beta(shape1, shape2);
}
"""

# Run mcmc
fit = pystan.stan(model_code=stan_code, data=mydata, iter=5000, chains=3,
                   warmup=2500, n_jobs=3)

# Output
print fit
=====

      mean   se_mean     sd    2.5%     25%     50%     75%   97.5%   n_eff Rhat
beta0    0.34   6.3e-4   0.02    0.29     0.32     0.34     0.35    0.38   1426.0   1.0
beta1    1.43   1.2e-3   0.04    1.34     1.4       1.43     1.46    1.52   1417.0   1.0
theta    15.1     0.01   0.48   14.2     14.78    15.09    15.42   16.04   1578.0   1.0
```

```
lp_ _ 1709.3    0.03  1.22  1706.1  1708.7  1709.6  1710.2  1710.7  1326.0  1.0
```

Notice that, in this example, we skipped the `transformed parameters` block and defined all intermediate steps in the `model` block. As a consequence, the code does not track the evolution of the parameters `eta` and `pi` or the shape parameters. The reader should also be aware that when parameters are defined in the `transformed parameter` block the declared constraints are checked every time the log-posterior is calculated; this does not happen when variables are declared in the `model` block. Thus, this approach should be avoided for non-trivial parameter constraints.

5.3 Bayesian Binomial Models

Binomial models have a number of very useful properties. Perhaps the foremost characteristic of a binomial model is that the fitted or predicted value is a probability. A second characteristic, related to the first, is that the response term to be modeled is a binary variable. It is assumed that the values of the binary response are 0 and 1. In fact, the software dealing with the estimation of binomial parameters assumes that the variable being modeled has values of only 0 or 1. If the data being modeled is cast as 1, 2 for example, the software converts it to 0 and 1 prior to estimation. Some software will simply not accept any values other than 0 and 1. In fact, we recommend formatting all binary variables in a statistical model, whether response variable or predictor, as 0, 1.

There are two major parameterizations of the binomial probability distribution, as well as corresponding models based on each parameterization. The most commonly used parameterization is based on the Bernoulli PDF, which is a subset of the full binomial PDF. The binomial distribution can be expressed as

$$f(y; p, m) = \binom{m}{y} p^y (1 - p)^{m-y}, \quad (5.17)$$

where

$$\binom{m}{y} = \frac{m!}{y! (m - y)!} \quad (5.18)$$

is the binomial normalization term, y is the response term and binomial numerator, m is the binomial denominator, and p represents the probability that y has the value 1. Thus $y = 1$ indicates success, usually thought of as success in obtaining whatever the model is testing for; $y = 0$ indicates a lack of success, or failure. A binary response such as this does not allow intermediate values, or values less than 0 or over 1. A more detailed examination of the full binomial PDF and model is provided in the section on grouped logistic or binomial models later in this chapter. Figure 5.13 shows examples of binomial distributions with different probabilities p and numbers of trials $n \equiv m$.

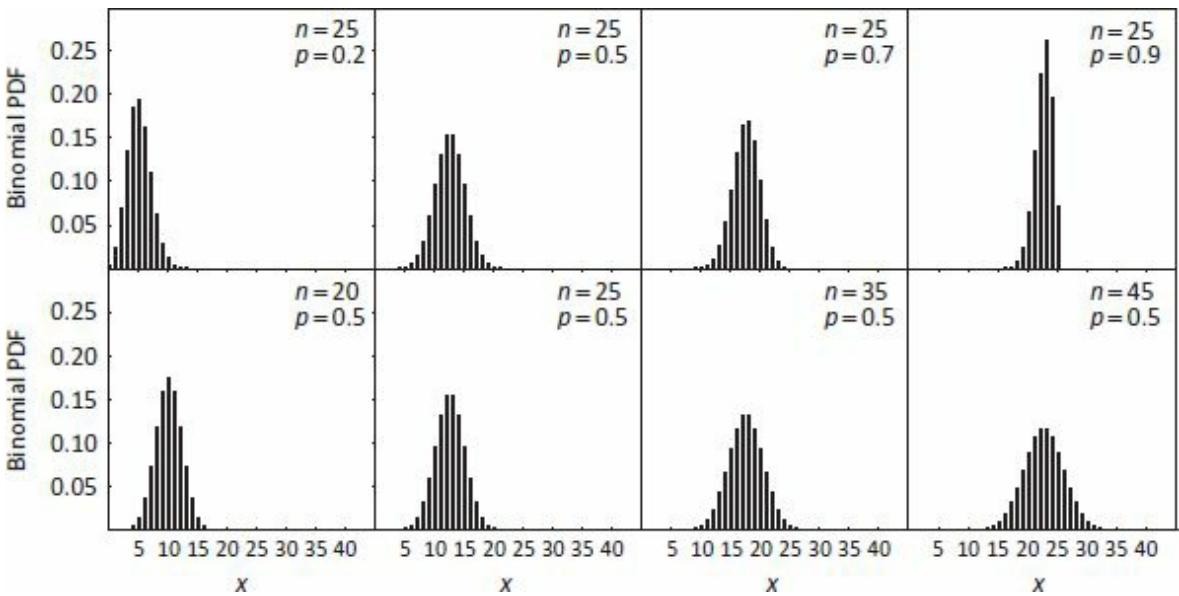


Figure 5.13 Upper: Set of binomial probability distribution functions with different values for the parameter p ; $n = 25$. Lower: Set of binomial probability distribution functions with different values for n ; $p = 0.5$.

The Bernoulli PDF sets m to the value 1, which eliminates the choose function; the choose function serves as the normalization term, ensuring that the individual probabilities sum to 1. The Bernoulli PDF is therefore expressed as

$$f(y; p) = p^y(1 - p)^{1-y}. \quad (5.19)$$

Note that frequently π or μ is used in place of p to symbolize the probability that $y = 1$.

5.3.1 Bayesian Bernoulli Logit Models

The Bernoulli logit model is nearly always referred to as simply the logistic model or logit model. The response term being modeled is binary, $y \in \{0, 1\}$. The predictors may be continuous, binary, or categorical.

Binary data is most appropriately modeled on the basis of the Bernoulli probability distribution. The logit link is the canonical or natural link for the Bernoulli distribution as well as for the more general binomial and beta binomial distributions, which we shall discuss later. Other link functions may be used as well. The most popular alternative is the probit link, which is theoretically based on a dichotomized normally distributed variable. We shall discuss the probit model in the next section. Other common links are the complementary loglog (cloglog) and loglog links. All these links constrain the predicted probability of the response variable to be between 0 and 1. The logit and probit links are symmetric functions, and the cloglog and loglog links are asymmetric (see e.g. [Hilbe, 2015](#)).

The Bernoulli log-likelihood function is used to describe the binary data being modeled. Prior

distributions are multiplied with the log-likelihood to construct a posterior distribution for each parameter in the model. For the Bayesian logistic model the only parameters estimated are the intercept and predictor posteriors. The Bernoulli log-likelihood function in its full form may be expressed as

$$\mathcal{L}(p, y) = \sum_{i=1}^n \left\{ y_i \ln \left(\frac{p_i}{1 - p_i} \right) + \ln(1 - p_i) \right\}. \quad (5.20)$$

This form of distribution is called the exponential-family form, as discussed earlier. As previously mentioned, the logistic parameterization is the canonical or natural form. The logistic link function is therefore $\ln(p/(1-p))$. The first derivative of the cumulant term $\ln(1-p)$ with respect to the link is the mean, p . The second derivative is the variance, $p(1-p)$. Figure 5.14 displays an illustration of Bernoulli-distributed data. The dots represent the synthetic data, and the gray line the probability of success as a function of x .

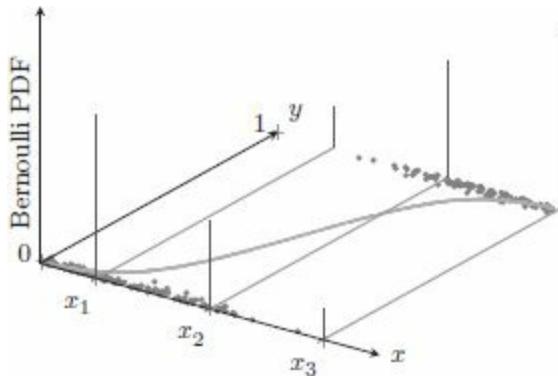


Figure 5.14 Illustration of Bernoulli-distributed data.

Bayesian Logistic Model using R

For this model we shall create a synthetic logistic model using two predictors, x_1 as a binary predictor and x_2 as a random uniform continuous variable. We give x_1 a coefficient (or parameter mean in Bayesian terminology) of **0.75** and x_2 a coefficient (mean) of **-5**. The intercept parameter is specified as 2. We use the `rbinom` function to produce the random Bernoulli response variable using the values assigned to model predictors. Note that the option `size` is given a value of 1. This specifies that the binomial denominator has a value of 1, i.e., that the distribution is in fact Bernoulli. When the data are grouped as in Section 5.3.3 the `size` option will be given a different value. The code below sets up the data set, consisting of **5000** observations. Note that on average 60% of the values in x_1 are 1 and 40% are 0. The synthetic data is saved in a data frame named `logitmod`.

Code 5.15 Synthetic data from logistic model in R.

```
=====
set.seed(13979)
nobs <- 5000
```

```

x1 <- rbinom(nobs, size = 1, 0.6)
x2 <- runif(nobs)
xb <- 2 + 0.75*x1 - 5*x2
exb <- 1/(1+exp(-xb))
by <- rbinom(nobs, size = 1, prob = exb)
logitmod <- data.frame(by, x1, x2)
=====
```

We use the `MCMClogit` function, to be found in the package `MCMCpack` on CRAN, to model `logitmod` data. As the name of the function implies, it uses MCMC sampling to determine the mean, standard deviation, and credible intervals of the model parameters. The code within the single lines produces a summary of the parameter statistics plus plots of the trace and density. The trace plot should have no deviations from the random line in progressing from the left to the right side of the box. For the density plot, generally one sees a normal bell-shaped curve for the posterior distributions of predictors. Skewed – even highly skewed – curves are sometimes displayed for the variance and scale parameters. The mean of each parameter is at the mode of the respective density plot. The median of the posterior is also sometimes used as the reported summary statistic for the distribution. It should be noted that, when the log-likelihood, AIC, and BIC are given sampling distributions, these are nearly always highly skewed.

Code 5.16 Logistic model using R.

```

=====
library(MCMCpack)
myL <- MCMClogit(by ~ x1 + x2,
                  burnin = 5000,
                  mcmc = 10000,
                  data = logitmod)
summary(myL)

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:
   Mean      SD    Naive SE Time-series SE
(Intercept) 1.9408  0.08314  0.0008314  0.002651
x1          0.8506  0.07183  0.0007183  0.002356
x2         -5.0118  0.14463  0.0014463  0.004709

2. Quantiles for each variable:
   2.5%     25%     50%     75%    97.5%
(Intercept) 1.7772  1.884   1.943   1.9970  2.096
x1          0.7116  0.803   0.850   0.8963  1.000
x2         -5.2904 -5.109  -5.016  -4.9131 -4.719
=====
```

The parameter values for the intercept, `x1`, and `x2` are close to what we specified. The trace plot looks good and the density plots are as expected. The credible intervals, together with the trace and density plots in Figure 5.15, appear to confirm that the model is well fitted.

```
> plot(myL) # Produces Figure 5.15
```

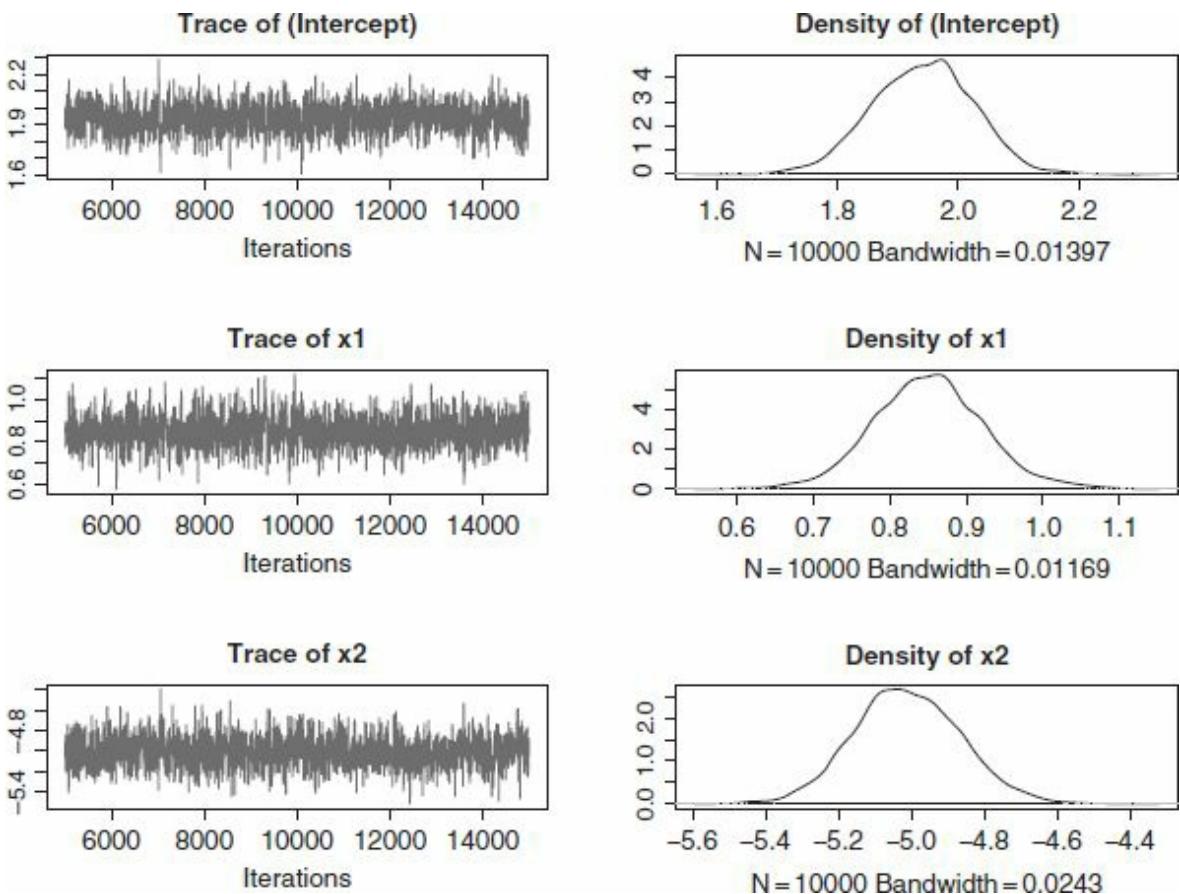


Figure 5.15 Trace plot and posteriors for the three regression parameters of the logit model.

Bayesian Logistic Model in R using JAGS

The same `logitmod` data is modeled using JAGS. We expect that the results will be close to the results determined from the R `MCMClogit` function.

Code 5.17 Logistic model in R using JAGS.

```
=====
attach(logitmod)
require(R2jags)

X <- model.matrix(~ x1 + x2,
                  data = logitmod)
K <- ncol(X)
model.data <- list(Y = logitmod$by,
                     N = nrow(logitmod),
                     X = X,
                     K = K,
                     LogN = log(nrow(logitmod)),
                     b0 = rep(0, K),
                     B0 = diag(0.00001, K)
)
sink("LOGIT.txt")
cat("
model{
  # Priors
  beta ~ dnorm(b0[], B0[,])
  # Likelihood
  for (i in 1:N){
```

```

Y[i] ~ dbern(p[i])
logit(p[i]) <- eta[i]
# logit(p[i]) <- max(-20,min(20,eta[i])) used to avoid numerical
# instabilities
# p[i] <- 1/(1+exp(-eta[i])) can use for logit(p[i]) above
eta[i] <- inprod(beta[], X[i,])
LLi[i] <- Y[i] * log(p[i]) +
           (1 - Y[i]) * log(1 - p[i])
}
LogL <- sum(LLi[1:N])
AIC <- -2 * LogL + 2 * K
BIC <- -2 * LogL + LogN * K
}
", fill = TRUE)
sink()

# Initial parameter values
inits <- function () {
  list(
    beta  = rnorm(K, 0, 0.1)
  )
}
params <- c("beta", "LogL", "AIC", "BIC")

LOGT <- jags(data = model.data,
               inits = inits,
               parameters = params,
               model.file = "LOGIT.txt",
               n.thin = 1,
               n.chains = 3,
               n.burnin = 5000,
               n.iter = 10000)
print(LOGT, intervals=c(0.025, 0.975), digits=3)
=====
      mu.vect  sd.vect   2.5%   97.5%   Rhat  n.eff
AIC      5080.34    2.54  5077.47  5086.90  1.03   140
BIC      5099.89    2.54  5097.02  5106.45  1.03   140
LogL     -2537.17   1.27 -2540.45  2535.73  1.03   140
beta[1]    1.95    0.09    1.78    2.12  1.01   920
beta[2]    0.85    0.07    0.70    1.00  1.00  4600
beta[3]   -5.01    0.15   -5.30   -4.72  1.01   580
deviance   5074.34    2.54  5071.47  5080.90  1.03   140

```

pD = 3.2 and DIC = 5077.5

The parameter values obtained with the JAGS code are nearly the same as those obtained using `MCMClogit`. To display the chains and histograms of the model parameters (Figures 5.16 and 5.17) use

```

source("CH-Figures.R")
out <- LOGT$BUGSoutput
MyBUGSHist(out,c(uNames("beta",K),"AIC", " BIC", "LogL"))
MyBUGSChains(out,c(uNames("beta",K),"AIC", " BIC", "LogL"))

```

Figure 5.18 displays the fitted model. The y -axis represents the probability of success as function of x_1 for $x_2 = 1$ (lower curve) and $x_2 = 0$ (upper curve).

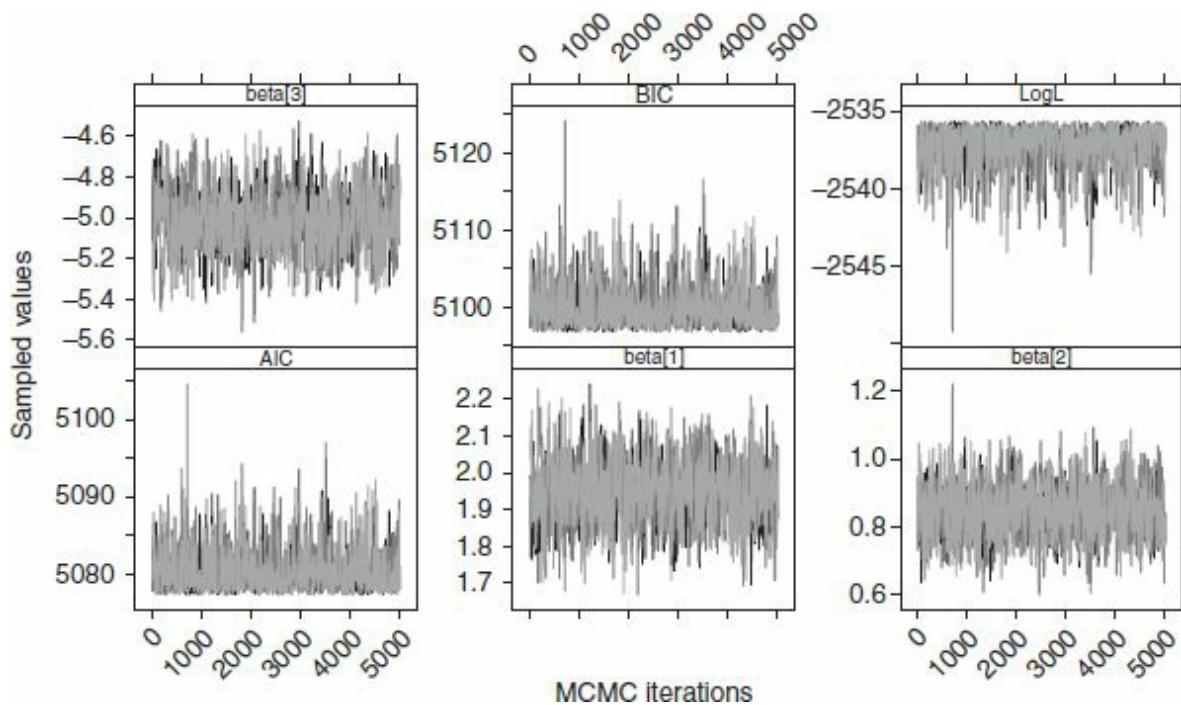


Figure 5.16 MCMC chains for the three model parameters, β_1 , β_2 , and β_3 , and for the log-likelihood, the AIC and BIC, for the Bernoulli model.

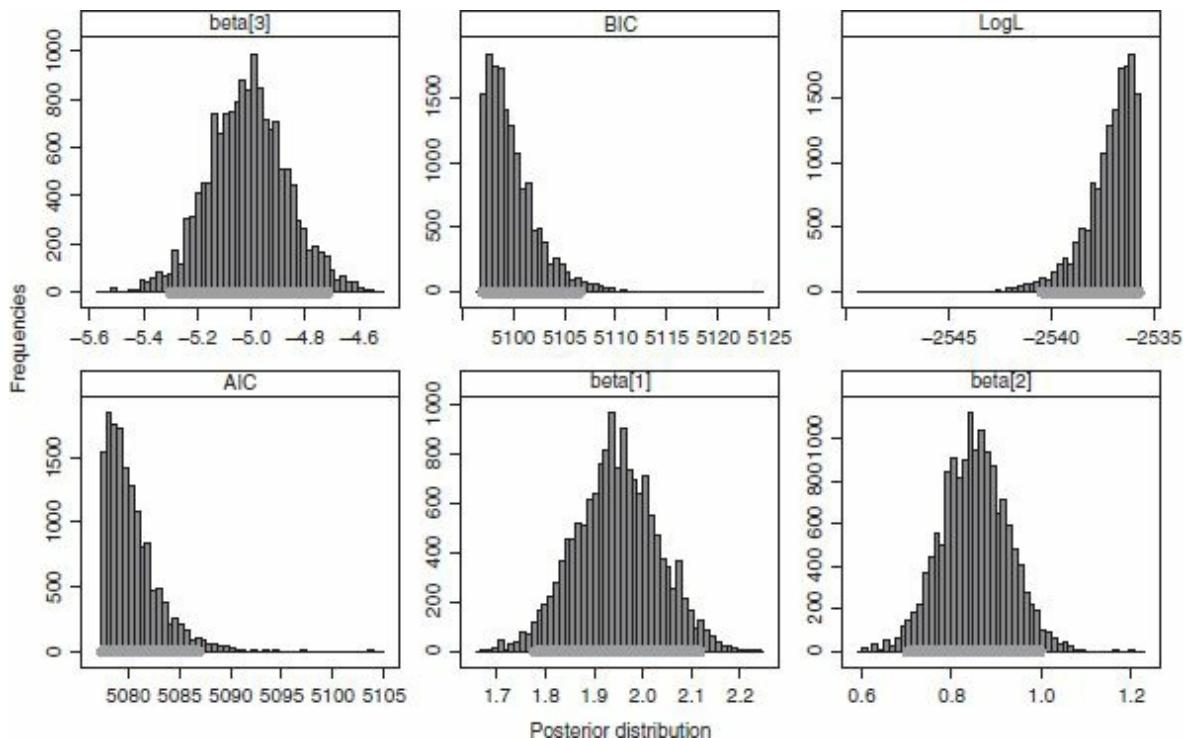


Figure 5.17 Histogram of the MCMC iterations for each parameter. The thick line at the base of each histogram represents the 95% credible interval. Note that no 95% credible interval contains 0.

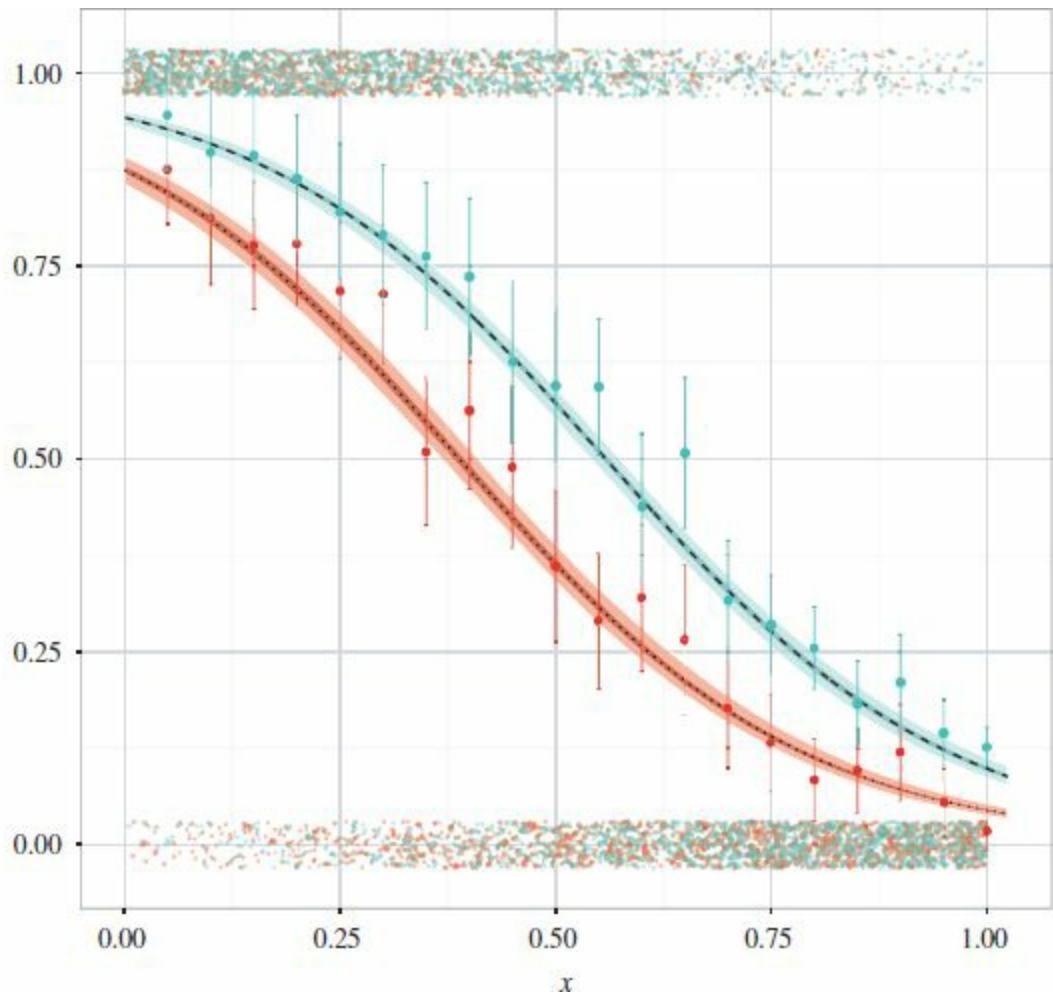


Figure 5.18 Visualization of the synthetic data from the Bayesian logistic model. The dashed and dotted lines and respective shaded areas show the fitted and 95% probability intervals. The dots in the upper horizontal border correspond to observed successes for each binary predictor and those in the lower horizontal border correspond to observed failures. The dots with error bars denote the fraction of successes, in bins of 0.05.

Note that in Code 5.17 the line

```
p[i] <- 1/(1+exp(-eta[i]))
```

can be substituted for

```
logit(p[i]) <- eta[i]
```

This is the inverse link function, which defines the logistic fitted probability, or μ , but in fact p is often used to symbolize the predicted probability, as previously mentioned. It is better programming practice to use the `logit()` and similar functions in situations like this rather than the actual formula. We show both in case you use this code for models that do not have a function like `logit()`, e.g., a Bernoulli model with a complementary loglog link. The above substitution can be applied to the Bernoulli, binomial, and beta binomial models, as we shall observe.

We have added a calculation of the AIC and BIC goodness-of-fit statistics. These fit-test statistics are comparative tests; they have no real informative power by themselves. They may be used on both nested and non-nested data, unlike most fit tests. The deviance information criterion (DIC; Spiegelhalter *et al.*, 2002) test is similar to AIC and BIC, which are acronyms for the Akaike information criterion (Akaike, 1974) and the Bayesian information criterion (Schwarz, 1978) respectively. However, the DIC is specific to Bayesian models and should be used when assessing comparative model value in preference to AIC and BIC. We shall discuss the deviance, the DIC, and pD tests in Chapter 9. The way in which we set up the AIC and BIC tests in this model using JAGS, however, can be used for the other models we address in this volume.

There are times when the binary response data that we are modeling is correlated. Remember, a likelihood, or log-likelihood, function is based on a probability distribution. As such, it involves the assumption that each observation in the data is independent of the others. But what if the model data has been gathered in various small groups? A typical example used in the social sciences is data collected about students in schools throughout a large metropolitan area. It is likely that teaching methods are more similar within schools than between schools. The data is likely to be correlated on the basis of the differences in within-school teaching methods. To assess student learning across all the schools in the area without accounting for the within-school correlation effect would bias the standard errors. This same panel or nesting effect may also occur in astrophysical data, when, for example, comparing properties of galaxies within galaxy clusters and between galaxy clusters. In frequency-based modeling, analysts typically scale the model standard errors, apply a sandwich or robust adjustment to the standard errors, or model the data as a fixed or random effects panel model. We discuss panel models in Chapter 8.

When employing a Bayesian logistic model on the data, it is still important to account for possible correlations in the data. Using sandwich or robust adjustments to the model is recommended as a first type of adjustment when the model has been estimated using maximum likelihood. However, such an adjustment is not feasible for a Bayesian model. Scaling can be, though. The scaling of a logistic model produces what R calls a quasi-binomial model. It simply involves multiplying the standard error by the square root of the dispersion statistic. This may be done to the posterior standard deviations. We define the dispersion statistic as the ratio of the Pearson χ^2 statistic and the residual degrees of freedom. For binomial models the deviance statistic may also be used in place of the Pearson χ^2 . The deviance statistic is produced in the R `glm` model output, as well as in the default JAGS output. The residual degrees of freedom is calculated as the number of model observations less the number of parameters in the model. We suggest scaling the binomial posterior standard deviations only if there is a substantial difference in values between the scaled and default standard deviation values. You will then need to calculate new credible intervals based on the scaled standard deviation values. `HPDinterval()`, from the package `lme4`, for calculating credible intervals may also be used in place of scaling; `HPD` is an acronym for highest posterior density intervals. Again, we recommend this method of adjustment only if there is a substantial panel-effect correlation in the data. But in that case it would be preferable to employ a hierarchical model on the data (Chapter 8).

Bayesian Logistic Model in Python using `pymc3`

Code 5.18 Logistic model using `pymc3`.¹

```

=====
import numpy as np
from scipy.stats import bernoulli, uniform, binom
import pymc3 as pm
import pylab as plt
import pandas

def invlogit(x):
    """ Inverse logit function.

        input: scalar
        output: scalar
    """
    return 1.0 / (1 + np.exp(-x))

# Data
np.random.seed(13979)                      # set seed to replicate example
nobs = 5000                                  # number of obs in model

x1 = binom.rvs(1, 0.6, size=nobs)
x2 = uniform.rvs(size=nobs)

beta0 = 2.0
beta1 = 0.75
beta2 = -5.0

xb = beta0 + beta1 * x1 + beta2 * x2
exb = 1.0/(1 + np.exp(-xb))                 # logit link function

by = binom.rvs(1, exb, size=nobs)

df = pandas.DataFrame({'x1': x1, 'x2': x2, 'by': by})   # re-write data

# Fit
niter = 5000                                     # parameters for MCMC

with pm.Model() as model_glm:
    # Define priors
    beta0 = pm.Flat('beta0')
    beta1 = pm.Flat('beta1')
    beta2 = pm.Flat('beta2')

    # Likelihood
    p = invlogit(beta0 + beta1 * x1 + beta2 * x2)
    y_obs = pm.Binomial('y_obs', n=1, p=p, observed=by)

    # Inference
    start = pm.find_MAP()
    step = pm.NUTS()
    trace = pm.sample(niter, step, start, progressbar=True)

# Print summary to screen
pm.summary(trace)

# Show graphical output
pm.traceplot(trace)
plt.show()
=====

beta0:
      Mean        SD       MC Error     95% HPD interval
-----+-----+-----+-----+-----+
      1.920      0.083      0.002      [1.757, 2.078]
Posterior quantiles:
      2.5        25         50         75        97.5
      |-----+-----+-----+-----+-----|
      1.760      1.865      1.917      1.976      2.081

beta1:
      Mean        SD       MC Error     95% HPD interval
-----+-----+-----+-----+-----+
      0.753      0.071      0.001      [0.617, 0.897]
Posterior quantiles:
      2.5        25         50         75        97.5
      |-----+-----+-----+-----+-----|
      0.611      0.7055     0.754      0.802      0.893

beta2:
      Mean        SD       MC Error     95% HPD interval
-----+-----+-----+-----+-----+

```

```

-4.883          0.144          0.004      [-5.186, -4.620]
Posterior quantiles:
 2.5           25            50            75          97.5
|-----|=====|=====|=====|-----|
-5.179       -4.978       -4.879       -4.787       -4.612

```

Bayesian Logistic Model in Python using Stan

The following code shows the Python/Stan equivalent of the logistic model discussed earlier. This implementation consumes a lot of memory, so we advise keeping `n_jobs=1` when calling the `pystan` function.

Code 5.19 Logistic model in Python using Stan.

```

=====
import numpy as np
import statsmodels.api as sm
import pystan

from scipy.stats import uniform, bernoulli

# Data
np.random.seed(13979)                      # set seed to replicate example
nobs = 5000                                    # number of obs in model

x1 = bernoulli.rvs(0.6, size=nobs)
x2 = uniform.rvs(size=nobs)

beta0 = 2.0
beta1 = 0.75
beta2 = -5.0

xb = beta0 + beta1 * x1 + beta2 * x2
exb = 1.0/(1 + np.exp(-xb))                  # logit link function

by = bernoulli.rvs(exb, size=nobs)

mydata = {}
mydata['K'] = 3
mydata['X'] = sm.add_constant(np.column_stack((x1,x2)))
mydata['N'] = nobs
mydata['Y'] = by
mydata['LogN'] = np.log(nobs)

# Fit
stan_code = """
data{
    int<lower=0> N;
    int<lower=0> K;
    matrix[N, K] X;
    int Y[N];
    real LogN;
}
parameters{
    vector[K] beta;
}
transformed parameters{
    vector[N] eta;
        eta = X * beta;
}
model{
    Y ~ bernoulli_logit(eta);
}
generated quantities{
    real LL[N];
    real AIC;
    real BIC;
    real LogL;
    real<lower=0, upper=1.0> pnew[N];
    vector[N] etanew;
    etanew = X * beta;
}
```

```

for (i in 1:N){
  pnew[i] = inv_logit(etanew[i]);
  LL[i] = bernoulli_lpmf(1|pnew[i]);
}
LogL = sum(LL);
AIC = -2 * LogL + 2 * K;
BIC = -2 * LogL + LogN * K;
}
"""
fit = pystan.stan(model_code=stan_code, data=mydata, iter=10000, chains=3,
                   warmup=5000, n_jobs=1)
# Output
lines = list(range(8)) + [2 * nobs + 8, 2 * nobs + 9, 2 * nobs + 10]
output = str(fit).split('\n')

for i in lines:
  print (output[i])
=====

```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
beta[0]	1.92	2.4e-3	0.08	1.76	1.87	1.92	1.98	2.08	1115.0	1.0
beta[1]	0.75	2.0e-3	0.07	0.62	0.7	0.75	0.8	0.9	1296.0	1.0
beta[2]	-4.89	4.3e-3	0.15	-5.18	-4.99	-4.88	-4.79	-4.6	1125.0	1.0
AIC	9651.1	5.24	217.68	9239.7	9501.8	9647.8	9798.4	1.0e4	1723.0	1.0
BIC	9670.7	5.24	217.68	9259.2	9521.3	9667.4	9818.0	1.0e4	1723.0	1.0
LogL	-4822	2.62	108.84	-5039	-4896	-4820	-4747	-4616	1723.0	1.0

5.3.2 Bayesian Bernoulli Probit Models

The binary probit model is a non-canonical Bernoulli model. The difference between a logistic and a probit model in terms of programming is the substitution of the cumulative standard normal distribution function in place of the logit function. The code displayed in Code 5.19 is the same as for a simple synthetic logistic model, except that `pnorm(xb)` replaces `1/(1+exp(-xb))` in defining `exb`. Furthermore, `1/(1+exp(-xb))` is identical to `exp(xb)/(1+exp(xb))` and is the inverse logistic link function. It is the same as the `inv.logit` function, which is in the `boot` package. The following code demonstrates their equivalence:

```

> library(boot)
> inv.logit(0.4)
[1] 0.5986877

```

and

```

> 1/(1+exp(-0.4))
[1] 0.5986877

```

Code 5.20 Synthetic probit data and model generated in R

```

=====
set.seed(135)
nobs <- 1:2000
x1 <- runif(nobs)
x2 <- 2*runif(nobs)
xb <- 2 + 0.75 * x1 - 1.25 * x2
exb <- pnorm(xb)      # probit inverse link
py <- rbinom(nobs, size=1, prob=exb)
probdata <- data.frame(py, x1, x2)
=====
```

We add a note on when a probit model should be used. The origin of the probit model derives from the case where statisticians divide a normally distributed variable into two components, 0 and 1. The goal might be to determine which pattern of explanatory predictors produces predicted probabilities greater than 0.5 and which pattern produces predicted probabilities less than 0.5.

Whatever goal a statistician might have, though, the important point is that if the binary data originate from an underlying, latent, normally distributed variable, which may be completely unknown, a probit model is preferable to a logit, complementary loglog, or loglog model. Typically, predicted probit probabilities are little different in value to predicted logit probabilities, so which model one uses may be a matter of simple preference. However, when modeled using maximum likelihood, exponentiated logistic model coefficients are interpreted as odds ratios,¹ which is very useful in fields such as medicine, social science, and so forth. Exponentiated probit coefficients cannot be interpreted as odds ratios and have no interesting interpretation. For this reason the maximum likelihood (and GLM) logistic regression model is very popular. In the Bayesian context, exponentiated logistic parameter means are not technically odds ratios although some analysts have interpreted them in such a manner. Their values may be identical to those produced in maximum likelihood estimation, but the theoretical basis for this interpretation is missing. In astronomy the odds and odds-ratio concepts are rarely if ever used.

When making the selection of either a logistic or a probit Bayesian model, the best tactic is to evaluate the comparative diagnostic statistics and compare the DIC statistics. If the diagnostics are nearly the same, but the DIC statistic of a logistic model is less than that for a probit model on the same data, the logistic model should be preferred.

Bayesian Probit Model using R

We now model the same data as we did for the Bernoulli logistic model using the `MCMCprobit` function in `MCMCpack`. The same burn-in value is given and the same number of sampling iterations.

Code 5.21 Probit model using R.

```
=====
library(MCMCpack)

myPL <- MCMCprobit(py ~ x1 + x2,
                     burnin = 5000,
                     mcmc = 100000,
                     data = probdata)
summary(myPL)
plot(myPL)
=====

      Mean        SD   Naive SE Time-series SE
(Intercept) 1.9391 0.10794 0.0003413 0.0008675
x1          0.7185 0.12486 0.0003948 0.0008250
x2         -1.2496 0.07079 0.0002239 0.0006034
Quantiles for each variable:
      2.5%     25%     50%     75%    97.5%
(Intercept) 1.7302 1.8662 1.9381 2.0119 2.1535
x1          0.4756 0.6339 0.7185 0.8022 0.9645
x2         -1.3900 -1.2970 -1.2489 -1.2017 -1.1120
```

The results show that the parameter means are close to those specified in the synthetic probit data. As for the logistic model, the trace and density plots appear to fit the model well.

Bayesian Probit Model in R using JAGS

As we did for the Bernoulli logistic data, we model the same data as for the `MCMCprobit` model. We use JAGS this time. Notice that the only difference in the code between the logit and probit models

is the following line:

```
probit(p[i]) <- max(-20, min(20, eta[i]))
```

Again, this is due to the fact that the logit and probit models are both based on the Bernoulli distribution and only differ in their link functions.

Code 5.22 Probit model in R using JAGS.

```
=====
attach(probdata)
require(R2jags)
set.seed(1944)

X <- model.matrix(~ x1 + x2,
                  data = probdata)
K <- ncol(X)
model.data <- list(Y = probdata$py,
                     N = nrow(probdata),
                     X = X,
                     K = K,
                     LogN = log(nrow(probdata)),
                     b0 = rep(0, K),
                     B0 = diag(0.00001, K)
                    )
sink("PROBIT.txt")
cat(")
model{
  # Priors
  beta ~ dmnorm(b0[], B0[,])

  # Likelihood
  for (i in 1:N){
    Y[i] ~ dbern(p[i])
    probit(p[i]) <- max(-20, min(20, eta[i]))
    eta[i] <- inprod(beta[], X[i,])
    LLi[i] <- Y[i] * log(p[i]) +
      (1 - Y[i]) * log(1 - p[i])
  }
  LogL <- sum(LLi[1:N])

  # Information criteria
  AIC <- -2 * LogL + 2 * K
  BIC <- -2 * LogL + LogN * K
}
",fill = TRUE)
sink()

# Initial parameter values
inits <- function () {
  list(
    beta = rnorm(K, 0, 0.1)
  )
}

# Parameters and statistics to display
params <- c("beta", "LogL", "AIC", "BIC")

PROBT <- jags(data = model.data,
               inits = inits,
               parameters = params,
               model.file = "PROBIT.txt",
               n.thin = 1,
               n.chains = 3,
               n.burnin = 5000,
               n.iter = 10000)
print(PROBT, intervals=c(0.025, 0.975), digits=3)
=====

mu.vect    sd.vect    2.5%    97.5%    Rhat  n.eff
AIC       1635.908   2.376   1633.115   1641.988  1.008   290
BIC       1652.711   2.376   1649.918   1658.790  1.008   290
LogL     -814.954   1.188   -817.994   -813.558  1.008   290
beta[1]    1.934   0.107    1.720    2.148  1.012   190
beta[2]    0.714   0.128    0.472    0.968  1.005   480
beta[3]   -1.245   0.068   -1.381   -1.114  1.007   460
deviance   1629.908   2.376   1627.115   1635.988  1.008   290

pD = 2.8 and DIC = 1632.7
```

The results are nearly identical to the parameter estimates obtained with `MCMCprobit`.

Other non-canonical Bernoulli models include the complementary cloglog and loglog models.

These links are found in GLM software but are rarely used. Both these alternatives are asymmetric about the distributional mean, 0.5. In terms of Bayesian applications, an analyst may amend the line of code we used to distinguish JAGS logit from probit models, in order to model cloglog and loglog models. This is done by using the formulae for their respective inverse link functions. In Table 5.2 we display a list of Bernoulli (binomial) inverse link functions, which are used to define `mu`, the fitted or predicted value.

Table 5.2 Bernoulli inverse link functions.

Logit	$1/(1 + \exp(-\eta))$ or $\exp(\eta)/(1 + \exp(\eta))$	<code>inv.logit(η)</code>
Probit	$\Phi(\eta)$	<code>pnorm(η)</code>
Cloglog	$1 - \exp(-\exp(\eta))$	
Loglog	$\exp(-\exp(-\eta))$	

Bayesian Probit Model using Python

Code 5.23 Probit model in Python.

```
=====
import numpy as np
from scipy.stats import norm, uniform, bernoulli
import pymc3 as pm
import pylab as plt
import pandas
import theano.tensor as tsr

def probit_phi(x):
    """Probit transformation."""
    mu = 0
    sd = 1
    return 0.5 * (1 + tsr.erf((x - mu) / (sd * tsr.sqrt(2)))) 

# Data
np.random.seed(135)                      # set seed to replicate example
nobs = 5000                                # number of obs in model

x1 = uniform.rvs(size=nobs)
x2 = 2 * uniform.rvs(size=nobs)

beta0 = 2.0                                  # coefficients for linear predictor
beta1 = 0.75
beta2 = -1.25

xb = beta0 + beta1 * x1 + beta2 * x2      # linear predictor
exb = 1 - norm.sf(xb)                      # inverse probit link

py = bernoulli.rvs(exb)

df = pandas.DataFrame({'x1': x1, 'x2': x2, 'by': py})  # re-write data

# Fit
niter = 10000                               # parameters for MCMC

with pm.Model() as model_glm:
    # define priors
    beta0 = pm.Flat('beta0')
    beta1 = pm.Flat('beta1')
    beta2 = pm.Flat('beta2')

    # define likelihood
    theta_p = beta0 + beta1*x1 + beta2 * x2
    theta = probit_phi(theta_p)
    y_obs = pm.Bernoulli('y_obs', p=theta, observed=py)

    # inference
    start = pm.find_MAP()                  # find starting value by optimization
    step = pm.NUTS()
```

```

trace = pm.sample(niter, step, start, random_seed=135, progressbar=True)

# Print summary to screen
pm.summary(trace)

# Show graphical output
pm.traceplot(trace)
plt.show()
=====
beta0:
    Mean      SD      MC Error      95% HPD interval
    -----+-----+-----+-----+-----+
    1.976    0.072    0.001      [1.837, 2.116]
Posterior quantiles:
    2.5      25      50      75      97.5
    |-----+=====+=====+=====+-----|
    1.836    1.928    1.976    2.024    2.116

beta1:
    Mean      SD      MC Error      95% HPD interval
    -----+-----+-----+-----+-----+
    0.760    0.084    0.001      [0.602, 0.927]
Posterior quantiles:
    2.5      25      50      75      97.5
    |-----+=====+=====+=====+-----|
    0.598    0.703    0.759    0.815    0.923

beta2:
    Mean      SD      MC Error      95% HPD interval
    -----+-----+-----+-----+-----+
    -1.234   0.048    0.001      [-1.330, -1.142]
Posterior quantiles:
    2.5      25      50      75      97.5
    |-----+=====+=====+=====+-----|
    -1.329   -1.266   -1.234   -1.202   -1.140

```

Bayesian Probit Model in Python using Stan

Code 5.24 Probit model in Python using Stan.

```

=====
import numpy as np
import pystan
import statsmodels.api as sm

from scipy.stats import uniform, norm, bernoulli

# Data
np.random.seed(1944)                      # set seed to replicate example
nobs = 2000                                  # number of obs in model
x1 = uniform.rvs(size=nobs)
x2 = 2 * uniform.rvs(size=nobs)

beta0 = 2.0
beta1 = 0.75
beta2 = -1.25

xb = beta0 + beta1 * x1 + beta2 * x2
exb = 1 - norm.sf(xb)                      # inverse probit link

py = bernoulli.rvs(exb)

# Data transformation for illustrative purposes
K = 3                                         # number of coefficients
X = np.column_stack((x1, x2))
X = sm.add_constant(X)

# Fit
probit_data = {}
probit_data['N'] = nobs
probit_data['K'] = K
probit_data['X'] = X
probit_data['Y'] = py
probit_data['logN'] = np.log(nobs)

probit_code = """
data{
```

```

int<lower=0> N;
int<lower=0> K;
matrix[N,K] X;
int Y[N];
real logN;
}
parameters{
  vector[K] beta;
}
transformed parameters{
  vector[N] xb;
  xb = X * beta;
}
model{
  for (i in 1:N) Y[i] ~ bernoulli(Phi(xb[i])); # likelihood
}
generated quantities{
  real LLi[N];
  real AIC;
  real BIC;
  real LogL;
  vector[N] xb2;
  real p[N];
  xb2 = X * beta;

  for (i in 1:N){
    p[i] = Phi(xb2[i]);
    LLi[i] = Y[i] * log(p[i]) + (1-Y[i]) * log(1 - p[i]);
  }

  LogL = sum(LLi);
  AIC = -2 * LogL + 2 * K;
  BIC = -2 * LogL + logN * K;
}
"""

fit = pystan.stan(model_code=probit_code, data=probit_data, iter=5000,
                   chains=3, warmup=3000, n_jobs=3)

# Output
lines = list(range(8)) + [2 * nobs + 8, 2 * nobs + 9, 2 * nobs + 10]
output = str(fit).split('\n')

for i in lines:
  print(output[i])

=====
      mean   se_mean     sd   2.5%   25%   50%   75%   97.5%   n_eff   Rhat
beta[0]   1.92   3.4e-3   0.11   1.71   1.85   1.92   1.99   2.14   1075.0   1.0
beta[1]   0.67   3.7e-3   0.12   0.44   0.59   0.68   0.76   0.92   1147.0   1.0
beta[2]   -1.2   2.2e-3   0.07  -1.34  -1.25  -1.2   -1.16  -1.06   1085.0   1.0
AIC      1643.0   0.08   2.43  1640.3  1641.2  1642.4  1644.1  1649.1   911.0   1.0
BIC      1659.8   0.08   2.43  1657.1  1658.0  1659.2  1660.9  1665.9   911.0   1.0
LogL    -818.5   0.04   1.21  -821.5  -819.0  -818.2  -817.6  -817.1   911.0   1.0

```

5.3.3 Bayesian Grouped Logit or Binomial Model

The grouped logistic model, or binomial regression, is based on a response parameterized in terms of a ratio. The response y represents the number of successes out of m identical covariate patterns. Such data is usually found in tabular form.

The format of the grouped data can be explained using a simple example. Twenty-seven observations are represented in the table below. The response y indicates the number of binary $\in \{0, 1\}$ successes for observations having unique predictor profiles for the data; m represents the number of observations having a specific predictor profile. For instance, in the first line, of the eight

observations having values $x_1 = 1$ and $x_2 = 0$, two have y values of 1. The third line tells us that, of the three observations with values of 1 for both x_1 and x_2 , none have y values of 1. The fourth observation tells us that all seven of the observations with the value 0 for both predictors have y values of 1.

y	m	x_1	x_2
2	8	1	0
6	9	0	1
0	3	1	1
7	7	0	0

The binomial and beta binomial models are appropriate for modeling such data. The beta binomial model, however, is used when a binomial model is overdispersed. We discuss the beta binomial in the next section.

The binomial model is the probability distribution most often used for binary response data. When the response term is binary and the binomial denominator is 1, the distribution is called Bernoulli. The standard logistic and probit models with binary response data are based on the Bernoulli distribution, which itself is a subset of the binomial distribution. We described the structure of a binomial data set above. We now turn to the binomial probability distribution, the binomial log-likelihood, and other equations fundamental to the estimation of binomial-based models. We shall discuss the logistic and probit binomial models in this section, noting that they are frequently referred to as grouped logistic and grouped probit models. Our main discussion will relate to the binomial logistic model since it is used in research far more than is the probit or other models such as the complementary loglog and loglog models. These are binomial models as well, but with different link functions.

The binomial PDF in terms of p is expressed as

$$f(y; p, m) = \binom{m_i}{y_i} p_i^{y_i} (1 - p_i)^{m_i - y_i}. \quad (5.21)$$

The log-likelihood PDF in terms of μ may be given as

$$\mathcal{L}(\mu; y, m) = \sum_{i=1}^n \left\{ y_i \ln \left(\frac{\mu_i}{1 - \mu_i} \right) + m \ln(1 - \mu_i) + \ln \binom{m_i}{y_i} \right\} \quad (5.22)$$

where the choose function within the final term of the log-likelihood can be re-expressed in terms of factorials as

$$\frac{m_i!}{y_i!(m_i - y_i)!}. \quad (5.23)$$

The log of the choose function may be converted to log-gamma functions and expressed as

$$\ln \Gamma(m_i + 1) - \ln \Gamma(y_i + 1) - \ln \Gamma(m_i - y_i + 1). \quad (5.24)$$

These three terms are very often used in estimation algorithms to represent the final term of the log-likelihood, which is the normalization term of the binomial PDF. Normalization of a PDF guarantees that the individual probabilities in the PDF sum to 1.

The pseudocode for the binomial log-likelihood can be given as

```
LL <- y*log(mu/(1 - mu)) + m * log(1 - mu) +
loggam(m + 1) - loggam(y + 1) - loggam(m - y + 1)
```

The pseudocode for the binomial log-likelihood in term of the linear predictor xb can be given as

```
LL <- y * log(1/(1 + exp(xb))) + (m - y) * log(1/(1 + exp(-xb))) +
loggam(m + 1) - loggam(y + 1) - loggam(m - y + 1)
```

We provide this parameterization of the binomial log-likelihood function since it is so frequently used in estimation. It is typical, though, when writing Bayesian code to convert the linear predictor xb , to μ using the inverse binomial logistic link function defining μ as $m/(1+exp(-xb))$. Note also that the `loggam` function is used in JAGS for the log-gamma function. The binomial log-likelihood function may also be given in the estimation process using the combination term for the normalization, as shown below:

```
L1 <- ln(comb(n, y))
L2 <- L1 + y * log(1/(1 + exp(xb))) + (m - y) * log(1/(1 + exp(-xb)))
LL <- L1 + L2
```

`MCMCpack` does not have a built-in binomial or grouped logistic function. It is possible to create one's own function as an option to `MCMCregress`, but it is much easier to use JAGS from within R to do the modeling.

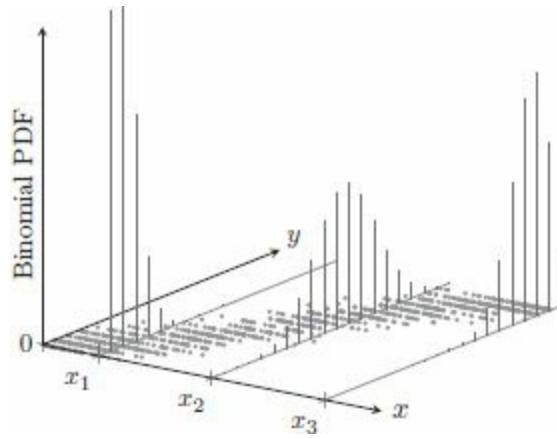


Figure 5.19 Illustration of binomial-distributed data.

Binomial Model in R using JAGS

We will demonstrate the grouped logistic, or binomial, model using simulated data with specified parameter mean values. The model data consist of an intercept with value -2 and slope (mean) values for x_1 as -1.5 and x_2 as 3 . There are 2000 observations in the data.

Code 5.25 Synthetic data from a binomial model in R.

```
=====
set.seed(33559)
nobs = 2000
m = 1 + rpois(nobs, 5)
x1 = runif(nobs)
x2 = runif(nobs)
xb <- -2 - 1.5 * x1 + 3 * x2
exb <- exp(xb)
p <- exb/(1 + exb) # prob of p=0

y <- rbinom(nobs, prob=p, size=m)
bindata=data.frame(y=y, m=m, x1=x1, x2=x2)
=====
```

The following code may be used to obtain maximum likelihood or GLM results from modeling the above data. Note that the parameter estimates closely approximate the values specified in the code creating the data. They also will be close to the posterior parameter means calculated from the JAGS code 5.26 below.

```
> noty <- m - y
> mybin <- glm(cbind(y, noty) ~ x1 + x2, family=binomial, data=bindata)
> summary(mybin)
Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -1.99406   0.06354 -31.38 <2e-16 ***
x1          -1.59785   0.08162 -19.58 <2e-16 ***
x2           3.09066   0.08593  35.97 <2e-16 ***
```

The simulated binomial numerator values are contained in y and the binomial denominator values in m . The JAGS code for modeling the above data is given in Code 5.26. In place of using the written out code for the binomial log-likelihood, we use the internal JAGS `dbin` function with the m option for the log-likelihood. It is generally faster.

Code 5.26 Binomial model in R using JAGS.

```
=====
library(R2jags)
X <- model.matrix(~ x1 + x2, data = bindata)
K <- ncol(X)

model.data <- list(Y = bindata$y,
                     N = nrow(bindata),
                     X = X,
                     K = K,
                     m = bindata$m)
)

sink("GLOGIT.txt")
cat("
model{
# Priors
# Diffuse normal priors betas
for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001)}

# Likelihood
for (i in 1:N){
Y[i] ~ dbin(p[i],m[i])
logit(p[i]) <- eta[i]
eta[i] <- inprod(beta[], X[i,])
}
}
",fill = TRUE)
sink()

inits <- function () {
list(
  beta = rnorm(K, 0, 0.1)
) }

params <- c("beta")

BINL <- jags(data = model.data,
              inits = inits,
              parameters = params,
              model.file = "GLOGIT.txt",
              n.thin = 1,
              n.chains = 3,
              n.burnin = 3000,
              n.iter = 5000)
print(BINL, intervals=c(0.025, 0.975), digits=3)
=====

      mu.vect    sd.vect    2.5%    97.5%   Rhat   n.eff
beta[1]     -1.962     0.179    -2.113    -1.539  1.001   5900
beta[2]     -1.577     0.174    -1.773    -1.110  1.004   570
beta[3]      3.042     0.295     2.209     3.261  1.003   5500
deviance    4980.431   265.824   4945.788   5126.447  1.001   6000
=====

pD = 35342.8 and DIC = 40323.3
```

The estimated mean parameter values are close to the values we specified when the data were created. The specified parameter mean values for the intercept and for x_1 and x_2 are -2 , -1.5 , and 3 . The model parameter values are -1.96 , -1.58 , and 3.0 respectively.

It should be noted that the line

```
logit(p[i]) <- eta[i]
```

can be changed to a clear representation of the inverse link function:

```
p[i] <- 1/(1 + exp(-eta[i]))
```

Here `logit(p[i])` is identical to the inverse link, which provides the fitted probability. Substituting one line for the other results in the same results, given a sampling error, that is.

Modeling Real Data in Grouped Format

Given that data can usually be presented in terms of a table, or in blocks or groups of information, we will show a real example of how Code 5.26 can be used to model 582 observations with the information cast into 12 groups. The data has the same structure as that in the above model, except that we have added a third parameter (predictor) explaining the response. To differentiate between this data and the synthetic data used above, we shall call it `bindata1`.

Code 5.27 Real data for a grouped binomial model.

```
=====
y <- c(6,11,9,13,17,21,8,10,15,19,7,12)
m <- c(45,54,39,47,29,44,36,57,62,55,66,48)
x1 <- c(1,1,1,1,1,0,0,0,0,0,0,0)
x2 <- c(1,1,0,0,1,1,0,0,1,1,0,0)
x3 <- c(1,0,1,0,1,0,1,0,1,0,1,0)
bindata1 <- data.frame(y,m,x1,x2,x3)
=====
```

The only lines that need to be amended in Code 5.26 are those specifying the data set and the added predictor, `x3`. All else remains the same.

```
X <- model.matrix(~ x1 + x2 + x3, data = bindata1)
model.data <- list(Y = bindata1$y,
                     N = nrow(bindata1),
                     m = bindata1$m)
```

The resulting parameter means, standard deviations, and credible intervals are as follows:

	mu.vect	sd.vect	2.5%	97.5%	Rhat	n.eff
beta[1]	-1.325	0.179	-1.683	-0.982	1.001	2900
beta[2]	0.250	0.204	-0.162	0.635	1.001	3700
beta[3]	0.488	0.207	0.092	0.899	1.002	1700
beta[4]	-0.296	0.193	-0.678	0.075	1.001	3400
deviance	239042.577	2.778	239039.093	239049.584	1.000	1

pD = 3.9 and DIC = 239046.4

Given that the “true” parameters for the intercept and three coefficient parameters are, respectively, -1.333 , 0.25 , 0.5 , and -0.3 , the parameter values found by MCMC sampling are very close. The algorithm is fast owing to the grouped nature of the data.

Binomial Model in Python using Stan

Code 5.28 Binomial model in Python using Stan.

```
=====
import numpy as np
import statsmodels.api as sm
import pystan

from scipy.stats import uniform, poisson, binom

# Data
np.random.seed(33559)                      # set seed to replicate example
nobs = 2000                                     # number of obs in model
m = 1 + poisson.rvs(5, size=nobs)
```

```

x1 = uniform.rvs(size=nobs)           # random uniform variable
x2 = uniform.rvs(size=nobs)

beta0 = -2.0
beta1 = -1.5
beta2 = 3.0

xb = beta0 + beta1 * x1 + beta2 * x2
exb = np.exp(xb)
p = exb / (1 + exb)

y = binom.rvs(m, p)                 # create y as adjusted

mydata = {}
mydata['K'] = 3
mydata['X'] = sm.add_constant(np.column_stack((x1,x2)))
mydata['N'] = nobs
mydata['Y'] = y
mydata['m'] = m

# Fit
stan_code = """
data{
    int<lower=0> N;
    int<lower=0> K;
    matrix[N, K] X;
    int Y[N];
    int m[N];
}
parameters{
    vector[K] beta;
}
transformed parameters{
    vector[N] eta;
    vector[N] p;
    eta = X * beta;
    for (i in 1:N) p[i] = inv_logit(eta[i]);
}
model{
    Y ~ binomial(m, p);
}
"""

fit = pystan.stan(model_code=stan_code, data=mydata, iter=5000, chains=3,
                   warmup=3000, n_jobs=3)

# Output
nlines = 8
output = str(fit).split('\n')
for item in output[:nlines]:
    print(item)

=====
      mean   se_mean     sd   2.5%   25%   50%   75%  97.5%   n_eff   Rhat
beta[0] -2.08   1.9e-3  0.06  -2.2  -2.12  -2.08  -2.03  -1.95  1135.0   1.0
beta[1] -1.55   2.4e-3  0.08  -1.71  -1.61  -1.56  -1.5   -1.4   1143.0   1.0
beta[2]  3.15   2.6e-3  0.09   2.98   3.09   3.15   3.21   3.33  1179.0   1.0

```

Analogously to what was shown in the previous section, if we wish to explicitly use the example code 5.27 in Python, the data section of Code 5.28 must be changed as follows:

Code 5.29 Adaptation of binomial model data section of Code 5.28 allowing it to handle explicit three-parameter data.

```
=====
# Data
np.random.seed(33559)                      # set seed to replicate example

y = [6,11,9,13,17,21,8,10,15,19,7,12]
m = [45,54,39,47,29,44,36,57,62,55,66,48]
x1 = [1,1,1,1,1,1,0,0,0,0,0,0]
x2 = [1,1,0,0,1,1,0,0,1,1,0,0]
x3 = [1,0,1,0,1,0,1,0,1,0,1,0]

mydata = {}
mydata['K'] = 4
mydata['X'] = sm.add_constant(np.column_stack((x1,x2,x3)))
mydata['N'] = len(y)
mydata['Y'] = y
mydata['m'] = m

# Output nlines = 9
=====
```

The other sections remain unchanged. The screen output reads

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
beta[0]	-1.33	5.1e-3	0.19	-1.72	-1.46	-1.33	-1.21	-0.98	1310.0	1.0
beta[1]	0.25	5.4e-3	0.2	-0.13	0.12	0.25	0.39	0.66	1406.0	1.0
beta[2]	0.5	5.6e-3	0.21	0.1	0.36	0.5	0.64	0.91	1400.0	1.0
beta[3]	-0.3	5.3e-3	0.19	-0.68	-0.43	-0.3	-0.16	0.09	1360.0	1.0

which is consistent with the R result shown before.

5.3.4 Bayesian Grouped Probit Model

... in R using JAGS

The grouped logistic or binomial model can be made into a grouped or binomial probit model by changing a line in the above JAGS code from

```
logit(p[i]) <- eta[i]
```

to

```
probit(p[i]) <- eta[i]
```

The output when run on the real grouped data that we used for the logistic model above appears as

	mu_vect	sd_vect	2.5%	97.5%	Rhat	n.eff
beta[1]	-0.809	0.101	-1.010	-0.613	1.007	370
beta[2]	0.154	0.118	-0.075	0.385	1.003	930
beta[3]	0.293	0.113	0.067	0.517	1.002	1800
beta[4]	-0.176	0.113	-0.397	0.043	1.004	710
deviance	239042.292	2.657	239038.963	239048.871	1.000	1

```
pD = 3.5 and DIC = 239045.8
```

No seed value was given to the data or sampling algorithm, so each run will produce different results. The results will not differ greatly and should be within the range of the credible intervals 95% of the time.

...in Python using Stan

In Stan this corresponds to changing

```
p[i] = inv_logit(eta[i]);
```

to

```
p[i] = Phi(eta[i]);
```

in Code [5.28](#).

Using the data shown in Code [5.27](#) and the changes mentioned above, the output on the screen should look like

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
beta[0]	-0.81	2.9e-3	0.11	-1.01	-0.88	-0.81	-0.73	-0.6	1358.0	1.0
beta[1]	0.15	3.2e-3	0.12	-0.09	0.07	0.15	0.23	0.38	1394.0	1.0
beta[2]	0.29	3.2e-3	0.12	0.05	0.21	0.29	0.38	0.53	1450.0	1.0
beta[3]	-0.17	3.1e-3	0.11	-0.4	-0.25	-0.17	-0.1	0.05	1354.0	1.0

5.3.5 Bayesian Beta–Binomial Models

Beta–binomial models are used when there is evident overdispersion in binomial or grouped data. When the binomial-model dispersion statistic (Section [6.1](#)) is greater than 1.0 then the standard errors and therefore p -values are biased. Beta–binomial regression is a preferred way of dealing with such data. However, how one handles binomial overdispersion or underdispersion for that matter largely depends on the reason why overdispersion or extra correlation exists in the data. We shall discuss this topic later in the book, exploring alternative ways of ameliorating, or even eliminating, the effects of overdispersion in binomial and count model data. Beta–binomial regression is, however, a standard way of adjusting for extra-dispersion in binomial data.

The beta–binomial model is based on an underlying PDF that is a combination of beta and binomial distributions. Fortunately, the beta and binomial PDFs are conjugate, so the combination is relatively simple. Moreover, in a similar manner to Bernoulli and binomial models, the beta binomial may have a canonical logistic link, a probit link, a complementary loglog link, or a loglog link. The beta binomial models found in research are only very rarely other than logit-linked models. We therefore describe the code for this model.

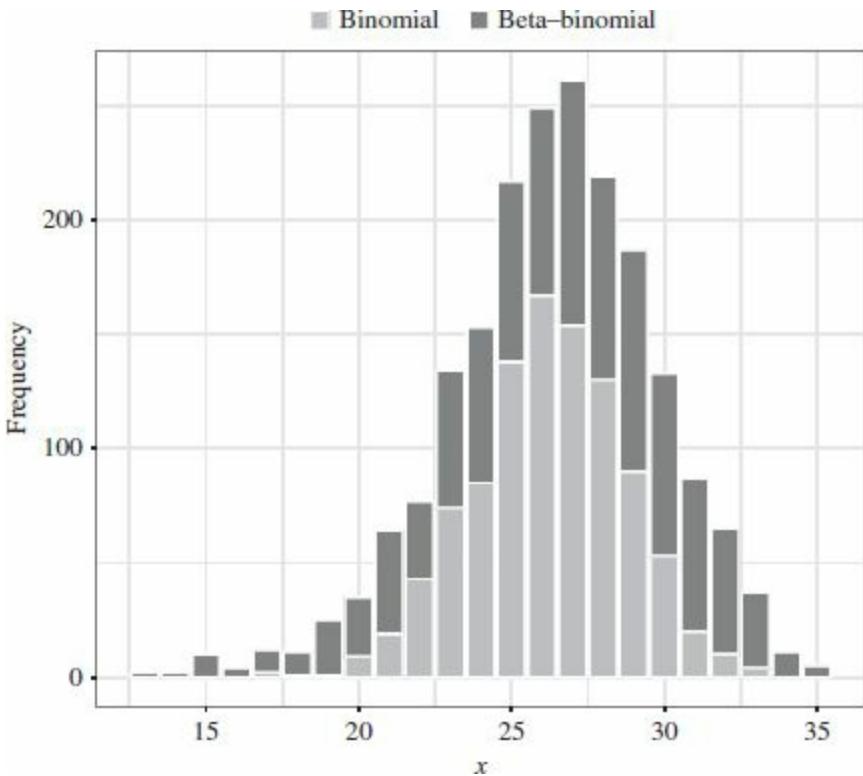


Figure 5.20 Histograms of a binomial distribution and a beta-binomial distribution with same total number of trials and probability of success.

Beta-Binomial Regression – Logit Link

We have

$$f(y; \mu, \sigma) = \frac{\ln \Gamma(n+1)}{\ln \Gamma(y+1) \ln \Gamma(n-y+1)} \frac{\ln \Gamma(1/\sigma) \ln \Gamma(y+\mu/\sigma) \ln \Gamma[n-y+(1-\mu)/\sigma]}{\ln \Gamma(n+1/\sigma) \ln \Gamma(\mu/\sigma) \ln \Gamma[(1-\mu)/\sigma]}. \quad (5.25)$$

JAGS code may be given for the log-likelihood below, for which the logit inverse link is used. This log-likelihood may be converted to a probit, complementary loglog, or loglog beta-binomial by using the desired inverse link function. The remainder of the log-likelihood function remains the same. Simply amending one line changes the model being used.

Code 5.30 Beta-binomial distribution in R.

```
=====
mu[i] <- 1/(1+exp(-eta[i])) # inverse logit link
L1 <- loggam(n+1) - loggam(Y[i]+1) - loggam(n-Y[i]+1)
L2 <- loggam(1/sigma) + loggam(Y[i]+mu[i]/sigma)
L3 <- loggam(n-Y[i]+(1-mu[i])/sigma) - loggam(n+1/sigma)
L4 <- -loggam(mu[i]/sigma) - loggam((1-mu[i])/sigma)
LL[i] <- L1[i] + L2[i] + L3[i] + L4[i]
=====
```

Beta-Binomial model in R using JAGS

For an example of the beta-binomial model using JAGS we use a synthetic data set constructed like the one we used to illustrate the binomial model, except that we use the parameter values of the beta-binomial distribution to create the data.

Code 5.31 Simulated beta-binomial data in R.

```
=====
# Required packages
require(R2jags)
require(boot)
require(VGAM)

# Simulation
set.seed(33559)
nobs = 2500
m = 1 + rpois(nobs,5)
x1 = runif(nobs)

beta1 <- -2
beta2 <- -1.5
eta <- beta1 + beta2 * x1
sigma <- 20

p <- inv.logit(eta)
shape1 = sigma*p
shape2 = sigma*(1 - p)

y<-rbetabinom.ab(n=nobs, size=m, shape1=shape1, shape2=shape2)
bindata=data.frame(y=y,m=m,x1)
=====
```

Now we use JAGS to develop a Bayesian beta-binomial model with parameter values that closely resemble the values we assigned to the simulated data. We base this model on the values given to the two beta-binomial parameters, `beta1` (`- 2.0`) and `beta2` (`- 1.50`), and `sigma` (`20`).

Code 5.32 Beta-binomial synthetic model in R using JAGS.

```
=====
x <- model.matrix(~ x1, data = bindata)
K <- ncol(x)
model.data <- list(Y = bindata$y,
                     N = nrow(bindata),
                     X = X,
                     K = K,
                     m = m )

sink("GLOGIT.txt")
cat("
model{
# Diffuse normal priors betas
for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001)}

# Prior for theta
sigma~dgamma(0.01,0.01)

for (i in 1:N){
Y[i] ~ dbin(p[i],m[i])
p[i] ~ dbeta(shape1[i],shape2[i])
shape1[i] <- sigma*pi[i]
shape2[i] <- sigma*(1-pi[i])
logit(pi[i]) <- eta[i]
eta[i] <- inprod(beta[],X[i,])
}
},
",fill = TRUE)
sink()

inits <- function () {
list(
  beta = rnorm(K, 0, 0.1)
```

```

        )
params <- c("beta", "sigma")

BBIN <- jags(data = model.data,
               inits = inits,
               parameters = params,
               model.file = "GLOGIT.txt",
               n.thin = 1,
               n.chains = 3,
               n.burnin = 3000,
               n.iter = 5000)
print(BBIN, intervals=c(0.025, 0.975), digits=3)
=====
      mu.vect  sd.vect    2.5%   97.5%   Rhat  n.eff
beta[1]     -1.948   0.063   -2.075   -1.827  1.008  1500
beta[2]     -1.595   0.129   -1.833   -1.322  1.004   950
sigma       20.539   3.528   14.671   28.577  1.013   380
deviance    3337.284  74.220  3188.404  3477.857  1.011   380
pD = 2741.5 and DIC = 6078.8

```

The results are close to the values -2 and -1.5 we gave to the simulated beta-binomial parameters. The value of `sigma` that we assigned (20) is closely estimated as 20.539.

As we did for the binomial model, we provide the same synthetic data set for modeling the beta-binomial data grouped in standard format. Instead of using `p[i]` for the predicted probability, we shall use `mu[i]` for the beta-binomial models. Recall, though, that they symbolize the same thing – the fitted value. Again, we repeat the display of the binomial data for convenience.

Code 5.33 Explicitly given beta-binomial data in R.

```

=====
y <- c(6,11,9,13,17,21,8,10,15,19,7,12)
m <- c(45,54,39,47,29,44,36,57,62,55,66,48)
x1 <- c(1,1,1,1,1,0,0,0,0,0,0,0)
x2 <- c(1,1,0,0,1,1,0,0,1,1,0,0)
x3 <- c(1,0,1,0,1,0,1,0,1,0,1,0)
bindata <- data.frame(y,m,x1,x2,x3)
=====
```

The JAGS code used to determine the beta-binomial parameter means, standard deviations, and credible intervals for the above data is found in Code 5.34 below. Note that the line `mu[i] <- 1/(1+exp(-eta))` is equivalent to `logit(mu[i]) <- max(-20, min(20, eta[i]))`. The second expression, however, sets limits on the range of predicted values allowed by the inverse logistic link. When the zero trick is used then the log-likelihood of the distribution underlying the model must be used. The beta-binomial log-likelihood is provided over four lines of code. The line above the start of the log-likelihood defines the link that will be used with the model, which is the logistic link for this model.

Code 5.34 Beta-binomial model (in R using JAGS) for explicitly given data and the zero trick.

```

=====
library(R2jags)
X <- model.matrix(~ x1 + x2 + x3, data = bindata)
K <- ncol(X)
model.data <- list(Y = bindata$y,
                     N = nrow(bindata),
                     X = X,
                     K = K,
                     m = m,
                     Zeros = rep(0, nrow(bindata)))
}
sink("BBL.txt")
cat("model{
```

```

# Diffuse normal priors betas
for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001)}

# Prior for sigma
sigma ~ dunif(0, 100)

C <- 10000
for (i in 1:N){
  Zeros[i] ~ dpois(Zeros.mean[i])
  Zeros.mean[i] <- -LL[i] + C

#   mu[i] <- 1/(1+exp(-eta[i]))      can use for logit(mu[i]) below
  logit(mu[i]) <- max(-20, min(20, eta[i]))
  L1[i] <- loggam(m[i]+1) - loggam(Y[i]+1) - loggam(m[i]-Y[i]+1)
  L2[i] <- loggam(1/sigma) + loggam(Y[i]+mu[i]/sigma)
  L3[i] <- loggam(m[i] - Y[i]+(1-mu[i])/sigma) - loggam(m[i]+1/sigma)
  L4[i] <- loggam(mu[i]/sigma) + loggam((1-mu[i])/sigma)
  LL[i] <- L1[i] + L2[i] + L3[i] - L4[i]
  eta[i] <- inprod(beta[], X[,i])
}
}

",fill = TRUE)
sink()

inits <- function () {
  list(
    beta = rnorm(K, 0, 0.1)
  )
}
params <- c("beta", "sigma")

BBIN0 <- jags(data = model.data,
                inits = inits,
                parameters = params,
                model.file = "BBL.txt",
                n.thin = 3,
                n.chains = 3,
                n.burnin = 10000,
                n.iter = 15000)
print(BBIN0, intervals=c(0.025, 0.975), digits=3)
=====

      mu.vect    sd.vect    2.5%    97.5%     Rhat  n.eff
beta[1]    -1.225    0.425   -2.077   -0.385  1.002  1800
beta[2]     0.235    0.469   -0.733    1.174  1.001  3100
beta[3]     0.436    0.453   -0.475    1.340  1.005  520
beta[4]    -0.241    0.437   -1.119    0.624  1.002  1300
sigma      0.103    0.074    0.021    0.293  1.003  970
deviance  240078.941   4.277  240073.080  240089.421  1.000    1

pD = 9.1 and DIC = 240088.1

```

In Code 5.34 above we provide a beta-binomial model using the inverse link for sigma. The log link was used in the above parameterization of the beta-binomial. Note that we do not use the zero trick this time, which facilitates speed.

Code 5.35 Beta-binomial model with inverse link in R using JAGS.

```

=====
library(R2jags)

X <- model.matrix(~ x1 + x2 + x3, data = bindata)
K <- ncol(X)
glogit.data <- list(Y = bindata$y,
                      N = nrow(bindata),
                      X = X,
                      K = K,
                      m = m)

sink("BBI.txt")
cat("
model{
# Diffuse normal priors betas
for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001)}

# Prior for sigma
sigma~ dgamma(0.01,0.01)
"
)
```

```

for (i in 1:N){
  Y[i] ~ dbin(p[i],m[i])
  p[i] ~ dbeta(shape1[i],shape2[i])
  shape1[i] <- sigma*pi[i]
  shape2[i] <- sigma*(1-pi[i])
  logit(pi[i]) <- eta[i]
  eta[i] <- inprod(beta[],X[i,])
}
}",fill = TRUE)

sink()

# Determine initial values
inits <- function () {
  list(beta = rnorm(K, 0, 0.1))}

# Identify parameters
params <- c("beta", "sigma")

BB1 <- jags(data = glogit.data,
             inits = inits,
             parameters = params,
             model.file = "BBI.txt",
             n.thin = 1,
             n.chains = 3,
             n.burnin = 6000,
             n.iter = 10000)

print(BB1, intervals=c(0.025, 0.975), digits=3)
=====
      mu.vect    sd.vect   2.5%   97.5%   Rhat  n.eff
beta[1]   -1.251     0.373  -2.006  -0.491  1.007   330
beta[2]    0.241     0.422  -0.611   1.079  1.003  1800
beta[3]    0.447     0.421  -0.378   1.297  1.005  550
beta[4]   -0.259     0.395  -1.050   0.543  1.001 10000
sigma    20.972    17.863   4.623   65.326  1.002  1500
deviance  60.661     5.321  52.438  72.992  1.001 12000

pD = 14.2 and DIC = 74.8

```

The values closely approximate those given in the results of the initial model.

Beta–Binomial Model using Stan

Code 5.36 Beta–binomial model with synthetic data in Python using Stan.

```

=====
import numpy as np
import statsmodels.api as sm
import pystan

from scipy.stats import uniform, poisson, binom
from scipy.stats import beta as beta_dist

# Data
np.random.seed(33559)                      # set seed to replicate example
nobs = 4000                                    # number of obs in model
m = 1 + poisson.rvs(5, size=nobs)
x1 = uniform.rvs(size=nobs)                  # random uniform variable

beta0 = -2.0
beta1 = -1.5

eta = beta0 + beta1 * x1
sigma = 20

p = np.exp(eta) / (1 + np.exp(eta))
shape1 = sigma * p
shape2 = sigma * (1-p)

# Binomial distribution with p ~ beta
y = binom.rvs(m, beta_dist.rvs(shape1, shape2))

mydata = []
mydata['K'] = 2
mydata['X'] = sm.add_constant(np.transpose(x1))

```

```

mydata['N'] = nobs
mydata['Y'] = y
mydata['m'] = m

# Fit
stan_code = """
data{
    int<lower=0> N;
    int<lower=0> K;
    matrix[N, K] X;
    int Y[N];
    int m[N];
}
parameters{
    vector[K] beta;
    real<lower=0> sigma;
}
transformed parameters{
    vector[N] eta;
    vector[N] pi;
    vector[N] shape1;
    vector[N] shape2;

    eta = X * beta;
    for (i in 1:N){
        pi[i] = inv_logit(eta[i]);
        shape1[i] = sigma * pi[i];
        shape2[i] = sigma * (1 - pi[i]);
    }
}
model{
    Y ~ beta_binomial(m, shape1, shape2);
}
"""
fit = pystan.stan(model_code=stan_code, data=mydata, iter=7000, chains=3,
                   warmup=3500, n_jobs=3)

# Output
nlines = 8
output = str(fit).split('\n')
for item in output[:nlines]:
    print(item)
=====
      mean   se_mean     sd   2.5%   25%   50%   75%   97.5%   n_eff   Rhat
beta[0]  -2.07  1.1e-3  0.05  -2.17  -2.1  -2.06  -2.03  -1.97  2136.0   1.0
beta[1]  -1.44  2.3e-3  0.11  -1.65  -1.51  -1.44  -1.37  -1.22  2102.0   1.0
sigma    17.56    0.04  2.16   13.9  16.02  17.38  18.88  22.31  2394.0   1.0

```

Beta–Binomial Regression – Probit Link in JAGS

Code 5.32 for the beta–binomial model with a canonical logit link may be amended to produce a beta–binomial model with a probit link.

For the first parameterization (in Code 5.32), where we use beta–binomial parameters to generate synthetic data, we simply substitute

```
logit(pi[i]) <- eta[i]
```

by

```
probit(pi[i]) <- eta[i]
```

in the code. The results are

	mu.vect	sd.vect	2.5%	97.5%	Rhat	n.eff
beta[1]	-1.159	0.035	-1.227	-1.091	1.037	60
beta[2]	-0.771	0.068	-0.898	-0.635	1.024	100

```

sigma      20.797     3.221    15.646   28.141   1.026     150
deviance  3343.224   67.743  3213.654 3478.579   1.014     220

```

pD = 2274.9 and DIC = 5618.2

For the second model, where we provide grouped data to the JAGS code, we do exactly the same as we did for the grouped binomial model in the previous section. All that needs to be done is to change

```
logit(mu[i]) <- max(-20, min(20, eta[i]))
```

to

```
probit(mu[i]) <- max(-20, min(20, eta[i]))
```

The probit–beta binomial results are

	mu.vect	sd.vect	2.5%	97.5%	Rhat	n.eff
beta[1]	-0.736	0.255	-1.238	-0.206	1.001	5000
beta[2]	0.136	0.276	-0.415	0.677	1.001	5000
beta[3]	0.250	0.282	-0.337	0.786	1.001	5000
beta[4]	-0.129	0.262	-0.641	0.394	1.001	5000
sigma	0.109	0.083	0.021	0.327	1.002	1300
deviance	240079.176	4.510	240072.968	240090.221	1.000	1

pD = 10.2 and DIC = 240089.3

Beta–binomial complementary loglog and loglog links can also be created from the above code by amending the same line as that used for a probit link. The inverse link functions provided in Table 5.1 can be used. The second model we gave can also be converted to a probit model by changing the line `logit(pi[i]) <- eta[i]` to the line `probit(pi[i]) <- eta[i]`.

Beta–Binomial Regression – Probit Link in Stan

In Code 5.36 this is equivalent to substituting

```
pi[i] = inv_logit(eta[i]);
```

by

```
pi[i] = Phi(eta[i]);
```

which results in

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
beta[0]	-1.22	5.9e-4	0.03	-1.28	-1.24	-1.22	-1.21	-1.17	2048.0	1.0
beta[1]	-0.68	1.1e-3	0.05	-0.78	-0.72	-0.68	-0.65	-0.58	2010.0	1.0
sigma	17.54	0.05	2.2	13.77	16.0	17.36	18.86	22.38	2274.0	1.0

This concludes the section on binomial models, including Bernoulli, full binomial, and beta–binomial combination models. In Chapter 6 we shall discuss hierarchical or generalized linear mixed-effect models. Some statisticians use the terms “random intercept” and “random slopes” to refer to these models. However, we next turn, in Chapter 6, to presenting an overview of Bayesian count models. We discuss the standard Bayesian Poisson count model, as well as the Bayesian negative

binomial, generalized Poisson, zero-truncated, and three-parameter NB-P models. This discussion is followed by an examination of Bayesian zero-inflated mixture models and two-part hurdle models.

¹ This code uses snippets from <http://people.duke.edu/~ccc14/sta-663/PyMC3.html>.

¹ The *odds of success* is defined as the ratio of the probability of success and the probability of failure. The *odds ratio* is the ratio of the odds of success for two different classes.

6 GLMs Part II – Count Models

Count data refer to observations made about events or enumerated items. In statistics they represent observations that have only non-negative integer values, for example, the number of globular clusters in a galaxy or the number of exoplanets orbiting a stellar system. Examples of discrete count models discussed in this chapter are displayed in Table 6.1.

Table 6.1 Examples of discrete count models.

Name of model	description
Poisson	Basic count model mean and variance are assumed identical
Negative binomial	Two-parameter used to model overdispersed Poisson data
Generalized Poisson	Two-parameter used to model under and overdispersed count data
Zero-truncated count	Model used when no zeros can exist in the data
NB-P	Three-parameter negative binomial model
Zero-inflated	Model used when there are excessive zeros in the data
Two-part hurdle	Models the zero counts separately from the counts

A count response consists of any discrete number of counts: for example, the number of hits recorded by a Geiger counter, patient days in a hospital, or sunspots appearing per year. All count models aim to explain the number of occurrences, or counts, of an event. The counts themselves are intrinsically heteroskedastic, i.e., right skewed, and have a variance that increases with the mean of the distribution. If the variance is greater than the mean, the model is said to be overdispersed; if less than the mean, the model is said to be underdispersed; the term “extra-dispersed” includes both these possibilities. The Poisson model is assumed to be equidispersed, with the mean equal to the variance. Violations of this assumption in a Poisson model lead to biased standard errors. Models other than the Poisson are aimed to adjust for what is causing the data to be under- or overdispersed. Refer to [Hilbe \(2014\)](#) for a full text on the modeling of count data.

6.1 Bayesian Poisson Models

The Poisson model is the traditional standard model to use when modeling count data. Counts are values ranging from 0 to a theoretical infinity. The important thing to remember is that count data are discrete. A count response variable modeled using a Poisson regression model (for frequency-based estimation) or a Poisson model (for Bayesian estimation) consists of zero and positive integers. The Poisson probability distribution may be expressed as

$$f(y; \mu) = \frac{e^{-\mu_i} \mu_i^{y_i}}{y_i!} \quad (6.1)$$

and the Poisson log-likelihood as

$$\mathcal{L} = \sum_{i=1}^n \{y_i \ln(\mu_i) - \mu_i - \ln(y_i)\} \quad (6.2)$$

with $\mu_i = \exp(x_i \beta)$, where $x_i \beta$ is the linear predictor for each observation in the model. The general expression in Equation 5.1 then takes the form

$$\begin{aligned} Y_i &\sim \text{Poisson}(\mu_i); \\ \mu_i &= e^{\eta_i}, \\ \eta_i &= \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p. \end{aligned}$$

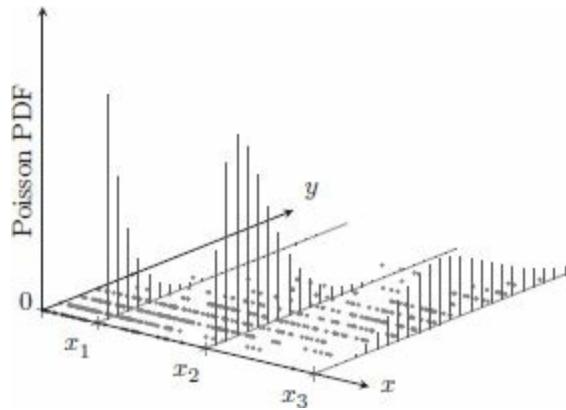


Figure 6.1 Illustration of Poisson-distributed data.

When the data are formatted as integers, it is not appropriate to model them using normal, lognormal, or other continuous-response models. Prior to when Poisson regression became available in commercial statistical software, analysts faced with modeling positive count data logged the counts, usually adding 0.5 to their value so that zero count values would not end up with missing values. Of course, doing this changes the data and, most importantly, violates the distributional assumptions of the normal model. Logging the response variable and modeling it with a normal or

linear regression still corresponds to a normal model – and is based on various important assumptions.

Probably the main reason not to log a count variable and model it using ordinary least squares regression (OLS) is that in linear regression it is assumed that the variance of the counts remains constant throughout the range of count values. This is unrealistic and rarely if ever occurs with real data. Nevertheless, when the model is estimated using a GLM-based normal or linear regression, the variance is set to 1. When estimated as a two-parameter model the normal model estimates both the mean and variance, with the variance having a constant value for all observations. In fact, in count data the variance nearly always varies with the mean, becoming larger as the mean becomes higher. To model a count variable as if it were continuous violates the distributional assumptions of both the normal and count model distributions. Typically the standard errors become biased, as well as the predicted values. Also, goodness-of-fit statistics such as the AIC or BIC are generally substantially higher for count data estimated using OLS rather than a standard count model.

We shall first provide a simple example of synthetic Poisson data with a corresponding Poisson model. In the model below we create a 500-observation data set. We provide an intercept value of 1, and a predictor x , with parameter value 2. We have combined lines by placing the inverse link transformation within the Poisson random number generator rather than on a separate line before it. We have also encased the regression within the summary function to save another step. We can further simplify the code so that the synthetic data is created on two lines; this is displayed below the following *simple* model.

Code 6.1 Synthetic data following a Poisson distribution in R.

```
=====
set.seed(2016)
nobs <- 500
x <- runif(nobs)
xb <- 1 + 2*x
py <- rpois(nobs, exp(xb))
summary(myp <- glm(py ~ x, family=poisson))
=====
```

Coefficients:

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	1.00106	0.04037	24.80	<2e-16 ***
x	2.02577	0.05728	35.37	<2e-16 ***

Null deviance: 1910.87 on 499 degrees of freedom
Residual deviance: 540.07 on 498 degrees of freedom
AIC: 2428.8

A more concise coding is as follows:

```
set.seed(2016)
x <- runif(500)
py <- rpois(500, exp(1+2*x))
summary(myp <- glm(py ~ x, family=poisson))
```

The result displayed is identical to that for the first version. Although it is nearly always possible to shorten code, even though it works it may be challenging to decipher: The problem is that when you come back to the code later, you may find it difficult to interpret. It is better practice to write your code in a clear and understandable manner, including comments throughout to identify exactly what the code is for.

Given the above, we now create a more complex synthetic data set. The goal is to demonstrate how synthetic binary and continuous predictors can be used to simulate real data. In this case we format the synthetic data so that it has two predictors: a binary predictor x_{1_2} and a random normal continuous predictor x_2 , as shown in Figure 6.2.

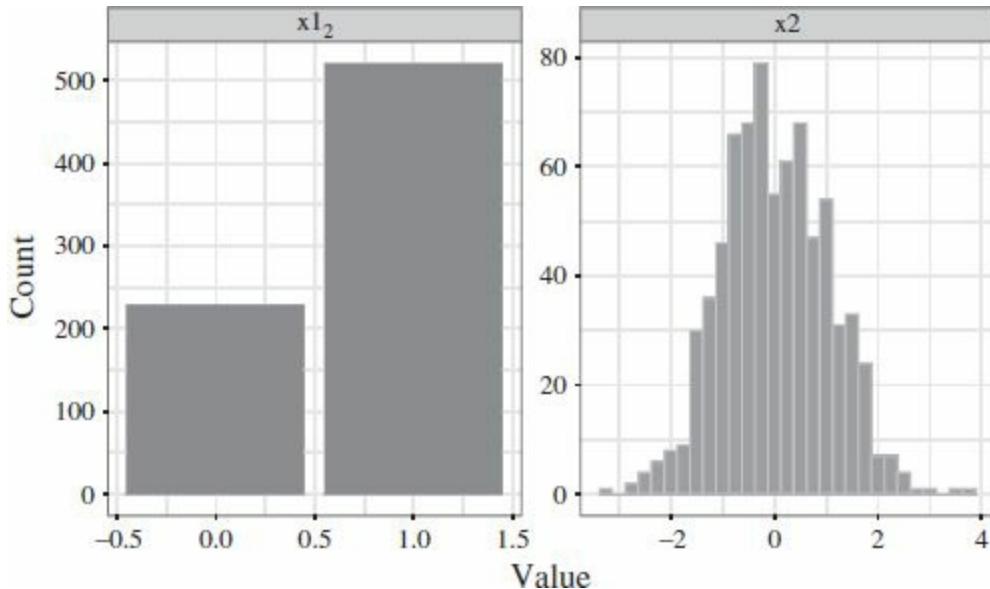


Figure 6.2 Binary and continuous predictors for the Poisson model.

Code to generate the data is given below:

Code 6.2 Synthetic Poisson data and model in R: binary and continuous predictors.

```
=====
set.seed(18472)
nobs <- 750
x1_2 <- rbinom(nobs, size=1, prob=0.7) # 70% 1's; 30% 0's
x2 <- rnorm(nobs, 0, 1)
xb <- 1 - 1.5*x1_2 - 3.5*x2
exb <- exp(xb)
py <- rpois(nobs, exb)
pois <- data.frame(py, x1_2, x2)
poi <- glm(py ~ x1_2 + x2, family=poisson, data=pois)
summary(poi)
=====
```

Coefficients:

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	1.003426	0.010634	94.36	<2e-16 ***
x1_2	-1.507078	0.004833	-311.81	<2e-16 ***
x2	-3.500726	0.004228	-828.00	<2e-16 ***

Null deviance: 1821132.15 on 749 degrees of freedom
Residual deviance: 598.59 on 747 degrees of freedom
AIC: 2439.6

The resulting model values are quite close to what we specified; considering that the data has only 750 observations, they are in fact very close. The coefficients we have for this data are what we use for comparison with a Bayesian model of the data. The data is not identified with the values we initially specified, but rather with the coefficients and parameter values actually created in the

sampling algorithm we used. If we employed larger data sets, the values would be closer to what we specified.

An important distributional assumption of the Poisson model is that the mean and variance of the count variable being modeled are the same, or at least almost the same. Various tests have been designed to determine whether this criterion has been met. Recall that it is not sufficient simply to tabulate the raw count variable; rather, it must be tabulated as adjusted by its predictors. This is not easy to do with models on real data, but is not difficult with synthetic data.

An easy way to determine whether the variance is greater than the mean is to check the Poisson model dispersion statistic. If the statistic is greater than 1.0, the model is said to be overdispersed, that is, there is more variability in the data than is allowed by the Poisson assumption (variance $>$ mean). If the dispersion statistic is less than 1.0, the model is said to be underdispersed (mean $>$ variance). Most real count data is overdispersed, which biases the model's standard errors: a predictor may appear to contribute significantly to the model when in fact it does not. The greater the dispersion, the greater the bias. Such a Poisson model should not be relied on to provide meaningful parameter estimates. The standard way to model Poisson overdispersed data is by using a negative binomial model, which has an extra dispersion parameter that is intended to adjust for the overdispersion. We discuss the negative binomial model in the following section.

Underdispersion is fairly rare with real data, but it does happen when by far the largest number of observations in the data have low count values. A negative binomial model cannot be used with Poisson underdispersed data; however, underdispersion may be adjusted for by using a generalized Poisson model. The generalized Poisson can also model Poisson overdispersed data. We shall discuss the generalized Poisson later in this chapter.

It is probably wise to use a Poisson model first when modeling count data, i.e., a model with a count response variable. Then one checks the Pearson dispersion statistic to determine whether the model is under- or overdispersed. If the dispersion statistic is close to 1 then a Poisson model is appropriate. A boundary likelihood ratio test when one is using a negative binomial model can provide a p-value to determine whether a count model is Poisson. We shall deal with that as well, in the next section.

When using R's `glm` function for a Poisson model, the output does not include the dispersion statistic. The `glm` function in the major commercial packages provides a dispersion statistic as part of the default output. It is easy, however, to calculate the dispersion statistic ourselves when using R. The dispersion statistic is the ratio of the Pearson χ^2 statistic and the model's residual degrees of freedom. We can calculate the Pearson χ^2 statistic as the sum of squared Pearson residuals. The number of residual degrees of freedom is calculated as the number of observations in the model less the number of parameters, which includes the intercept. For the above model the dispersion statistic may be determined using the following code:

```
pr <- resid(poi, type="pearson")
N <- nrow(pois)
p <- length(coef(poi))
sum(pr^2) / (N - p)
[1] 1.071476
```

The COUNT package on CRAN has a function that automatically calculates the Pearson χ^2 statistic and dispersion following `glm`. It is called `P_disp()` (two underscores). It can be used on our data as

```
library(COUNT)
P_disp(poi)
pearson.chi2    dispersion
 800.392699    1.071476
```

We can also direct the `glm` function to display a Pearson dispersion characteristic by using the following code:

```
> poi <- glm(py ~ x1_2 + x2, family=poisson, data=pois)
> disp <- sum(residuals(poi, type="pearson")^2/poi$df.res)
> summary(poi, dispersion=disp)
```

The output, which is identical to a quasipoisson model owing to the scaling of standard errors by the dispersion, displays the following relevant line:

```
(Dispersion parameter for poisson family taken to be 1.071476)
```

As an aside, recall that in the section on Bernoulli logistic models we mentioned the scaling of standard errors (in maximum likelihood models) or of standard deviation statistics (in Bayesian posterior distributions). Scaling is used perhaps more with Poisson and negative binomial models than with any other model. If there is evidence of extra-dispersion in the otherwise Poisson data, and we cannot identify its source, many statisticians scale standard errors by multiplying the standard error by the square root of the dispersion statistic. It is important for count models to base the dispersion on the Pearson χ^2 statistic and not on the deviance, which would bias the true dispersion upward. The standard advice to use robust or sandwich-adjusted standard errors for all count models is only applicable for maximum likelihood estimated models. Scaling may be used on Bayesian Poisson models, but we advise against it. If the source of Poisson overdispersion is unknown then we advise employing a negative binomial model, as discussed in the following section. If the Poisson data is underdispersed, and the source of the underdispersion is unknown, we advise using a generalized Poisson model, which we address later in this chapter. In R, one may scale maximum likelihood Poisson standard errors by using the `glm` family option `quasipoisson`. This family consists of nothing more than scaling, although the `glm` model documentation does not mention this.

It is also instructive to check the range of the count variable in our model by using the `summary` function. Here we find that values of the response range from 0 to 63920, the median and mean values being quite different. Because the mean is so much greater than the median, we know that the data is likely to be highly skewed to the right, which is not untypical of count data.

```
summary(py)
Min. 1st Qu. Median Mean 3rd Qu. Max.
0.0    0.0    1.0  307.8   11.0   63920.0
```

A tabulation of the counts allows us to determine whether there are any counts or sections of counts without values, thus indicating the possible need for a two-part model. It is important to check a tabulation table of counts to determine whether there are zero counts and, if so, whether

there are more than are allowed on the basis of the Poisson distribution:

```
barplot(table(py), log="y", col="gray75")
```

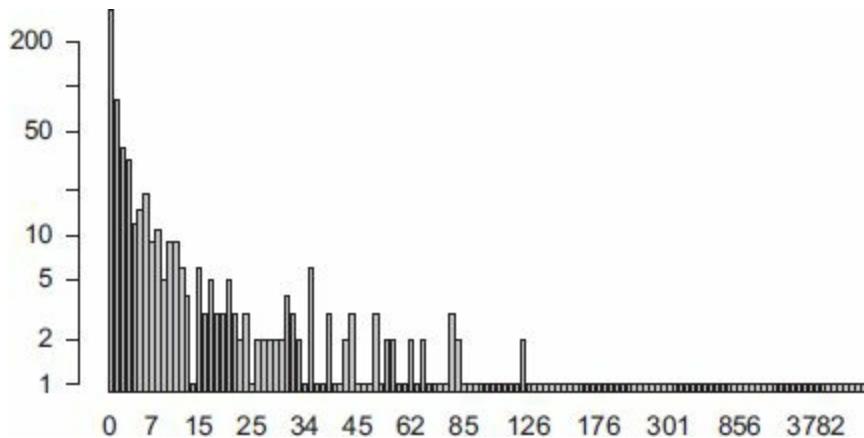


Figure 6.3 Frequency data for the Poisson model.

To determine the percentage of zeros that theoretically there should be in a Poisson model with a given mean, we use the expression

$$P(Y_i = 0|\mu_i) = \frac{\mu_i^0 \times e^{-\mu_i}}{0!} = e^{-\mu_i}; \quad (6.3)$$

thus the probability of zeros for the data set above can be estimated with the following command:

```
P_zeros = mean(exp(-xb))
P_zeros
[1] 0.4461361
```

We convert this value to zero counts with the data we are modeling:

```
> nobs(poi) * P_zeros
[1] 334.6021
```

There are in fact `sum(py==0) = 329` zero counts in `py`, which is very close to the expected value. In cases where a Poisson model expects many fewer zeros than are actually observed in the data, this might indicate a problem when modeling the data as Poisson; then it may be better to use a zero-inflated model or a two-part hurdle model on this data, as will be discussed later in this book.

Having excessive zero counts in the response variable being modeled nearly always leads to overdispersion. The primary way in which statisticians model overdispersed Poisson data is by using a negative binomial, a hurdle, or a zero-inflated model in place of Poisson. This is true for Bayesian models as well. Recent work on which model best fits Poisson overdispersed data has been leaning towards the use of a negative binomial or hurdle model, but it really depends on the source of the overdispersion. There is no single catch-all model to use for overdispersed count data. However,

employing the Bayesian negative binomial model or other advanced Bayesian count models introduced in this volume greatly expands our ability to model overdispersed count data appropriately.

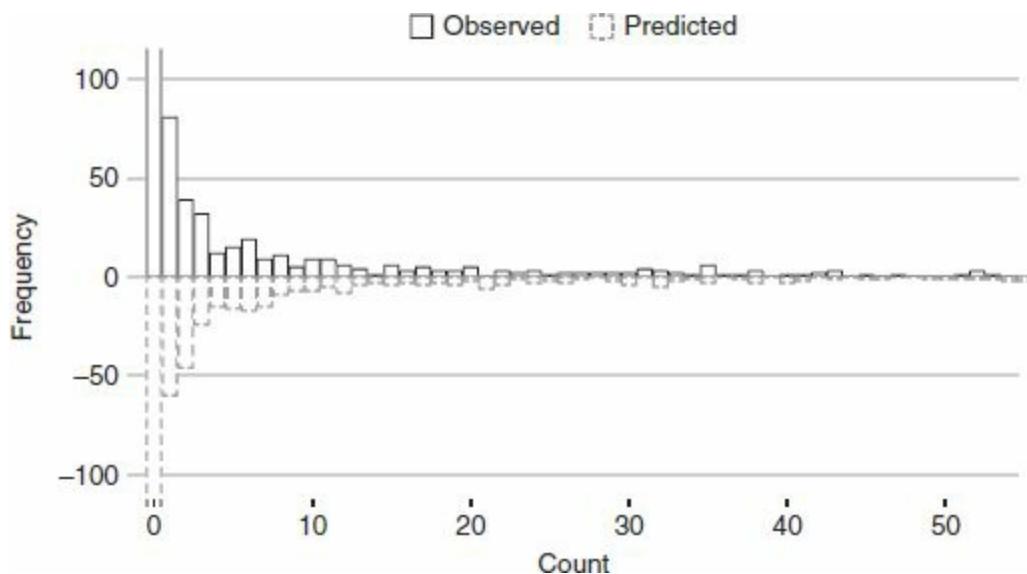


Figure 6.4 Observed vs. predicted count frequency per cell for the Poisson model.

We can add that there are still other models and adjustments that can be made in case of over- or underdispersion. The remaining models discussed in this chapter can be used for this purpose.

6.1.1 Poisson Models with R

When we discussed the Bayesian modeling of normal data in Chapter 3, the `MCMCpack` library was used to estimate model parameters. We turn to this library again for implementing a Bayesian Poisson model. The built-in function for this purpose is called `MCMCpoisson`, which is set up in the same manner as `MCMCregress`. We shall use the default diffuse or so-called “non-informative” priors on the parameters in order to maximize the model data in the estimation process. Recall that the resulting posterior distributions provide the mean, standard deviation, and credible intervals of each parameter in the model. For the predictor parameters, these are analogous to maximum likelihood coefficients, standard errors, and confidence intervals. Other maximum likelihood parameters have standard errors and confidence intervals associated with them, but they are not given much attention. In Bayesian modeling they are.

The data that was created in Code 6.2 in Section 6.1.1 will be used with the `MCMCpoisson` function, as we have done before. To reiterate, we expect that the parameter means and standard deviations will approximate the Poisson model coefficients and standard errors. Bayesian standard deviations are, however, typically smaller than the standard errors produced in MLE for the same data, particularly in large and more complex models. Here they should be approximately the same.

```
=====
library(MCMCpack)

mypoisL <- MCMCpoisson(py ~ x1_2 + x2,
  burnin = 5000,
  mcmc = 10000,
  data = pois)
summary(mypoisL)
plot(mypoisL)
=====
```

	Mean	SD	Naive SE	Time-series SE
(Intercept)	1.003	0.010645	3.366e-05	1.120e-04
x1_2	-1.507	0.004807	1.520e-05	5.136e-05
x2	-3.501	0.004226	1.336e-05	4.439e-05

Quantiles for each variable:

	2.5%	25%	50%	75%	97.5%
(Intercept)	0.9823	0.9962	1.003	1.011	1.024
x1_2	-1.5165	-1.5103	-1.507	-1.504	-1.498
x2	-3.5091	-3.5036	-3.501	-3.498	-3.493

The parameter means and standard deviations are in fact nearly identical to the Poisson regression coefficients and standard errors. The plot function generated with this code will display histograms that appear normally distributed, as we expect, with the distributional mode centered over the estimated parameter mean. The trace plots can confirm that there are no evident problems with convergence. The output plots are not shown here but the reader can find them in the online material.

6.1.2 Poisson Models with JAGS

The same data as that created in Section 6.1 is used for our JAGS Poisson code. Note that we use the same general format for this model as we have for previous models. The Poisson model is, though, a very simple model with only intercept and predictor posterior parameters to be estimated.

Code 6.4 Bayesian Poisson model in R using JAGS.

```
=====
require(R2jags)

X <- model.matrix(~ x1_2 + x2, data = pois)
K <- ncol(X)

model.data <- list(Y = pois$py,
  X = X,
  K = K, # number of betas
  N = nrow(pois)) # sample size

sink("Poi.txt")
cat("
model{
  for (i in 1:K) {beta[i] ~ dnorm(0, 0.0001)}

  for (i in 1:N) {
    Y[i] ~ dpois(mu[i])
    log(mu[i]) <- inprod(beta[], X[i,])
  }

  }", fill = TRUE)
sink()

inits <- function () {
  list(beta = rnorm(K, 0, 0.1))}
params <- c("beta")

POI <- jags(data = model.data,
  inits = inits,
  parameters = params,
```

```

model = "Poi.txt",
n.thin = 1,
n.chains = 3,
n.burnin = 4000,
n.iter = 5000)
print(POI, intervals=c(0.025, 0.975), digits=3)
=====
      mu.vect   sd.vect    2.5%    97.5%     Rhat n.eff
beta[1]    1.005    0.028    0.983    1.025  1.001 3000
beta[2]   -1.507    0.008   -1.517   -1.497  1.001 2400
beta[3]   -3.500    0.012   -3.509   -3.492  1.001 3000
deviance  2444.016  169.683  2433.861  2443.870 1.001 3000

pD = 14405.7 and DIC = 16849.7

```

The JAGS results are nearly identical to those for the maximum likelihood (GLM) and MCMCpoisson models run on the data. We used only 1000 sampling iterations to determine the mean and standard deviation values. Four thousand samples were run but were discarded as burn-in samples.

6.1.3 Poisson Models in Python

Code 6.5 Simple Poisson model in Python.

```

import numpy as np
from scipy.stats import uniform, poisson
import statsmodels.api as sm

# Data
np.random.seed(2016)                      # set seed to replicate example
nobs = 1000                                 # number of obs in model

x = uniform.rvs(size=nobs)

xb = 1 + 2 * x                               # linear predictor
py = poisson.rvs(np.exp(xb))                 # create y as adjusted
X = sm.add_constant(x.transpose())           # covariates matrix

# Fit
# Build model
mfp = sm.GLM(py, X, family=sm.families.Poisson())

# Find parameter values
res = mfp.fit()

# Output
print(res.summary())
=====

Generalized Linear Model Regression Results
=====
Dep. Variable:                  y      No. Observations:      1000
Model:                          GLM      Df Residuals:          998
Model Family:                  Poisson      Df Model:               1
Link Function:                  log      Scale:                  1.0
Method:                          IRLS      Log-Likelihood: -2378.8
Date:              Tue, 21 Jun 2016      Deviance:             985.87
Time:                  12:05:03      Pearson chi2:        939.0
No. Iterations:                   8

            coef    std err          z     P>|z| [95.0% Conf. Int.]
const    0.9921    0.029    34.332      0.000      0.935      1.049
x1      2.0108    0.041     48.899      0.000      1.930      2.091

```

Code 6.6 Multivariate Poisson model in Python

```

x1_2 = binom.rvs(1, 0.7, size=nobs)
x2 = norm.rvs(loc=0, scale=1.0, size=nobs)

xb = 1 - 1.5 * x1_2 - 3.5 * x2      # linear predictor
exb = np.exp(xb)
py = poisson.rvs(exb)                 # create y as adjusted

my_data = {}
my_data['x1_2'] = x1_2
my_data['x2'] = x2
my_data['py'] = py

# Build model
mfp = smf.glm('py ~ x1_2 + x2', data=my_data, family=sm.families.Poisson())

# Find parameter values
res = mfp.fit()
# Output
print(res.summary())
=====
Generalized Linear Model Regression Results
=====
Dep. Variable:                  py   No. Observations:      750
Model:                          GLM   Df Residuals:          747
Model Family:                   Poisson   Df Model:             2
Link Function:                  log    Scale:               1.0
Method:                         IRLS   Log-Likelihood:     -1171.0
Date:              Tue, 21 Jun 2016   Deviance:            556.26
Time:                14:24:33   Pearson chi2:        643.0
No. Iterations:                  12
=====
      coeff    std err      z    P>|z|   [95.0% Conf. Int.]
Intercept    1.0021     0.012    83.174    0.000      0.979    1.026
x1_2       -1.5003     0.006   -252.682    0.000     -1.512   -1.489
x2        -3.5004     0.004   -796.736    0.000     -3.509   -3.492

```

Code 6.7 Bayesian Poisson model using pymc3.

```

=====
import numpy as np
import pandas
import pymc3 as pm

from scipy.stats import norm, binom, poisson

# Data
np.random.seed(18472)                      # set seed to replicate example
nobs = 750                                    # number of obs in model

x1_2 = binom.rvs(1, 0.7, size=nobs)
x2 = norm.rvs(loc=0, scale=1.0, size=nobs)

xb = 1 - 1.5 * x1_2 - 3.5 * x2            # linear predictor, xb
exb = np.exp(xb)
py = poisson.rvs(exb)                      # create y as adjusted

df = pandas.DataFrame({'x1_2': x1_2, 'x2': x2, 'py': py}) # rewrite data

# Fit
niter = 10000                                # parameters for MCMC

with pm.Model() as model_glm:
    # define priors
    beta0 = pm.Flat('beta0')
    beta1 = pm.Flat('beta1')
    beta2 = pm.Flat('beta2')

    # define likelihood
    mu = np.exp(beta0 + beta1 * x1_2 + beta2 * x2)
    y_obs = pm.Poisson('y_obs', mu, observed=py)

    # inference
    start = pm.find_MAP()                      # Find starting value by
                                                optimization
    step = pm.NUTS()
    trace = pm.sample(niter, step, start, progressbar=True)

# Output

```

```

pm.summary(trace)
=====
beta0:
    Mean      SD      MC Error   95% HPD interval
    -----|-----|-----|-----|
  1.002     0.027     0.000     [0.979, 1.025]

Posterior quantiles:
  2.5          25          50          75         97.5
  |-----|=====|=====|-----|-----|
  0.979     0.994     1.002     1.010     1.026

beta1:
    Mean      SD      MC Error   95% HPD interval
    -----|-----|-----|-----|
 -1.495     0.082     0.005     [-1.512, -1.489]

Posterior quantiles:
  2.5          25          50          75         97.5
  |-----|=====|=====|-----|-----|
 -1.512    -1.504    -1.500    -1.496    -1.488

beta2:
    Mean      SD      MC Error   95% HPD interval
    -----|-----|-----|-----|
 -3.498     0.079     0.003     [-3.509, -3.492]

Posterior quantiles:
  2.5          25          50          75         97.5
  |-----|=====|=====|-----|-----|
 -3.509    -3.503    -3.500    -3.497    -3.492

```

6.1.4 Poisson Models in Python using Stan

Code 6.8 Bayesian Poisson model in Python using Stan.

```

=====
import numpy as np
import pystan
import statsmodels.api as sm

from scipy.stats import norm, poisson, binom

# Data
np.random.seed(18472)                      # set seed to replicate example
nobs = 750                                     # number of obs in model

x1_2 = binom.rvs(1, 0.7, size=nobs)
x2 = norm.rvs(loc=0, scale=1.0, size=nobs)

xb = 1 - 1.5 * x1_2 - 3.5 * x2             # linear predictor
exb = np.exp(xb)
py = poisson.rvs(exb)                        # create y as adjusted

X = sm.add_constant(np.column_stack((x1_2, x2)))

mydata = {}                                    # build data dictionary
mydata['N'] = nobs                            # sample size
mydata['X'] = X                               # predictors
mydata['Y'] = py                               # response variable
mydata['K'] = mydata['X'].shape[1]              # number of coefficients

#Fit

stan_code = """
data{
    int N;
    int K;
    matrix[N,K] X;
    int Y[N];
}
parameters{
    vector[K] beta;
}
model{
    Y ~ poisson_log(X * beta);
}
"""

model = StanModel(model_code=stan_code)

```

```

}
"""

# Run mcmc
fit = pystan.stan(model_code=stan_code, data=mydata, iter=5000, chains=3,
                   warmup=4000, n_jobs=3)

# Output
print(fit)
=====
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
beta[0]	1.0	5.6e-4	0.01	0.98	0.99	1.0	1.01	1.03	508.0	1.0
beta[1]	-1.5	2.4e-4	5.9e-3	-1.51	-1.5	-1.5	-1.5	-1.49	585.0	1.0
beta[2]	-3.5	2.0e-4	4.6e-3	-3.51	-3.5	-3.5	-3.5	-3.49	505.0	1.0

6.2 Bayesian Negative Binomial Models

The negative binomial distribution is generally thought of as a mixture of the Poisson and gamma distributions. A model based on the negative binomial distribution is commonly regarded as a Poisson–gamma mixture model. However, the negative binomial distribution has been derived in a variety of ways. [Hilbe \(2011\)](#) accounts for some 13 of these methods, which include derivations as the probability of observing y failures before the r th success in a series of Bernoulli trials, as a series of logarithmic distributions, as a population growth model, and from a geometric series. Nevertheless, the now standard way of understanding the negative binomial model is as a Poisson–gamma mixture, in which the gamma scale parameter is used to adjust for Poisson overdispersion. The variance of a Poisson distribution (and model) is μ , and the variance of a two-parameter gamma model is μ^2/v . The variance of the negative binomial, as a Poisson–gamma mixture model, is therefore $\mu + \mu^2/v$. In this sense v can be regarded as a dispersion parameter. However, the majority of statisticians have preferred to define the negative binomial dispersion parameter as α , where $\alpha = 1/v$. Parameterizing the dispersion as α guarantees that there is a direct relationship between the mean and the amount of overdispersion, or correlation, in the data. The greater the overdispersion, the greater the value of the dispersion parameter. The direct form of the variance is $\mu + \alpha\mu^2$.

Keep in mind that there is a difference between the Poisson dispersion statistic discussed in the previous section and the negative binomial dispersion parameter. The dispersion statistic is the ratio of the Poisson model Pearson χ^2 statistic and the residual degrees of freedom. If the data is overdispersed, the dispersion statistic is greater than 1. The more overdispersion in the data to be modeled, the higher the value of the dispersion statistic. Values of the dispersion statistic less than 1 indicate underdispersion in the data.

As an aside, note that there are times when the Poisson dispersion statistic is greater than 1, yet the data is in fact equidispersed. This can occur when the data is apparently overdispersed. If transforming the predictor data, by for example logging or squaring a predictor, by adding a needed variable, by employing an interaction term, or by adjusting for outliers, changes the dispersion to approximately 1 then the data is not truly overdispersed – it is only apparently overdispersed and can be modeled using a Poisson model. Thus one should make certain that the model is adjusted for possible apparent overdispersion rather than concluding that it is truly overdispersed on the sole basis of a first look at the dispersion statistic.

Another item to keep in mind is that the negative binomial model itself has a dispersion statistic. It is an indicator of extra correlation or dispersion in the data over what we expect of a negative binomial model. The same corrective operations that we suggested for ameliorating apparent overdispersion for Poisson models can be used on a negative binomial model. But if the extra-dispersion persists, a three-parameter generalized negative binomial model may be used to adjust for this in a negative binomial model. Another tactic is to employ an alternative count model such as a generalized Poisson or a Poisson inverse Gaussian model. Sometimes these models fit the data better than either a Poisson or negative binomial. In any case, nearly every reference to overdispersion relates to the Poisson model. Most of the other more advanced count models that we discuss in this volume are different methods to adjust for the possible causes of overdispersion in a Poisson model.

We should also mention that all major general-purpose commercial statistical packages parameterize the negative binomial variance as having a direct relationship to the mean (or variance) and to the amount of dispersion or correlation in the data. Moreover, specialized econometric software such as `LIMDEP` and `XPORE` also employs a direct relationship. However, R's `glm` and `glm.nb` functions incorporate an indirect relationship between the mean and dispersion. The greater overdispersion in the data, the higher the dispersion statistic and lower the dispersion parameter. Using the direct relationship results in a negative binomial with $\alpha = 0$, the same as for a Poisson model. Since a true Poisson model has an identical mean and variance, it makes sense that an α value of 0 is Poisson. The greater the dispersion is above 0, the more dispersion over the basic Poisson model is indicated. For an indirect parameterization, with $\theta = 1/\alpha$, as θ approaches 0 there is infinite overdispersion, i.e., as the overdispersion reduces to Poisson, the dispersion in terms of θ approaches infinity.

For maximum likelihood models, the coefficients and standard errors are the same for both direct and indirect parameterizations. Likewise, they are the same when the negative binomial is modeled using Bayesian methodology. The difference in results between the direct and indirect parameterizations rests with the dispersion parameter and Pearson-based residuals and fit statistics. We warn readers to be aware of how a particular negative binomial function parameterizes the dispersion parameter when attempting to interpret results or when comparing results with models produced using other software.

It should be noted that the negative binomial functions in R's `gamlss` and COUNT packages are direct parameterizations, and their default results differ from `glm` and `glm.nb`. The R `nbinomial` function in the COUNT package (see the book [Hilbe and Robinson 2013](#)) has a default direct parameterization, but also an option to change results to an indirect parameterization. The `nbinomial` function also has an option to parameterize the dispersion parameter, providing predictor coefficients influencing the dispersion. The value of this is to show which predictors most influence overdispersion in the data.

In the subsection on the negative binomial model using JAGS below, we provide both parameterizations of the negative binomial. The negative binomial PDF for the direct parameterization is displayed in Equation 6.4 below. The PDF for an indirect parameterization can be obtained by inverting each α in the right-hand side of the equation. Again, the indirect dispersion parameter θ is defined as $\theta = 1/\alpha$. We use theta (θ) as the name of the indirect dispersion

because of R's `glm` and `glm.nb` use of the symbol.

The negative binomial probability distribution function can be expressed as terms of α as

$$f(y; \mu, \alpha) = \binom{y + 1/\alpha - 1}{1/\alpha - 1} \left(\frac{1}{1 + \alpha\mu} \right)^{1/\alpha} \left(\frac{\alpha\mu}{1 + \alpha\mu} \right)^y. \quad (6.4)$$

The negative binomial log-likelihood is therefore

$$\begin{aligned} \mathcal{L}(\mu; y, \alpha) &= \sum_{i=1}^n y_i \log \left(\frac{\alpha\mu_i}{1 + \alpha\mu_i} \right) - \frac{1}{\alpha} \log (1 + \alpha\mu_i) \\ &\quad + \log \Gamma \left(y_i + \frac{1}{\alpha} \right) - \log \Gamma(y_i + 1) - \log \Gamma \left(\frac{1}{\alpha} \right). \end{aligned} \quad (6.5)$$

The terms on the right-hand side of the above log-likelihood may be configured into other form. The form here is that of the exponential family. The canonical link is therefore $\log(\alpha\mu/(1 + \alpha\mu))$, and the variance, which is the second derivative of the cumulant $\log(1 + \alpha\mu)/\alpha$ with respect to the link, is $\mu + \alpha\mu^2$. Figure 6.5 gives a comparison between two data sets with similar means but one with samples from a negative binomial distribution and the other with a Poisson distribution. Note how the negative binomial data has a much wider spread.

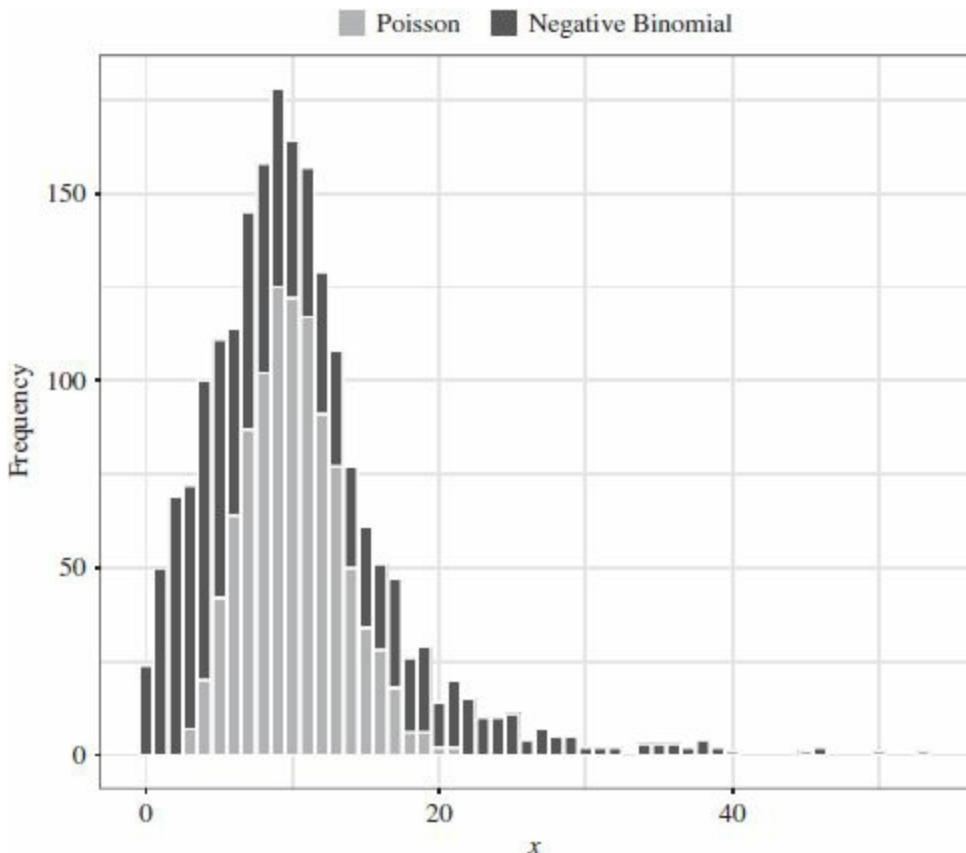


Figure 6.5 For comparison, Poisson-distributed and negative-binomial-distributed data.

The canonical form of the distribution is not used as the basis of the traditional negative binomial model. To be used as a means to adjust for Poisson overdispersion, the link is converted to the same link as that of the Poisson model, $\log(\mu)$. With respect to the linear predictor $x\beta$ the mean is defined as $\mu = \exp(x\beta)$.

Code 6.9 provides an example of a maximum likelihood synthetic negative binomial model with binary and continuous variables as predictors. The binary predictor, with 60 percent 1s and 40 percent 0s, is given a coefficient 2, the continuous predictor x_2 has coefficient -1.5 , and the intercept is 1. The direct dispersion parameter is 3.3, or $\approx 1/0.3$. Therefore θ is 0.3. The code can be used for either parameterization.

Code 6.9 Synthetic negative binomial data and model in R.

```
=====
library(MASS)
set.seed(141)
nobs <- 2500
x1 <- rbinom(nobs, size=1, prob=0.6)
x2 <- runif(nobs)
xb <- 1 + 2.0*x1 - 1.5*x2
a <- 3.3
theta <- 0.303                      # 1/a
exb <- exp(xb)
nby <- rnegbin(n=nobs, mu=exb, theta=theta)
negbml <- data.frame(nby, x1, x2)
nb2 <- glm.nb(nby ~ x1 + x2, data=negbml)
summary(nb2)
=====

Coefficients:
            Estimate Std. Error z value   Pr(>|z|)
(Intercept) 0.99273   0.08848  11.22 <2e-16 ***
x1          2.03312   0.08084  25.15 <2e-16 ***
x2         -1.61282   0.13550 -11.90 <2e-16 ***
(Dispersion parameter for Negative Binomial(0.2952) family taken to be 1)

Null deviance: 3028.2 on 2499 degrees of freedom
Residual deviance: 2356.1 on 2497 degrees of freedom
AIC: 11556
    Theta: 0.2952
    Std. Err.: 0.0107
2 x log-likelihood: -11548.4130

# Convert theta to alpha
> 1/0.2952
[1] 3.387534
```

The model may be estimated assuming a direct relationship between the mean and any overdispersion that may exist in the data. We use the `nbinomial` function from the COUNT package, which is available on CRAN. The model is given as

Code 6.10 Negative binomial model in R using COUNT.

```
=====
library(COUNT)
nb3 <- nbinomial(nby~ x1 + x2, data=negbml)
summary(nb3)

Deviance Residuals:
    Min.  1st Qu.  Median  Mean  3rd Qu.  Max.
-1.58600 -1.12400 -0.79030 -0.48950  0.06517  3.04800

Pearson Residuals:
    Min.  1st Qu.  Median  Mean  3rd Qu.  Max.
-0.539500 -0.510700 -0.445600  0.000435  0.067810 10.170000

Coefficients (all in linear predictor):
            Estimate      SE      Z      p      LCL      UCL
(Intercept) 0.993  0.0902  11.0  3.42e-28  0.816  1.17
x1          2.033  0.0808  25.2  8.4e-140  1.875  2.19
x2         -1.613  0.1372 -11.8  6.79e-32 -1.882 -1.34
(Intercept)_s 3.388  0.1233  27.5  4e-166  3.146  3.63

Null deviance: 3028.224 on 2498 d.f.
Residual deviance: 2356.052 on 2496 d.f.
Null Pearson: 4719.934 on 2498 d.f.
Residual Pearson: 2469.713 on 2496 d.f.
Dispersion: 0.9894685
AIC: 11556.41
=====
```

Note that the Pearson and deviance residuals are both provided in the results. The Pearson χ^2

and the negative binomial dispersion statistic are also provided. A negative binomial dispersion statistic value of 0.9894685 is close enough to 1 that it may be concluded that the model is well fitted as a negative binomial. We expect this result since the model is based on true synthetic negative binomial data with these specified parameters. The dispersion parameter is given as 3.38, which is the inverse of the θ dispersion parameter displayed in `g1m.nb` above. Notice also that the 95% confidence intervals come with the output by default. This is not the case with `g1m` or `g1m.nb`. Values of zero are not within the range of the confidence intervals, indicating that there is *prima facie* evidence that the predictors are significant.

We recommend that all frequentist-based count models be run using robust or sandwich standard errors. This is an adjustment to the standard errors that attempts to adjust standard errors to values that would be the case if there were no overdispersion in the data. This is similar to scaling the standard errors, which is done by multiplying the standard errors by the square root of the dispersion statistic. For both methods of adjustment, if the data is not over- (or under-) dispersed then the robust and scaled standard error reduces to a standard-model standard error. It is not something that Bayesian statisticians or analysts need to be concerned with, however. But it is wise to keep this in mind when modeling frequentist count models. It is a way to adjust for extra-dispersion in the data. Moreover, before waiting a long time while modeling data with a large number of observations and parameters, it is wise first to run a maximum likelihood model on the data if possible.

Recent literature in the area has recommended the negative binomial as a general-purpose model to adjust for Poisson overdispersion. However, the same problem with zeros exists for the negative binomial as for the Poisson. It may be that the data has too many zeros for either a Poisson or a negative binomial model. The formula for calculating the probability and expected number of zero counts for a negative binomial model is

$$P(Y = 0|\mu) = \binom{0 + 1/\alpha - 1}{1/\alpha - 1} \left(\frac{1}{1 + \alpha\mu}\right)^{1/\alpha} \left(\frac{\alpha\mu}{1 + \alpha\mu}\right)^0 = (1 + \alpha\mu)^{-1/\alpha}. \quad (6.6)$$

We can estimate the expected mean number of zeros for our data as follows:

```
alpha <- 1/nb2$theta
prnb <- mean((1 + alpha * exb)^(-1/alpha))
prnb
[1] 0.4547265

cnt <- 2500 * prnb
c(prnb, cnt)
[1] 0.4547265 1136.8163208

Nzeros <- sum(nby == 0)
Nzeros
[1] 1167           # number of 0s in data; compare with ideal 1137
Nzeros/2500
[1] 0.4668         # percentage of 0s; compare with ideal 0.4547

library(COUNT)
PDisp(nb2)
pearson.chi2 dispersion
2469.7136566   0.9890723  # NB dispersion statistic
```

Visualize frequency of counts

```
barplot(table(nby), log="y", col="gray")
```

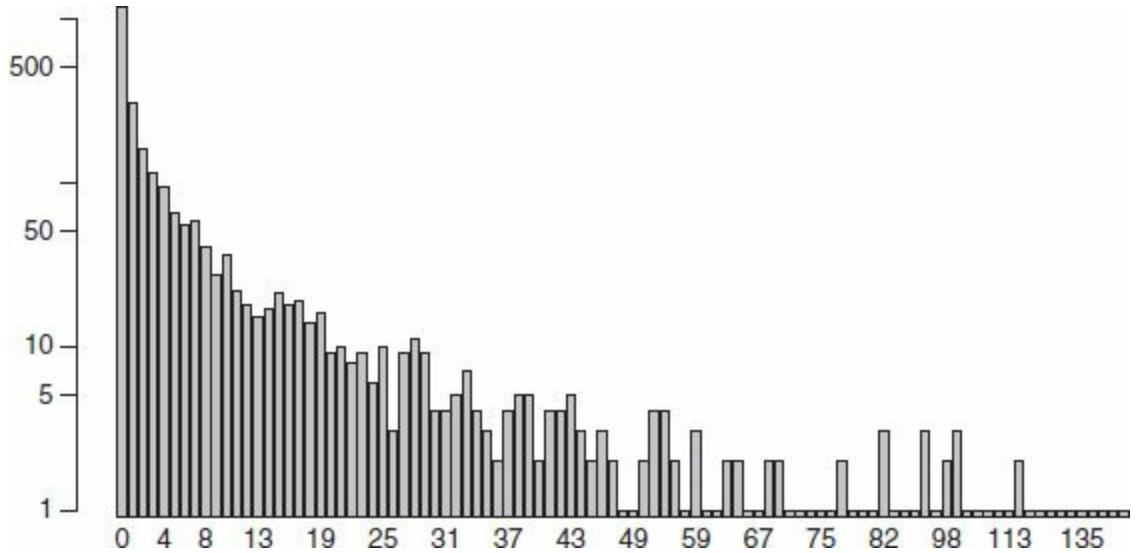


Figure 6.6 Data distribution for the negative binomial model.

In this case the predicted zero count for the negative binomial model is 1137 and the observed number of zeros is 1167, which again is pretty close as we should expect for synthetic data. Note that the number of zeros expected from a Poisson distribution with a similar mean would be

```
prpois <- mean(exp(-exb))
cntpois <- 2500 * prpois
cntpois
[1] 287.1995
```

This is much lower than is allowed by the negative binomial distribution. We should compare the range of observed versus predicted counts to inspect the fit, and check for places in the distributions where they vary. A fit test can be designed to quantify the fit. We shall discuss such a test later in the book.

6.2.1 Modeling the Negative Binomial using JAGS

The negative binomial is a very important count model. It is clear that using a Poisson model on count data is seldom warranted. Real data is most often overdispersed, is less often underdispersed, and is rarely equidispersed. So, the negative binomial is generally the ideal model to use with most count data. Moreover, when a direct parameterization of the dispersion parameter is used, a Poisson model may be thought of as a negative binomial having dispersion parameter zero or approximating zero. Moreover, recall that a negative binomial with a dispersion parameter equal to 1 is a geometric model. The negative binomial is, therefore, a good all-purpose tool for modeling overdispersed count data.

We will provide the reader with three methods of modeling a Bayesian negative binomial. Of course, any of these methods can be modified for a particular data problem facing the analyst. The code for an indirectly parameterized negative binomial is based on mixing the Poisson and gamma distributions. This is done using the code snippet

```
Y[i] ~ dpois(g[i])
g[i] ~ dgamma(alpha, rateParm[i])
```

The first model can be converted to a directly parameterized model by substituting the code provided in Code 6.12. Note also that the code can easily be used to model other data: simply amend these lines to specify different predictors, a different response variable, and model name.

Code 6.11 Bayesian negative binomial in R using JAGS.

```
=====
X <- model.matrix(~ x1 + x2, data=negbml)
NB.data <- list(
  Y = negbml$nby,
  N = nrow(negbml))

library(R2jags)
# Attach(negbml)
X <- model.matrix(~ x1_2 + x2)
K <- ncol(X)

model.data <- list(
  Y = negbml$nby,
  X = X,
  K = K,
  N = nrow(negbml))

sink("NBGLM.txt")
cat("
model{
  # Priors for coefficients
  for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001)}
  # Prior for dispersion
  theta ~ dunif(0.001, 5)

  # Likelihood function
  for (i in 1:N){
    Y[i] ~ dpois(g[i])
    g[i] ~ dgamma(theta, rateParm[i])
    rateParm[i] <- theta / mu[i]
    log(mu[i]) <- eta[i]
    eta[i] <- inprod(beta[], X[i,])
  }
}
", fill = TRUE)
sink()

inits <- function () {
  list(
    beta    = rnorm(K, 0, 0.1),      # regression parameters
    theta   = runif(0.00, 5)        # dispersion
  )
}

params <- c("beta", "theta")

NB2 <- jags(data      = model.data,
             inits     = inits,
             parameters = params,
             model     = "NBGLM.txt",
             n.thin    = 1,
             n.chains  = 3,
             n.burnin  = 3000,
             n.iter    = 5000)
print(NB2, intervals=c(0.025, 0.975), digits=3)
=====

          mu.vect    sd.vect      2.5%     97.5%   Rhat   n.eff
```

```

theta      0.295    0.011    0.274    0.316  1.002   2100
beta[1]    0.996    0.087    0.828    1.163  1.004   680
beta[2]    2.031    0.081    1.875    2.191  1.001   6000
beta[3]   -1.614    0.135   -1.868   -1.341  1.004   550
deviance  6610.096   62.684  6490.882  6730.957 1.001   6000
pD = 1965.2 and DIC = 8575.3

```

The results clearly approximate the parameter values we expected on the basis of the data set used for the model.

The above code produces an indirectly parameterized negative binomial. To change it to a direct parameterization insert Code 6.12 below in place of the log-likelihood code given in Code 6.11 above.

Code 6.12 Negative binomial with direct parameterization.

```
=====
alpha ~ dunif(0.001, 5)
# Likelihood function
for (i in 1:N){
  Y[i] ~ dpois(g[i])
  g[i] ~ dgamma(1/alpha, rateParm[i])
  rateParm[i] <- (1/alpha) / mu[i]
  log(mu[i]) <- eta[i]
  eta[i] <- inprod(beta[], X[i,])
}
=====
```

The lines in the `inits` and `params` components are as follows:

```
alpha = runif(0.00, 5) # dispersion
params <- c("beta", "alpha")
```

The resulting output is the same as that for the indirect dispersion, except that the line for alpha now reads

```
alpha      3.306    0.084    3.141    3.479  1.007   930
```

Code 6.13 below provides a directly parameterized negative binomial; but, instead of mixing the Poisson and gamma distributions, we use the `dnegbin` function. Internally this function does exactly the same thing as the two lines in Code 6.12 that mix distributions.

Code 6.13 Negative binomial: direct parameterization using JAGS and `dnegbin`.

```
=====
library(R2jags)

X <- model.matrix(~ x1 + x2, data=negbml)
K <- ncol(X)

model.data <- list(
  Y = negbml$nby,
  X = X,
  K = K,
  N = nrow(negbml))

# Model
sink("NBDGLM.txt")
cat("
model{
  # Priors for coefficients
  for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001)}
```

```

# Prior for alpha
alpha ~ dunif(0.001, 5)

# Likelihood function
for (i in 1:N){
  Y[i] ~ dnegbin(p[i], 1/alpha)    # for indirect, (p[i], alpha)
  p[i] <- 1/(1 + alpha*mu[i])
  log(mu[i]) <- eta[i]
  eta[i] <- inprod(beta[], X[i,])
}
",fill = TRUE)
sink()

inits <- function () {
  list(
    beta     = rnorm(K, 0, 0.1), # regression parameters
    alpha    = runif(0.00, 5)
  )
}

params <- c("beta", "alpha")

NB3 <- jags(data      = model.data,
             inits     = inits,
             parameters = params,
             model     = "NBDGLM.txt",
             n.thin    = 1,
             n.chains  = 3,
             n.burnin  = 3000,
             n.iter    = 5000)

print(NB3, intervals=c(0.025, 0.975), digits=3)
=====
          mu.vect  sd.vect    2.5%   97.5%   Rhat n.eff
alpha       3.404   0.123   3.174   3.650  1.001  6000
beta[1]     1.003   0.094   0.819   1.186  1.003  1000
beta[2]     2.033   0.083   1.871   2.198  1.002  1200
beta[3]    -1.628   0.141  -1.902  -1.345  1.002  1300
deviance  11552.547  2.847 11548.920 11559.807  1.002  2400

pD = 4.1 and DIC = 11556.6

```

To change the above direct parameterization we simply amend `1/alpha` to `alpha` in the line with the comment `# for indirect, (p[i], alpha)`. To display the chains for each parameter (Figure 6.7) one can type:

```

source("CH-Figures.R")
out <- NB3$BUGSoutput
MyBUGSChains(out,c(uNames("beta",K),"alpha"))

```

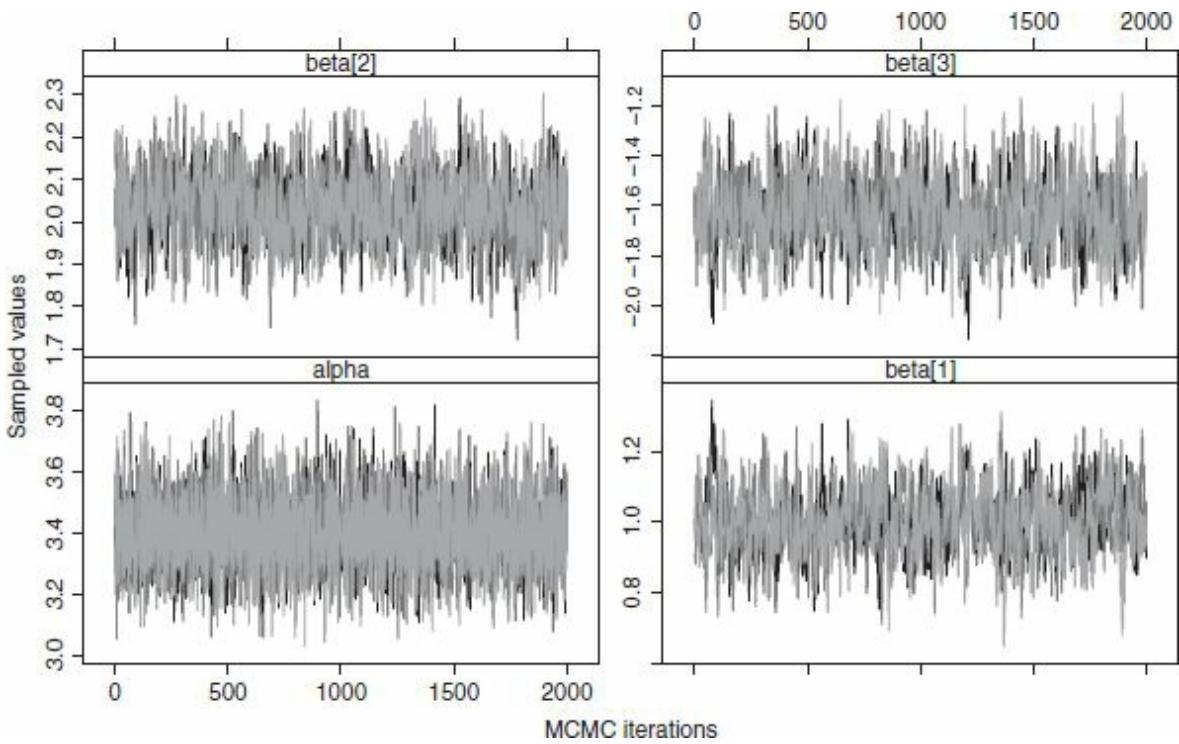


Figure 6.7 MCMC chains for the three model parameters $\beta_1, \beta_2, \beta_3$ and for α , for the negative binomial model.

Use the following code to display the histograms of the model parameters (Figure 6.8):

```
out <- NB3$BUGSoutput
MyBUGSHist(out,c(uNames("beta"),K), "alpha"))
```

Finally, we will use the zero trick with a negative binomial to model the same data. The zero-trick method is very useful if the analyst knows only the log-likelihood of the model.

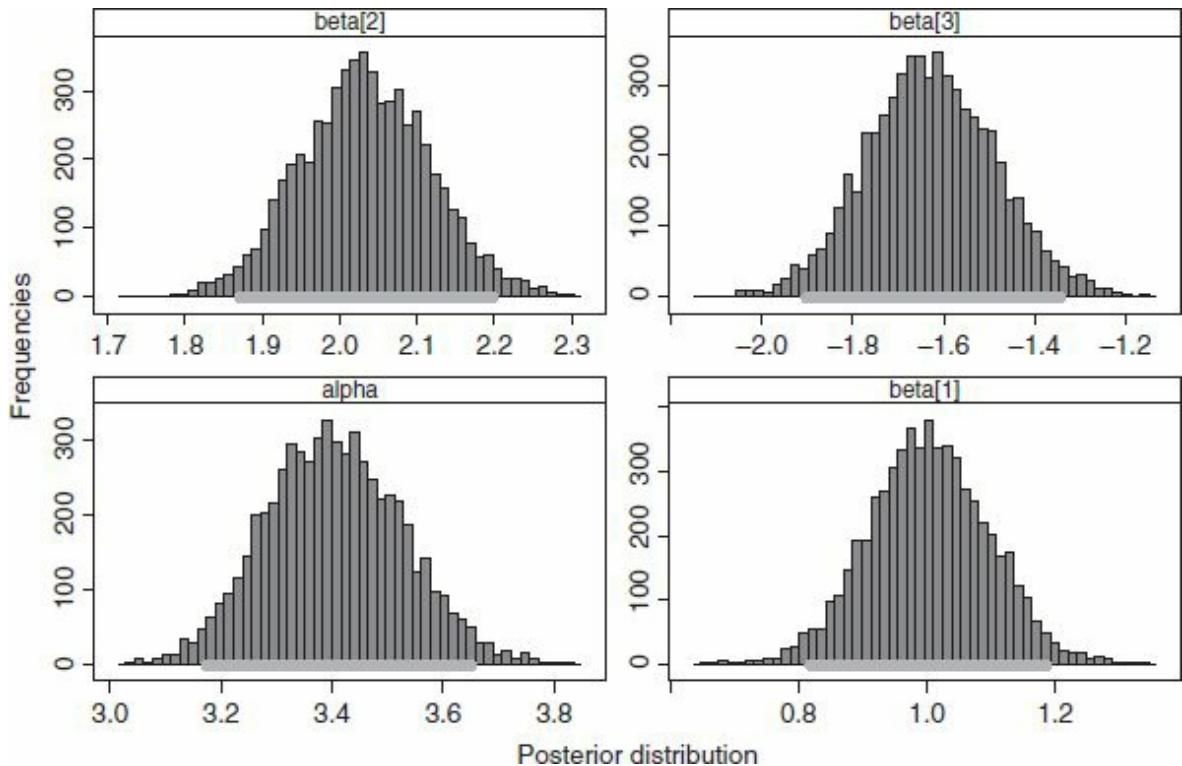


Figure 6.8 Histogram of the MCMC iterations for each parameter. The thick line at the base of each histogram represents the 95% credible interval. Note that no 95% credible interval contains 0.

The code provides a direct parameterization of the dispersion parameter. The downside of using the zero trick when modeling is that convergence usually takes longer to achieve.

Code 6.14 Negative binomial with zero trick using JAGS directly.

```
=====
library(R2jags)

X <- model.matrix(~ x1 + x2, data=negbml)
K <- ncol(X)
N <- nrow(negbml)

model.data <- list(
  Y      = negbml$nb,
  N      = N,
  K      = K,
  X      = X,
  Zeros  = rep(0, N)
)
sink("NB0.txt")
cat("model{
# Priors regression parameters
for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001)}
# Prior for alpha
numS ~ dnorm(0, 0.0016)
denomS ~ dnorm(0, 1)
alpha <- abs(numS / denomS)

C <- 10000
for (i in 1:N) {
  # Log-likelihood function using zero trick:
  Zeros[i] ~ dpois(Zeros.mean[i])
  Zeros.mean[i] <- -L[i] + C
  l1[i] <- 1/alpha * log(u[i])}
```

```

12[i] <- Y[i] * log(1 - u[i])
13[i] <- loggam(Y[i] + i/alpha)
14[i] <- loggam(1/alpha)
15[i] <- loggam(Y[i] + 1)
L[i] <- l1[i] + l2[i] + l3[i] - l4[i] - l5[i]

u[i]      <- (1/alpha) / (1/alpha + mu[i])
log(mu[i]) <- max(-20, min(20, eta[i]))
eta[i]    <- inprod(X[i,], beta[])
}

}

",fill = TRUE)
sink()

inits1 <- function () {
  list(beta  = rnorm(K, 0, 0.1),
       numS   = rnorm(1, 0, 25) ,
       denoms = rnorm(1, 0, 1)
     )
}

params1 <- c("beta", "alpha")

NB01 <- jags(data = model.data,
              inits = inits1,
              parameters = params1,
              model = "NB0.txt",
              n.thin = 1,
              n.chains = 3,
              n.burnin = 3000,
              n.iter = 5000)
print(NB01, intervals=c(0.025, 0.975), digits=3)
=====
mu.vect    sd.vect      2.5%      97.5%   Rhat  n.eff
alpha        3.399    0.124      3.164      3.644  1.001  5200
beta[1]      0.996    0.087      0.825      1.168  1.015  150
beta[2]      2.031    0.080      1.874      2.185  1.012  180
beta[3]     -1.611    0.133     -1.870     -1.342  1.004  830
deviance   50011552.305   2.747  50011548.892  50011559.245  1.000    1

pD = 3.8 and DIC = 50011556.1

```

To employ an indirect parameterization, substitute the following lines in Code 6.14:

Code 6.15 Change to indirect parameterization in R.

```

=====
l1[i] <- alpha * log(u[i])
l2[i] <- Y[i] * log(1 - u[i])
l3[i] <- loggam(Y[i] + alpha)
l4[i] <- loggam(alpha)
l5[i] <- loggam(Y[i] + 1)
L[i] <- l1[i] + l2[i] + l3[i] - l4[i] - l5[i]
u[i]      <- alpha / (alpha + mu[i])
log(mu[i]) <- max(-20, min(20, eta[i]))
=====
```

The parameter values that have been determined using the above three Bayesian negative binomial algorithms are all nearly the same. Seeds were not given to the runs to demonstrate the sameness of the results. Moreover, each algorithm can easily be amended to allow for either a direct or indirect parameterization of the dispersion. Under a direct parameterization $\alpha = 3.39 - 3.41$; under an indirect parameterization, $\alpha = 0.29 - 0.31$. Depending on the parameters and priors given to the sampling algorithm as well as the burn-in number, thinning rate, and so forth, the range of values for α can be considerably wider.

When modeling count data that has evidence of excessive correlation, or overdispersion, analysts should employ a negative binomial model to determine parameter values and the associated standard deviation and credible intervals. We recommend that a direct parameterization be given

between the mean and the value given to the dispersion parameter. It is easy to run a Bayesian negative binomial model with an indirect parameterization of the dispersion parameter, but doing so appears to be contrary to how statisticians normally think of variability. See [Hilbe \(2014\)](#) for an in-depth evaluation of this discussion and its consequences.

A negative binomial is appropriate for use with overdispersed Poisson data. In the following section we turn to a model that can be used for either underdispersed or overdispersed count data – the generalized Poisson model. First, however, we look at a Python implementation of the negative binomial.

6.2.2 Negative Binomial Models in Python using pymc3

Code 6.16 Negative binomial model in Python using pymc3.

```
=====
import numpy as np
import pandas
import pylab as plt
import pymc3 as pm
import pystan
from scipy.stats import uniform, binom, nbinom
import statsmodels.api as sm

# Data
np.random.seed(141)                      # set seed to replicate example
nobs = 2500                                 # number of obs in model

x1 = binom.rvs(1, 0.6, size=nobs)          # categorical explanatory variable
x2 = uniform.rvs(size=nobs)                # real explanatory variable

theta = 0.303
xb = 1 + 2 * x1 - 1.5 * x2               # linear predictor

exb = np.exp(xb)
nby = nbinom.rvs(exb, theta)

df = pandas.DataFrame({'x1': x1, 'x2': x2, 'nby': nby})    # re-write data

# Fit
niter = 10000                                # parameters for MCMC

with pm.Model() as model_glm:
    # Define priors
    beta0 = pm.Flat('beta0')
    beta1 = pm.Flat('beta1')
    beta2 = pm.Flat('beta2')

    # Define likelihood
    linp = beta0 + beta1 * x1 + beta2 * x2
    mu = np.exp(linp)
    mu2 = mu * (1 - theta)/theta    # compensate for difference between
                                    # parameterizations from pymc3 and scipy
    y_obs = pm.NegativeBinomial('y_obs', mu2, theta, observed=nby)

    # Inference
    start = pm.find_MAP()           # find starting value by optimization
    step = pm.NUTS()
    trace = pm.sample(niter, step, start, progressbar=True)

# Print summary to screen
pm.summary(trace)

# Show graphical output
pm.traceplot(trace)
plt.show()
=====

beta0:
  Mean           SD        MC Error      95% HPD interval
-----+-----+-----+-----+-----+
  1.020       0.089       0.002      [0.846, 1.193]
```

Posterior quantiles:

	2.5	25	50	75	97.5
-----	0.849	0.960	1.017	1.080	1.197

beta1:

Mean	SD	MC Error	95% HPD interval
1.989	0.078	0.001	[1.833, 2.138]

Posterior quantiles:

	2.5	25	50	75	97.5
-----	1.837	1.936	1.989	2.041	2.143

beta2:

Mean	SD	MC Error	95% HPD interval
-1.516	0.130	0.002	[-1.769, -1.256]

Posterior quantiles:

	2.5	25	50	75	97.5
-----	-1.777	-1.603	-1.512	-1.428	-1.261

6.2.3 Modeling the Negative Binomial in Python using Stan

Code 6.17 Negative binomial model in Python using Stan.¹

```
=====
import numpy as np
import pystan
from scipy.stats import uniform, binom, nbinom
import statsmodels.api as sm

# Data
np.random.seed(141)                      # set seed to replicate example
nobs = 2500                                # number of obs in model

x1 = binom.rvs(1, 0.6, size=nobs)          # categorical explanatory variable
x2 = uniform.rvs(size=nobs)                # real explanatory variable

theta = 0.303
X = sm.add_constant(np.column_stack((x1, x2)))
beta = [1.0, 2.0, -1.5]
xb = np.dot(X, beta)                      # linear predictor

exb = np.exp(xb)
nby = nbinom.rvs(exb, theta)

mydata = {}                                  # build data dictionary
mydata['N'] = nobs                         # sample size
mydata['X'] = X                            # predictors
mydata['Y'] = nby                          # response variable
mydata['K'] = len(beta)

# Fit
stan_code = """
data{
    int N;
    int K;
    matrix[N,K] X;
    int Y[N];
}
parameters{
    vector[K] beta;
    real<lower=0, upper=5> alpha;
}
transformed parameters{
    vector[N] mu;
    mu = exp(X * beta);
}
model{
    for (i in 1:K) beta[i] ~ normal(0, 100);
    Y ~ neg_binomial(mu, alpha);
}
"""
# Run mcmc
fit = pystan.stan(model_code=stan_code, data=mydata, iter=10000, chains=3,
                   warmup=5000, n_jobs=3)

# Output
nlines = 9                                    # number of lines in screen output

output = str(fit).split('\n')
for item in output[:nlines]:
    print(item)
=====

      mean   se_mean     sd   2.5%   25%   50%   75% 97.5%   n_eff   Rhat
beta[0]  1.05  8.8e-4  0.05  0.95  1.02  1.05  1.08 1.14  3181.0  1.0
beta[1]  1.97  5.0e-4  0.03  1.91  1.95  1.97  1.99 2.03  3692.0  1.0
beta[2] -1.52  5.3e-4  0.03 -1.58 -1.54 -1.52 -1.5 -1.45  3725.0  1.0
alpha    0.44  2.3e-4  0.02  0.41  0.43  0.44  0.46 0.48  7339.0  1.0
```

6.3 Bayesian Generalized Poisson Model

There are two foremost parameterizations of the generalized Poisson model. Both parameterizations allow for the modeling of both overdispersed as well as underdispersed count data. The first parameterization was developed by [Consul and Famoye \(1992\)](#) and is the method used in current software implementations of the model. The second parameterization was developed by [Famoye and Singh \(2006\)](#). The models are built upon the generalized Poisson distribution. The model was recently given a complete chapter in [Hilbe \(2014\)](#), to which we refer the interested reader. The approach is maximum likelihood but the logic of the model is the same regardless of its method of estimation. This is the only discussion of the Bayesian generalized Poisson of which we are aware, but it can be a powerful tool in modeling both under- and overdispersed count data.

The use of either parameterization results in the same parameter estimates. The generalized Poisson model has an extra dispersion parameter, similar to that of the negative binomial model. The difference is, however, that it can have both positive and negative values. If the data being modeled is overdispersed, the generalized Poisson model dispersion parameter will be positive. If the data is underdispersed, the dispersion parameter is negative. Negative binomial models can only model overdispersed data. If the data is equidispersed (Poisson) then the dispersion parameter has the value zero, or approximately zero. We note that frequently the generalized Poisson model fits overdispersed data better than does a negative binomial model. Unfortunately most researchers have not had access to the generalized Poisson for use in their modeling endeavors. There are only a very few software implementations of the model. Bayesian generalized Poisson model software has not appeared before.

The probability distribution and log-likelihood for the generalized Poisson can be expressed as

$$f(y_i; \mu, \delta) = \frac{\mu_i(\mu_i + \delta y_i)^{y_i-1} e^{-\mu_i - \delta y_i}}{y_i!} \quad y_i = 0, 1, 2, \dots \quad (6.7)$$

where $\mu > 0$ – $1 \leq \delta \leq 1$, and

$$\mathcal{L} = \log(\mu_i) + (y_i - 1) \log(\mu_i + \delta y_i) - \mu_i - \delta y_i - \log \Gamma(y_i + 1). \quad (6.8)$$

The mean and variance are given by

$$E(y) = \frac{\mu}{1 - \delta} \quad Var(y) = \frac{\mu}{(1 - \delta)^2}, \quad (6.9)$$

thus when $\delta = 0$, $E(y) = Var(y) = \mu$ and the distribution recovers the usual Poisson distribution (see e.g. the article [Hardin and Hilbe, 2012](#)). The log-likelihood will be used in the JAGS code to estimate Bayesian parameters. A generalized Poisson random number generator that we developed for this model is used to create synthetic generalized Poisson data with specified parameter estimates. We assign a coefficient value to the intercept of 1 and a coefficient value of 3.5 for the x1 predictor. The dispersion parameter, delta, is given the value – 0.3. As it turns out

the model is underdispersed, resulting in a delta parameter value -0.287 . Of course, since we are using a Bayesian generalized Poisson model, the intercept, x_1 and δ are all random parameters.

The generalized Poisson pseudo-random numbers are generated from the `rgp.R` file, which is on the book's website. We have located, or developed ourselves, a number of R pseudo-random number generators which can be used to create synthetic data based on a particular distribution, or mixture of distributions. They are all also available on the book's website, as well as in a file for this book on the first author's BePress site. Data sets are also included.

It should be noted that since the generalized Poisson dispersion statistic can have negative values, there are times when the predicted probabilities extend below 0. This is the case when the count response is large but delta is barely negative. Probabilities greater than 1 can also be produced given specific combinations of the count (y) and dispersion δ . Symbolizing the generalized Poisson PDF as GP and the predicted probabilities as p , the solution to constraining all probabilities to remain in the range 0, 1 is to use the formula

```
p <- min(1, max(0, GP)).
```

Code 6.18 Synthetic data for generalized Poisson.

```
=====
require(MASS)
require(R2jags)
source("c://rfiles/rgp.R")    # or wherever you store the file
set.seed(160)
nobs <- 1000

x1 <- runif(nobs)
xb <- 1 + 3.5 * x1
exb <- exp(xb)
delta <- -0.3

gpy <- c()
for(i in 1:nobs){
  gpy[i] <- rgp(1, mu=(1-delta) * exb[i], delta = delta)
}
gpdata <- data.frame(gpy, x1)
=====
```

6.3.1 Generalized Poisson Model using JAGS

The JAGS generalized Poisson code is provided below. We use a zero trick for the model. Code is also provided to calculate the generalized Poisson mean, variance, and residuals.

Code 6.19 Bayesian generalized Poisson using JAGS.

```
=====
X <- model.matrix(~ x1, data = gpdata)
K <- ncol(X)

model.data <- list(Y = gpdata$gpy,           # response
                     X = X,                 # covariates
                     N = nrow(gpdata),      # sample size
                     K = K,                 # number of betas
                     Zeros = rep(0, nrow(gpdata)))

sink("GP1reg.txt")
cat("model{
  # Priors beta
  for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001)}")
```

```

# Prior for delta parameter of GP distribution
delta ~ dunif(-1, 1)

C <- 10000
for (i in 1:N){
  Zeros[i] ~ dpois(Zeros.mean[i])
  Zeros.mean[i] <- -L[i] + C

  l1[i] <- log(mu[i])
  l2[i] <- (Y[i] - 1) * log(mu[i] + delta * Y[i])
  l3[i] <- -mu[i] - delta * Y[i]
  l4[i] <- -loggam(Y[i] + 1)
  L[i] <- l1[i] + l2[i] + l3[i] + l4[i]
  mu[i] <- (1 - delta)*exp(eta[i])
  eta[i] <- inprod(beta[], X[i,])
}

# Discrepancy measures: mean, variance, Pearson residuals
for (i in 1:N){
  ExpY[i] <- mu[i] / (1 - delta)
  VaryY[i] <- mu[i] / ((1 - delta)^3)
  Pres[i] <- (Y[i] - ExpY[i]) / sqrt(VaryY[i])
}
",fill = TRUE)
sink()

inits <- function () {
  list(
    beta = rnorm(ncol(X), 0, 0.1),
    delta = 0)}

params <- c("beta", "delta")

GP1 <- jags(data = model.data,
             inits = inits,
             parameters = params,
             model = "GP1reg.txt",
             n.thin = 1,
             n.chains = 3,
             n.burnin = 4000,
             n.iter = 5000)
print(GP1, intervals=c(0.025, 0.975), digits=3)
=====
      mu.vect   sd.vect       2.5%      97.5%     Rhat  n.eff
beta[1]      1.029    0.017      0.993     1.062  1.017    130
beta[2]      3.475    0.022      3.432     3.521  1.016    140
delta        -0.287    0.028     -0.344    -0.233  1.002   1900
deviance    20005178.615   2.442  20005175.824  20005184.992  1.000      1

pD = 3.0 and DIC = 20005181.6

```

The results are very close to the values specified in setting up the synthetic data. Again, this data could not have been modeled using a negative binomial since it is underdispersed. We know this is due to the negative value for `delta`. Of course, we set up the data to be underdispersed, but in real data situations it is unlikely that we would know this in advance unless it has been tested using a Poisson model. If the Pearson dispersion statistic from a Poisson model is less than 1.0, the model is very likely to be underdispersed. We always advise readers to first model the count data using a Poisson model and checking its dispersion statistic (not its parameter). Keep in mind, though, that generalized Poisson data may also be apparently overdispersed or underdispersed. We may be able to ameliorate apparent extra-dispersion by performing the operations on the data that we detailed when discussing the apparently extra-dispersed negative binomial model in the previous section.

We need to give a caveat regarding the parameterization of the dispersion parameter, `delta`. When the data is underdispersed, take care to be sure that the parameterization, given the dispersion statistic in the model code, is correct. The generalized Poisson dispersion needs to be parameterized with a hyperbolic tangent function or equivalent in order to produce negative values of `delta` when the data is underdispersed. Some R software with which we are familiar parameterizes the

dispersion using an exponential function instead, which does not allow negative values for delta in the case of underdispersed data. However, the maximum likelihood `vglm` function in the VGAM package provides both positive- and negative-valued dispersion parameters.

6.3.2 Generalized Poisson Model using Stan

Code 6.20 Generalized Poisson model in Python using Stan.

```
=====
import numpy as np
import pystan
import statsmodels.api as sm

from scipy.misc import factorial
from scipy.stats import uniform, rv_discrete

def sign(delta):
    """Returns a pair of vectors to set sign on
    generalized Poisson distribution. """
    if delta > 0:
        value = delta
        sig = 1.5
    else:
        value = abs(delta)
        sig = 0.5

    return value, sig

class gpoisson(rv_discrete):
    """Generalized Poisson distribution."""
    def _pmf(self, n, mu, delta, sig):
        if sig < 1.0:
            delta1 = -delta
        else:
            delta1 = delta
        term1 = mu * ((mu + delta1 * n) ** (n - 1))
        term2 = np.exp(-mu - n * delta1) / factorial(n)
        return term1 * term2

# Data
np.random.seed(160)                         # set seed to replicate example
nobs = 1000                                    # number of obs in model

x1 = uniform.rvs(size=nobs)
xb = 1.0 + 3.5 * x1                           # linear predictor
delta = -0.3
exb = np.exp(xb)

gen_poisson = gpoisson(name="gen_poisson", shapes='mu, delta, sig')
gpy = [gen_poisson.rvs(exb[i],
                      sign(delta)[0], sign(delta)[1]) for i in range(nobs)]

mydata = {}                                     # build data dictionary
mydata['N'] = nobs                             # sample size
mydata['X'] = sm.add_constant(np.transpose(x1)) # predictors
mydata['Y'] = gpy                               # response variable
mydata['K'] = 2                                  # number of coefficients

# Fit
stan_code = """
data{
    int N;
    int K;
    matrix[N, K] X;
    int Y[N];
}
parameters{
    vector[K] beta;
    real<lower=-1, upper=1> delta;
}
transformed parameters{
    vector[N] mu;
    mu = exp(X * beta);
```

```

}
model{
    vector[N] l1;
    vector[N] l2;
    vector[N] LL;

    delta ~ uniform(-1, 1);

    for (i in 1:N){
        l1[i] = log(mu[i]) + (Y[i] - 1) * log(mu[i] + delta * Y[i]);
        l2[i] = mu[i] + delta * Y[i] + lgamma(Y[i] + 1);
        LL[i] = l1[i] - l2[i];
    }
    target += LL;
}
"""

# Run mcmc
fit = pyStan.stan(model_code=stan_code, data=mydata, iter=5000, chains=3,
                   warmup=4000, n_jobs=3)

# Output
nlines = 8                                # number of lines in screen output

output = str(fit).split('\n')
for item in output[:nlines]:
    print(item)
=====
      mean se_mean    sd  2.5%   25%   50%   75% 97.5% n_eff Rhat
beta[0]    1.0  1.4e-3 0.03  0.93  0.98   1.0  1.02  1.06 520.0  1.0
beta[1]    3.51 1.1e-3 0.03  3.46  3.5   3.51  3.53  3.56 564.0  1.0
delta     -0.31 1.3e-3 0.03 -0.37 -0.33 -0.31 -0.29 -0.25 549.0  1.0

```

6.4 Bayesian Zero-Truncated Models

Zero-truncated models typically refer to count models for which the count response variable cannot include zero counts. For instance, in health care, the length-of-hospital-stay data begin from day one upwards. There are no zero lengths of stay when entering the hospital. Astronomical research also has many instances of this type of data. It is therefore important to recognize when Bayesian zero-truncated models should be used in place of standard Bayesian models. In particular, we shall discuss Bayesian zero-truncated Poisson (ZTP) and zero-truncated negative binomial (ZTNB) models. These are by far the most popular truncated model types in the statistical literature.

6.4.1 Bayesian Zero-Truncated Poisson Model

The zero-truncated Poisson (ZTP) model is a component of a Poisson hurdle model. A count hurdle model is a two-part model consisting of a zero-truncated count component and a binary, usually logit, component. The zero counts are typically modeled with a binary logistic model, and the positive-count component with a zero-truncated Poisson or negative binomial model. The equations relating to how to construct a zero-truncated model as part of a hurdle model are examined later in the present section. Here we shall look a bit more closely at the zero-truncated model as a model in its own right.

In the Poisson probability distribution, upon which Poisson models are based, it is assumed that there is a given zero-count number based on the mean value of the response variable being modeled. When it is not possible for zero counts to exist in the variable being modeled, the underlying PDF must be amended to exclude zeros and then the zero-less Poisson PDF must be adjusted in such a way that the probabilities sum to 1. The log-likelihood of this zero-truncated

Poisson probability function is given as

$$\mathcal{L}(\beta; y) = \sum_{i=1}^n \{y_i \ln(x_i \beta) - \exp(x_i \beta) - \ln \Gamma(y_i + 1) - \ln[1 - \exp(-\exp(x_i \beta))]\}. \quad (6.10)$$

Not all zero-truncated data need to have the underlying PDF adjusted as we discussed above. If the mean of the count response variable being modeled is greater than 5 or 6, it makes little difference to the Poisson PDF if there can be no zeros in the variable. But if the mean is less than 6, for example, it can make a difference.

Let us take as an example some count data which we desire to model as Poisson. Suppose that the mean of the count response is 3. Now, given that the probability of a Poisson zero count is $\exp(-\text{mean})$ we can calculate, for a specific mean, how much of the underlying Poisson probability and log-likelihood is lost by ignoring the fact that there are no zeros in the model. We may use R to determine the percentage of zeros that we should expect for a given mean on the basis of the Poisson PDF:

```
> exp(-3)
[1] 0.04978707
```

If there are 500 observations in the model then we expect the response to have 25 zero counts:

```
> exp(-3) * 500
[1] 24.89353
```

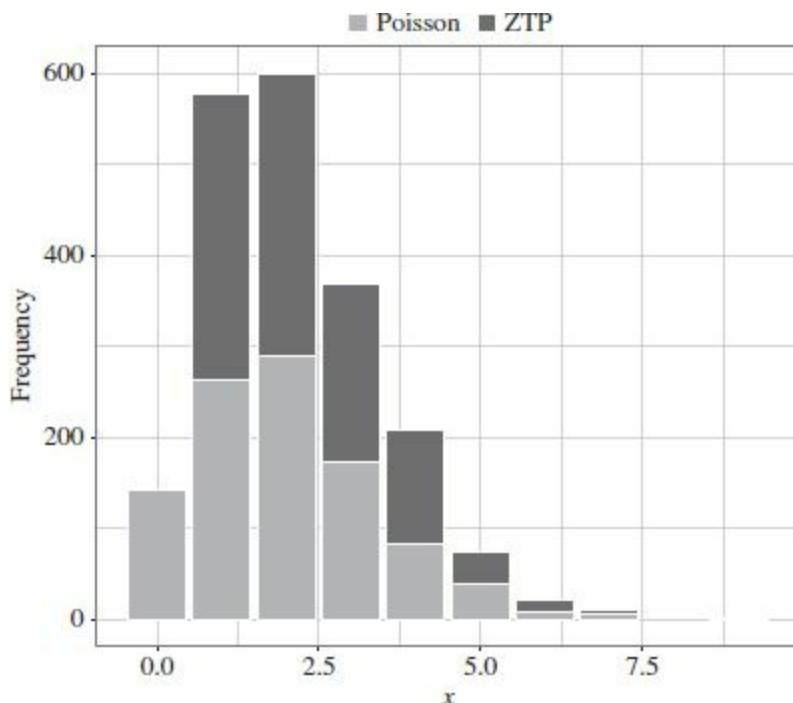


Figure 6.9 For comparison, Poisson and zero-truncated Poisson-distributed data.

Using a Poisson model on such data biases the results in proportion to the percentage of zero counts expected to be modeled. The Poisson dispersion statistic will most likely differ from 1 by a substantial amount, indicating either under- or overdispersion. Using a ZTP model on the data should ameliorate any extra-dispersion, particularly if there is no other cause for it in the data.

It does not take a much greater mean value than 3 to reduce the probability of zero counts to near zero. For instance, consider a count response variable with mean 6:

```
> exp(-6)
[1] 0.002478752
> exp(-6) * 500
[1] 1.239376
```

With mean 6, we should expect that the response data has about a one-quarter percent probability of having zero counts. For a 500-observation model we would expect only one zero in the response variable being modeled. Employing a Poisson model on the data will only result in a slight difference from a ZTP model on the data. Thus, when mean values exceed 6, it is generally not necessary to use a zero-truncated Poisson model.

However, and this is an important caveat, it may well be the case that a Poisson model on the data is inappropriate at the outset. If zero-truncated data remains overdispersed in spite of being adjusted by using a zero-truncated Poisson, it may be necessary to use a zero-truncated negative binomial. The formula we used for the probability of a zero count given a specific mean is only valid for a model based on the Poisson distribution. The probability of a zero negative binomial count is quite different.

The code below provides the user with a Bayesian zero-truncated Poisson model. Note that the zero-truncated Poisson log-likelihood uses the full Poisson log-likelihood with the addition of a term excluding zero counts. It is easiest to use the zero trick for such a model.

We first set up the zero-truncated Poisson data. We will use the same data as for the Poisson model earlier in the chapter but we add 1 to each count in *py*, the response count variable. Adding 1 to the response variable of synthetic data such as we have created is likely to result in an underdispersed Poisson model. Use of the zero-truncated model attempts to adjust the otherwise Poisson model for the structurally absent zero counts.

Code 6.21 Zero-truncated Poisson data.

```
=====
set.seed(123579)
nobs <- 1500
x1 <- rbinom(nobs, size=1, 0.3)
x2 <- rbinom(nobs, size=1, 0.6)
x3 <- runif(nobs)
xb <- 1 + 2*x1 - 3*x2 - 1.5*x3
exb <- exp(xb)

rztp <- function(N, lambda){
  p <- runif(N, dpois(0, lambda), 1)
  ztp <- qpois(p, lambda)
  return(ztp)
}
ztpy <- rztp(nobs, exb)
```

```
ztp <- data.frame(ztpy, x1, x2, x3)
=====
```

We display the counts to make certain that there are no zero counts in py . We also show that the mean of ztp is 2.43, which, as we learned before, does affect the modeling results.

```
> table(ztpy)
ztpy
  1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20  21  22  23  27
1014 207  80  31  22  12   9  24  12  15   9   8   8   8   5   8   6   8   5   1   1   3   2   1
  30
  1
> mean(ztpy)
[1] 2.43
```

The code listed in the table below first models the data using a standard maximum likelihood Poisson model. We can determine how closely the coefficients specified in our synthetic data are identified by the Poisson model. Remember, though, that we added 1 to py after assigning the coefficient values, so we expect the Poisson model coefficients to differ from what was originally specified in setting up the model. Moreover, no adjustment is being made by the Poisson model for the absent zero counts. The fact that there were originally 123 zeros in the data, i.e., 25 percent of the total, indicates that there will be a considerable difference in the Poisson coefficient values and in the Bayesian ZTP parameter means.

Code 6.22 Zero-truncated Poisson with zero trick.

```
=====
library(MASS)
library(R2jags)

X <- model.matrix(~ x1 + x2 + x3, data = ztp)
K <- ncol(X)

model.data <- list(Y = ztp$ztpy,
                     X = X,
                     K = K,           # number of betas
                     N = nobs,        # sample size
                     Zeros = rep(0, nobs))

ZTP<-
model{
  for (i in 1:K) {beta[i] ~ dnorm(0, 1e-4)}

  # Likelihood with zero trick
  C <- 1000
  for (i in 1:N) {
    Zeros[i] ~ dpois(-Li[i] + C)
    Li[i] <- Y[i] * log(mu[i]) - mu[i] -
      loggam(Y[i]+1) - log(1-exp(-mu[i]))
    log(mu[i]) <- inprod(beta[], X[i,])
  }
}

inits <- function () {
  list(beta = rnorm(K, 0, 0.1) )
}

params <- c("beta")

ZTP1 <- jags(data = model.data,
              inits = inits,
              parameters = params,
              model = textConnection(ZTP),
              n.thin = 1,
              n.chains = 3,
              n.burnin = 4000,
```

```

n.iter = 5000)
print(ZTP1, intervals=c(0.025, 0.975), digits=3)
=====
      mu.vect sd.vect    2.5%   97.5%   Rhat n.eff
beta[1]     0.985  0.060    0.859    1.096  1.027   100
beta[2]     1.996  0.054    1.895    2.107  1.017   130
beta[3]    -3.100  0.104   -3.308   -2.904  1.002  1400
beta[4]    -1.413  0.081   -1.566   -1.248  1.009   290
deviance  3002456.898  2.957 3002453.247 3002464.461  1.000       1

```

pD = 4.4 and DIC = 3002461.3

The maximum likelihood Poisson model provided the results given below. Again, no adjustment was made for the structural absence of zero counts in the data. Since μ has mean 2.43 we expected there to be a difference in results. It is clear that the Bayesian model provides closer estimates to the coefficients (posterior means) than were specified in the synthetic data. We used the `P_disp` function in the CRAN library LOGIT package to assess the Poisson dispersion. The model is, as expected, underdispersed:

```

poi <- glm(ztpy ~ x1 + x2 + x3, family=poisson, data=ztp)
summary(poi)

```

```

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) 1.35848   0.03921 34.64  <2e-16 ***
x1          1.27367   0.03390 37.57  <2e-16 ***
x2         -1.41602   0.03724 -38.02  <2e-16 ***
x3         -0.92188   0.06031 -15.29  <2e-16 ***

```

```

Null deviance: 4255.65 on 1499 degrees of freedom
Residual deviance: 918.52 on 1496 degrees of freedom
AIC: 4535.2

```

```

> library(LOGIT)
> P_disp(poi)

pearson.chi2    dispersion
943.0343784    0.6303706

```

6.4.2 Zero-Truncated Poisson in Python using Stan

In Stan, representing a truncated model requires the addition of a flag `tau[1,]` after the likelihood definition (see Code 6.23). The notation is general, so it can be applied to any truncation value `tau[low, high]`. It is important to notice that, at present, Stan does not allow the vectorization of truncated likelihoods or the truncation of multivariate distributions.

Code 6.23 Zero-truncated Poisson model in Python using Stan.

```
=====
import numpy as np
import pystan
import statsmodels.api as sm

from scipy.stats import uniform, binom, poisson

def ztpoisson(N, lambda_par):
    """Zero truncated Poisson distribution."""

    temp = poisson.pmf(0, lambda_par)
    p = [uniform.rvs(loc=item, scale=1-item) for item in temp]
    ztp = [int(poisson.ppf(p[i],lambda_par[i])) for i in range(N)]

    return np.array(ztp)

# Data
np.random.seed(123579)                         # set seed to replicate example
nobs = 3000                                     # number of obs in model

x1 = binom.rvs(1, 0.3, size=nobs)
x2 = binom.rvs(1, 0.6, size=nobs)
x3 = uniform.rvs(size=nobs)

xb = 1.0 + 2.0 * x1 - 3.0 * x2 - 1.5 * x3 # linear predictor
exb = np.exp(xb)

ztpy = ztpoisson(nobs, exb)                      # create y as adjusted

X = np.column_stack((x1,x2,x3))
X = sm.add_constant(X)

mydata = {}                                      # build data dictionary
mydata['N'] = nobs                             # sample size
mydata['X'] = X                                # predictors
mydata['Y'] = ztpy                             # response variable
mydata['K'] = X.shape[1]                        # number of coefficients

# Fit
stan_code = """
data{
    int N;
    int K;
    matrix[N, K] X;
    int Y[N];
}
parameters{
    vector[K] beta;
}
model{
    vector[N] mu;

    mu = exp(X * beta);

    # likelihood
    for (i in 1:N) Y[i] ~ poisson(mu[i]) T[1,];
}
"""
# Run mcmc
fit = pystan.stan(model_code=stan_code, data=mydata, iter=5000, chains=3,
                   warmup=4000, n_jobs=3)

# Output
print(fit)
=====
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
beta[0]	1.01	1.7e-3	0.04	0.94	0.99	1.01	1.04	1.1	566.0	1.0
beta[1]	1.98	1.6e-3	0.04	1.9	1.95	1.98	2.0	2.05	593.0	1.0
beta[2]	-3.05	2.6e-3	0.07	-3.19	-3.1	-3.04	-3.0	-2.9	792.0	1.0
beta[3]	-1.51	2.1e-3	0.05	-1.62	-1.55	-1.51	-1.48	-1.41	659.0	1.0

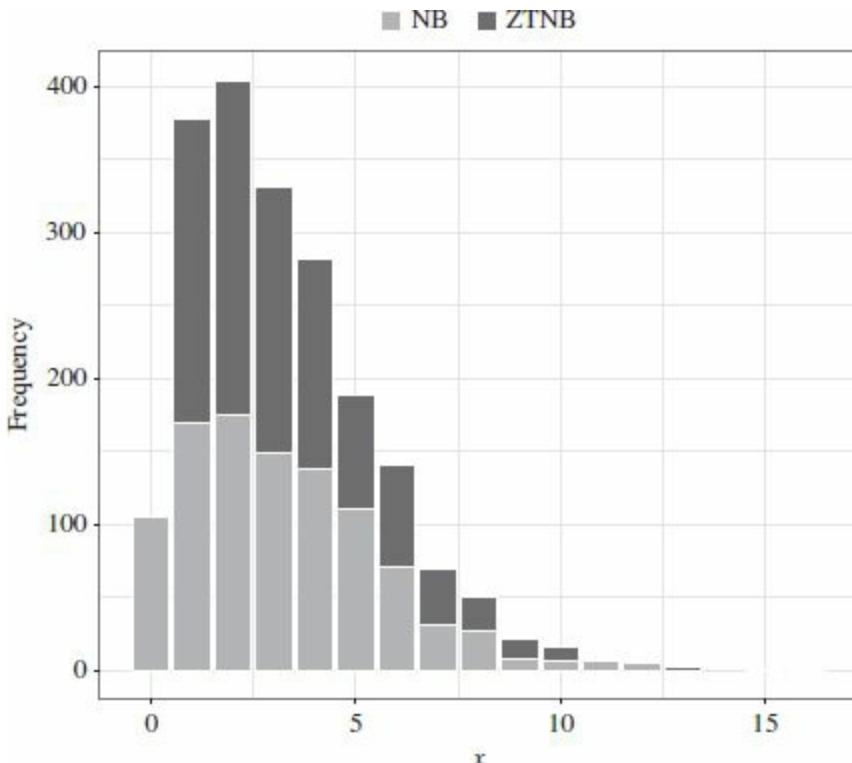


Figure 6.10 For comparison, negative binomial and zero-truncated negative binomial data.

6.4.3 Bayesian Zero-Truncated Negative Binomial Model

The logic of the zero-truncated negative binomial is the same as that of the zero-truncated Poisson, except that the negative binomial has a second parameter, the dispersion parameter. To calculate the probability of a zero count we must have values for both the mean and dispersion. Recall that the Poisson probability of a zero count given a mean of 3 is 0.05. For a 1000-observation model we would expect 50 zeros. For a negative binomial distribution, we use the formula

$$(1 + \alpha * \mu)^{-1/\alpha} \quad (6.11)$$

instead of $\exp(-\mu)$ for calculating the probability of zero counts. For a mean of 3, we display the expected number of zeros given α values of 0.5, 1, and 2:

$$\begin{aligned} (1+.5*3)^{(-1/.5)} * 1000 &= 160 \\ (1+1*3)^{(-1/1)} * 1000 &= 250 \\ (1+2*3)^{(-1/2)} * 1000 &= 378 \end{aligned}$$

The number of expected zero counts for a given mean is much higher for a negative binomial model than for a Poisson. We advise using the negative binomial as the default zero-truncated model since a wider range of count values is allowed, which make sense for a zero-truncated model. The DIC should be used as a general guide for deciding which model to use in a given modeling situation, all other considerations being equal.

The code in the following table is intended for modeling zero-truncated negative binomial data, for which direct parameterization of the dispersion parameter is used.

Code 6.24 Zero-truncated negative binomial with 0-trick using JAGS – direct.

```
=====
require(MASS)
require(R2jags)
require(VGAM)

set.seed(123579)
nobs <- 1000
x1 <- rbinom(nobs, size=1, 0.7)
x2 <- runif(nobs)
xb <- 1 + 2 * x1 - 4 * x2
exb <- exp(xb)
alpha = 5

rztnb <- function(n, mu, size){
  p <- runif(n, dzinegbin(0, munb=mu, size=size),1)
  ztnb <- qzinegbin(p, munb=mu, size=size)
  return(ztnb)
}

ztnby <- rztnb(nobs, exb, size=1/alpha)
ztnb.data <- data.frame(ztnby, x1, x2)
X <- model.matrix(~ x1 + x2, data = ztnb.data)
K <- ncol(X)
model.data <- list(Y = ztnb.data$ztnby,
                     X = X,
                     K = K, # number of betas
                     N = nobs,
                     Zeros = rep(0, nobs)) # sample size

ZTNB <- "
model{
  for (i in 1:K) {beta[i] ~ dnorm(0, 1e-4)}
  alpha ~ dgamma(1e-3,1e-3)

  # Likelihood with zero trick
  C <- 10000
  for (i in 1:N) {
    # Log likelihood function using zero trick:
    Zeros[i] ~ dpois(Zeros.mean[i])
    Zeros.mean[i] <- -L[i] + C
    l1[i] <- 1/alpha * log(u[i])
    l2[i] <- Y[i] * log(1 - u[i])
    l3[i] <- loggam(Y[i] + 1/alpha)
    l4[i] <- loggam(1/alpha)
    l5[i] <- loggam(Y[i] + 1)
    l6[i] <- log(1 - (1 + alpha * mu[i])^(-1/alpha))
    L[i] <- l1[i] + l2[i] + l3[i] - l4[i] - l5[i] - l6[i]
    u[i] <- 1/(1 + alpha * mu[i])
    log(mu[i]) <- inprod(X[i,], beta[])
  }
}

inits <- function () {
  list(beta = rnorm(K, 0, 0.1))}
params <- c("beta", "alpha")

ZTNB1 <- jags(data = model.data,
                inits = inits,
                parameters = params,
                model = textConnection(ZTNB),
                n.thin = 1,
                n.chains = 3,
                n.burnin = 2500,
                n.iter = 5000)
print(ZTNB1, intervals=c(0.025, 0.975), digits=3)
=====
```

	mu.vect	sd.vect	2.5%	97.5%	Rhat	n.eff
alpha	4.619	1.198	2.936	7.580	1.032	81
beta[1]	1.099	0.199	0.664	1.445	1.016	150
beta[2]	1.823	0.131	1.564	2.074	1.004	600

```

beta[3]      -3.957   0.199      -4.345      -3.571  1.008   290
deviance 20004596.833   2.861 20004593.165 20004603.978 1.000      1
pD = 4.1 and DIC = 20004600.9

```

A similar result can be achieved using Stan from Python. Code 6.25 demonstrates how this can be implemented. As always, it is necessary to pay attention to the different parameterizations used in Python and Stan. In this particular case, the `scipy.stats.binom` function takes as input (n, p) ¹ and the closest Stan negative binomial parameterization takes (α, β) , where $\alpha = n$ and $\beta = p/(1 - p)$.

Code 6.25 Zero-truncated negative binomial model in Python using Stan.

```

=====
import numpy as np
import pystan
import statsmodels.api as sm

from scipy.stats import uniform, nbinom, binom

def gen_ztnegbinom(n, mu, size):
    """Zero truncated negative binomial distribution"""

    temp = nbinom.pmf(0, mu, size)
    p = [uniform.rvs(loc=temp[i], scale=1-temp[i]) for i in range(n)]
    ztnb = [int(nbinom.ppf(p[i], mu[i], size)) for i in range(n)]

    return np.array(ztnb)

# Data
np.random.seed(123579)                      # set seed to replicate example
nobs = 2000                                     # number of obs in model

x1 = binom.rvs(1, 0.7, size=nobs)
x2 = uniform.rvs(size=nobs)

xb = 1.0 + 2.0 * x1 - 4.0 * x2             # linear predictor
exb = np.exp(xb)
alpha = 5
# Create y as adjusted
ztnby = gen_ztnegbinom(nobs, exb, 1.0/alpha)

X = np.column_stack((x1,x2))
X = sm.add_constant(X)

mydata = {}                                     # build data dictionary
mydata['N'] = nobs                            # sample size
mydata['X'] = X                               # predictors
mydata['Y'] = ztnby                           # response variable
mydata['K'] = X.shape[1]                       # number of coefficients

# Fit
stan_code = """
data{
    int N;
    int K;
    matrix[N, K] X;
    int Y[N];
}
parameters{
    vector[K] beta;
    real<lower=1> alpha;
}
model{
    vector[N] mu;

    # Covariates transformation
    mu = exp(X * beta);

    # likelihood
    for (i in 1:N) Y[i] ~ neg_binomial(mu[i], 1.0/(alpha - 1.0)) T[1,];
}
```

```

"""
# Run mcmc
fit = pystan.stan(model_code=stan_code, data=mydata, iter=5000, chains=3,
                    warmup=2500, n_jobs=3)

# Output
nlines = 9                                # number of lines in screen output

output = str(fit).split('\n')
for item in output[:nlines]
    print(item)
=====
      mean se_mean   sd  2.5%   25%   50%   75% 97.5%  n_eff Rhat
beta[0]  0.96  2.3e-3  0.08  0.79  0.91  0.96  1.02  1.12 1331.0  1.0
beta[1]  2.07  1.8e-3  0.07  1.94  2.02  2.07  2.11  2.2 1439.0  1.0
beta[2] -3.96  1.6e-3  0.07 -4.09 -4.01 -3.96 -3.92 -3.83 1879.0  1.0
alpha    4.84  4.7e-3  0.19  4.49  4.71  4.84  4.97  5.22 1577.0  1.0

```

6.5 Bayesian Three-Parameter NB Model (NB-P)

The three-parameter negative binomial P, or NB-P, model allows a much wider range of fitting capability than does a two-parameter count model. Designed by [William Greene \(2008\)](#) for econometric data, the model can be of very good use in astronomy. As a two-parameter negative binomial model can adjust for one-parameter Poisson overdispersion, the three-parameter NB-P can adjust for two-parameter negative binomial overdispersion.

The NB-P log-likelihood is defined as

$$\mathcal{L}(\mu; y, \alpha, \rho) = \sum_{i=1}^n \left\{ \alpha \mu^{2-P} \ln \left(\frac{\alpha \mu^{2-P}}{\alpha \mu^{2-P} + \mu} \right) + y \ln \left(\frac{\mu}{\alpha \mu^{2-P} + \mu} \right) - \ln \Gamma(1 - y) - \ln \Gamma(\alpha \mu^{2-P}) + \ln \Gamma(y + \alpha \mu^{2-P}) \right\}. \quad (6.12)$$

The evaluator now has four arguments: the log-likelihood, the linear predictor, the log of dispersion parameter, and P , which appears in the factor

$$\alpha \mu^{2-P}.$$

For the NB-P model we provide a simple data set based on the generic `rnegbin` random number generator. The code does not provide a specified value for α or P . The algorithm solves for Q , which is defined as $2 - P$. It is easier to estimate Q and then convert it to P than to estimate P . The model provides support for deciding whether an analyst should model data using an NB1 model or an NB2 model. NB1 has a variance of $\mu + \alpha \mu$ and NB2, the traditional negative binomial model, has a variance of $\mu + \alpha \mu^2$. The NB-P model parameterizes the exponent, understanding that the NB1 model can be expressed as $\mu + \alpha \mu$. If the NB-P model results in a Q value close to 2, the analyst will model the data using an NB2 model. Statisticians now do not concern themselves with choosing between NB1 and NB2 but prefer to utilize the power of using all three parameters to fit

the data better. The extra parameter can also be used to adjust for negative binomial over- or underdispersion. For a full explanation of the model see [Hilbe and Greene \(2008\)](#) and [Hilbe \(2011\)](#).

6.5.1 Three-Parameter NB-P Model using JAGS

For an example, we provide simple NB-P synthetic data. Note that the θ parameter that is specified as 0.5 is identical to an α of $1/0.5 = 2$.

Code 6.26 Create synthetic negative binomial data.

```
=====
require(R2jags)
require(MASS)
nobs <- 1500
x1 <- runif(nobs)
xb <- 2 - 5 * x1
exb <- exp(xb)
theta <- 0.5
Q = 1.4
nbpy <- rnegbin(n=nobs, mu = exb, theta = theta*exb^Q)
TP <- data.frame(nbpy, x1)
=====
```

...in R using JAGS

Code 6.27 Bayesian three-parameter NB-P – indirect parameterization with zero trick.

```
=====
X <- model.matrix(~ x1 , data = TP) # number of betas
K <- ncol(X)

model.data <- list(Y      = TP$nbpy,      # response
                    X      = X,          # covariates
                    N      = nobs,        # sample size
                    K      = K)
sink("NBPrep.txt")
cat("model{
# Diffuse normal priors on betas
for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001) }

# Prior for dispersion
theta ~ dgamma(0.001,0.001)

# Uniform prior for Q
Q ~ dunif(0,3)

# NB-P likelihood using the zero trick
for (i in 1:N){
theta_eff[i]<- theta*(mu[i]^Q)
Y[i] ~ dnegbin(p[i], theta_eff[i])
p[i] <- theta_eff[i]/(theta_eff[i] + mu[i])
log(mu[i]) <- eta[i]
eta[i] <- inprod(beta[], X[i,])
}

",
fill = TRUE)
sink()

# Inits function
inits <- function () {
list(beta  = rnorm(K, 0, 0.1),
     theta = 1,
     Q     = 1)
}
# Parameters to display n output
params <- c("beta",
           "theta",
           "Q"
)
```

```

NBP <- jags(data      = model.data,
              inits     = inits,
              parameters = params,
              model     = "NBPreg.txt",
              n.thin    = 1,
              n.chains  = 3,
              n.burnin  = 2500,
              n.iter    = 5000)
print(NBP, intervals=c(0.025, 0.975), digits=3)
=====
          mu.vect   sd.vect   2.5%   97.5%   Rhat  n.eff
Q           1.485    0.138   1.232   1.771   1.002  1900
beta[1]     1.927    0.061   1.809   2.045   1.001  3200
beta[2]    -4.697    0.266  -5.205  -4.173   1.002  2000
theta       0.461    0.073   0.327   0.612   1.002  2000
pD = 4.0 and DIC = 2348.3

```

The synthetic model is well recovered after 2500 samples, where a previous 2500 were discarded as burn-in. For the parameter value P in the model log-likelihood, this can be determined as $P = 2 - Q$. If $Q = 1.485$, $P = 0.515$. The direct parameterization of α equals $1/0.461$ or 2.169 .

6.5.2 Three-Parameter NB-P Models in Python using Stan

In what follows we present the equivalent code for a three-parameter negative binomial distribution in Python using Stan. In order to avoid misunderstandings regarding the parameterizations used in the generation of synthetic data, we used the R package `MASS` through the package `rpy2`. Notice also that one of the three parameterizations provided by Stan for negative binomial distributions allows us to skip the definition of the `p` parameter used in Code 6.27.

Code 6.28 Negative binomial model with three parameters in Python using Stan. Synthetic data generated with R package `MASS`.

```

=====
import numpy as np
import pystan
import statsmodels.api as sm

from rpy2.robjects import r, FloatVector
from scipy.stats import uniform, binom, nbinom, poisson, gamma

def gen_negbin(N, mu1, theta1):
    """Negative binomial distribution."""

    # Load R package
    r('library(MASS)')

    # Get R functions
    nbinomR = r['rnegbin']

    res = nbinomR(n=N, mu=FloatVector(mu1), theta=FloatVector(theta1))

    return res

# Data
nobs = 750                      # number of obs in model
x1 = uniform.rvs(size=nobs)       # categorical explanatory variable
xb = 2 - 5 * x1                  # linear predictor
exb = np.exp(xb)
theta = 0.5
Q = 1.4
nbpv = gen_negbin(nobs, exb, theta * (exb ** Q))

```

```

X = sm.add_constant(np.transpose(x1))      # format data for input

mydata = {}                                # build data dictionary
mydata['N'] = nobs                         # sample size
mydata['X'] = X                            # predictors
mydata['Y'] = npy                           # response variable
mydata['K'] = X.shape[1]

# Fit
stan_code = """
data{
    int N;
    int K;
    matrix[N,K] X;
    int Y[N];
}
parameters{
    vector[K] beta;
    real<lower=0> theta;
    real<lower=0, upper=3.0> Q;
}
transformed parameters{
    vector[N] mu;
    real<lower=0> theta_eff[N];

    mu = exp(X * beta);
    for (i in 1:N) {
        theta_eff[i] = theta * pow(mu[i], Q);
    }
}
model{
    Y ~ neg_binomial_2(mu, theta_eff);
}
"""

# Run mcmc
fit = pystan.stan(model_code=stan_code, data=mydata, iter=5000, chains=3,
                    warmup=2500, n_jobs=3)

# Output
nlines = 9                                  # number of lines in screen output

output = str(fit).split('\n')
for item in output[:nlines]:
    print(item)
=====
      mean se_mean     sd  2.5%   25%   50%   75%  97.5%  n_eff Rhat
beta[0]  1.96  1.9e-3  0.07  1.81  1.91  1.96  2.01   2.1 1417.0  1.0
beta[1] -4.97  9.0e-3  0.32 -5.58 -5.19 -4.98 -4.77  -4.31 1261.0  1.0
theta    0.52  2.5e-3  0.09  0.35  0.45  0.51  0.58   0.72 1414.0  1.0
Q       1.33  4.4e-3  0.16  1.03  1.21  1.32  1.43   1.67 1378.0  1.0

```

¹ Snippets from http://bebi103.caltech.edu/2015/tutorials/r7_pymc3.html.

¹ <http://docs.scipy.org/doc/scipy-0.17.0/reference/generated/scipy.stats.nbinom.html>

7 GLMs Part III – Zero-Inflated and Hurdle Models

7.1 Bayesian Zero-Inflated Models

Zero-inflated models are mixture models. In the domain of count models, zero-inflated models involve the mixtures of a binary model for zero counts and a count model. It is a mixture model because the zeros are modeled by both the binary and the count components of a zero-inflated model.

The logic of a zero-inflated model can be expressed as

$$\begin{aligned}\Pr(Y = 0) &: \Pr(Bin = 0) + [1 - \Pr(Bin = 0)] \times \Pr(Count = 0) \\ \Pr(Y \geq 0) &: 1 - \Pr(Bin = 0) + PDF_{count}\end{aligned}$$

Thus, the probability of a zero in a zero-inflated model is equal to the probability of a zero in the binary model component (e.g., the logistic) plus one minus the probability of a zero in the binary model times the probability of a zero count in the count model component. The probability that the response is greater than or equal to zero (as in e.g. the Poisson model) is equal to one minus the probability of a zero in the binary component plus the count model probability distribution. The above formulae are valid for all zero-inflated models.

7.1.1 Bayesian Zero-Inflated Poisson Model

We can apply the above formulae for a zero-inflated Poisson–logit model. The count component is a Poisson model and the binary component is a Bernoulli logistic model. Aside from the Poisson PDF, the key formulae for the zero-inflated Poisson–logit model, generally referred to as ZIP, include the probability of zero for a logistic model, $1/[1 + \exp(x\beta)]$, and the probability of a zero Poisson count, $\exp(-\mu)$. Given that $\mu = \exp(x\beta)$, the probability of a zero Poisson count with respect to the linear predictor $x\beta$ is $\exp[-\exp(x\beta)]$. The probability of all but zero counts is $1 - \Pr(0)$, or $1 - \exp[-\exp(x\beta)]$ or $1 - \exp(-\mu)$. The zero-inflated Poisson–logit model log-likelihood is given by the following expressions:

$$\text{if } y == 0, \mathcal{L} = \sum_{i=1}^n \ln \left(\frac{1}{1 + \exp(x_i \beta_l)} \right) + \frac{\exp[-\exp(x_i \beta_p)]}{1 + \exp(x_i \beta_l)}, \quad (7.1)$$

$$\text{if } y_2 > 0, \mathcal{L} = \sum_{i=1}^n -\ln [1 + \exp(x_i \beta_l) - \exp(x_i \beta_p) + y_2 x_i \beta_p - \ln \Gamma(y_2 + 1)], \quad (7.2)$$

where the subscripts p and l stand for betas on coefficients from the Poisson and logistic components respectively. The mean of the Poisson distribution is given by μW , where W is the outcome of the Bernoulli draw. This can be expressed as

$$\begin{aligned} W &\sim \text{Bernoulli}(\pi), \\ Y &\sim \text{Poisson}(\mu W), \\ \log(\mu) &= \beta_1 + \beta_2 X, \\ \text{logit}(\pi) &= \gamma_1 + \gamma_2 X. \end{aligned} \quad (7.3)$$

The interpretation of the zero-inflated counts differs substantially from how zero counts are interpreted for hurdle models. Hurdle models are two-part models, each component being separately modeled. The count component is estimated as a zero-truncated model. The binary component is usually estimated as a 0, 1 logistic model, with the logistic “1” values representing all positive counts from the count component. In a zero-inflated model, however, the zeros are modeled by both the count and binary components. That is where the mixture occurs.

Some statisticians call zero-inflated-model zeros, estimated using the binary component, false zeros, and the zeros from the count model true zeros. This terminology is particularly common in ecology, environmental science, and now in the social sciences. The notion behind labeling “true” and “false” zeros comes from considering them as errors. For instance, when one is counting the number of exoplanets identified each month over a two-year period in a given area of the Milky Way, it may be that for a specific month none is found, even though there were no problems with the instruments and weather conditions were excellent. These are true zeros. If exoplanets are not observed owing to long-lasting periods of inclement weather, or if the instruments searching for exoplanets broke down, the zeros recorded for exoplanet counts in these circumstances would be considered as “false” zeros. The zero counts from the count component are assigned as true zeros; the zeros of the binary component are assigned as false zeros.

When interpreting a zero-inflated model, it must be remembered that the binary component models the zero counts, not the 1s. This is unlike hurdle models, or even standard binary response models. A logistic model, for example, models the probability that $y == 1$. For a zero-inflated Poisson–logit model, the logistic component models the probability that $y == 0$. This is important to keep in mind when interpreting zero-inflated model results.

Zero-Inflated Poisson Model using JAGS

We shall use synthetic data created by the `zripois` function to build a synthetic data set with specifically defined coefficient values. Since we are estimating parameters using Bayesian methodology, we speak of parameter means and not coefficients,

The key to the function is that more zero counts are generated than are allowed for a true Poisson distribution. Again, given a specific value for the mean of the count variable being modeled, we expect a certain number of zero counts on the basis of the Poisson distribution. If there are substantially more zero counts than are allowed by the distributional assumptions of the Poisson model, then we have a zero-inflated model. We have created a JAGS model for the zero-inflated Poisson model, which is displayed in Code 7.1. Note that the lines in the log-likelihood that define $\gamma_{[i]}$ and $w_{[i]}$, and the two lines below the log-likelihood that define w , are the key lines that mix the Poisson and Bernoulli logit models. Moreover, linear predictors are specified for both the count and binary components of the model

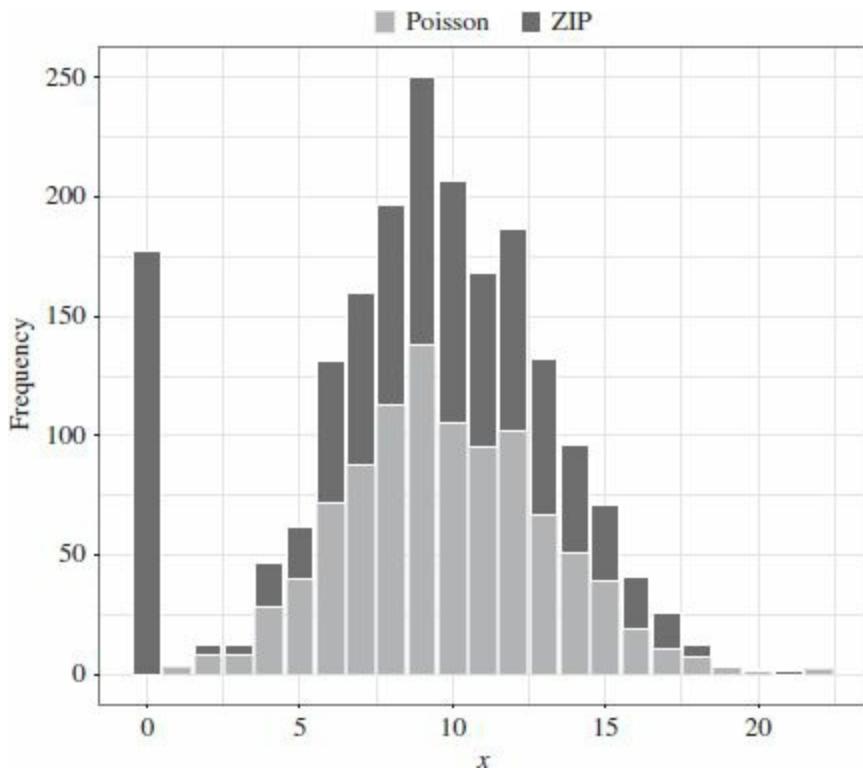


Figure 7.1 For comparison, random variables drawn from Poisson (lighter) and zero-inflated Poisson (darker) distributions.

... in R using JAGS

The JAGS code in Code 7.1 provides posterior means, standard errors, and credible intervals for each parameter from both components of the model.

Code 7.1 Bayesian zero-inflated Poisson model in R using JAGS.

```
=====  
require(MASS)
```

```

require(R2jags)
require(VGAM)
set.seed(141)
nobs <- 1000
x1 <- runif(nobs)
xb <- 1 + 2.0 * x1
xc <- 2 - 5.0 * x1
exb <- exp(xb)
exc <- 1/(1 + exp(-xc))
zipy <- rzipois(n=nobs, lambda=exb, pstr0=exc)
zipdata <- data.frame(zipy,x1)

Xc <- model.matrix(~ 1 + x1, data=zipdata)
Xb <- model.matrix(~ 1 + x1, data=zipdata)

Kc <- ncol(Xc)
Kb <- ncol(Xb)

model.data <- list(Y = zipdata$zipy, # response
                     Xc = Xc, # covariates
                     Kc = Kc, # number of betas
                     Xb = Xb, # covariates
                     Kb = Kb, # number of gammas
                     N = nobs)
ZIPPOIS<-"model{
  # Priors - count and binary components
  for (i in 1:Kc) { beta[i] ~ dnorm(0, 0.0001)}
  for (i in 1:Kb) { gamma[i] ~ dnorm(0, 0.0001)}

  # Likelihood
  for (i in 1:N) {
    W[i] ~ dbern(1 - Pi[i])
    Y[i] ~ dpois(W[i] * mu[i])
    log(mu[i]) <- inprod(beta[], Xc[i,])
    logit(Pi[i]) <- inprod(gamma[], Xb[i,])
  }
}"

W <- zipdata$zipy
W[zipdata$zipy > 0] <- 1

inits <- function() {
  list(beta = rnorm(Kc, 0, 0.1),
       gamma = rnorm(Kb, 0, 0.1),
       W = W)}
params <- c("beta", "gamma")

ZIP <- jags(data = model.data,
             inits = inits,
             parameters = params,
             model = textConnection(ZIPPOIS),
             n.thin = 1,
             n.chains = 3,
             n.burnin = 4000,
             n.iter = 5000)
print(ZIP, intervals = c(0.025, 0.975), digits=3)
=====

mu.vect sd.vect 2.5% 7.5% Rhat n.eff
beta[1] 1.019 0.045 0.932 1.104 1.103 25
beta[2] 1.955 0.058 1.843 2.066 1.096 26
gamma[1] 1.913 0.166 1.588 2.241 1.013 160
gamma[2] -4.828 0.319 -5.472 -4.208 1.014 150
deviance 3095.598 11.563 3079.409 3122.313 1.004 710

```

pD = 66.7 and DIC = 3162.3

The count or Poisson component of the output is provided in the upper beta means and the binary or logit component in the lower gamma means. Notice that the parameter values we gave the synthetic data are close to the means of the posterior parameters displayed in the output.

... in Python using Stan

The implementation of a zero-inflated Poisson model in Python using Stan requires a few extra

ingredients. First we repeat the strategy of using an R function to generate the synthetic data, since no equivalent well-established package is currently available in Python. Moreover, Stan does not allow for discrete parameters and, consequently, we include the log-likelihood instead of the w parameter used in Code 7.1. Using this simple adjustment and remembering that, in Stan,

```
model{
    y ~ poisson(lambda);
}
```

can be expressed as

```
model{
    increment_log_prob(poisson_log(y, lambda));
}
```

the zero-inflated Poisson model studied before can be implemented as

Code 7.2 Bayesian zero-inflated Poisson in Python using Stan.

```
=====
import numpy as np
import pystan
import statsmodels.api as sm

from rpy2.robj import r, FloatVector
from scipy.stats import uniform, norm

def zipoisson(N, lambda_par, psi):
    """Zero inflated Poisson sampler."""

    # Load R package
    r('library(VGAM')

    # Get R functions
    zipoissonR = r['rziropis']

    res = zipoissonR(N, FloatVector(lambda_par),
                      pstr0=FloatVector(psi))

    return np.array([int(item) for item in res])

# Data
np.random.seed(141)                      # set seed to replicate example
nobs = 5000                                # number of obs in model

x1 = uniform.rvs(size=nobs)

xb = 1 + 2.0 * x1                          # linear predictor
xc = 2 - 5.0 * x1

exb = np.exp(xb)
exc = 1.0 / (1.0 + np.exp(-xc))

zipy = zipoisson(nobs, exb, exc)          # create y as adjusted

X = np.transpose(x1)
X = sm.add_constant(X)

mydata = {}                                  # build data dictionary
mydata['N'] = nobs                         # sample size
mydata['Xb'] = X                            # predictors
mydata['Xc'] = X                            # response variable
mydata['Kb'] = X.shape[1]                   # number of coefficients
mydata['Kc'] = X.shape[1]

# Fit
stan_code = """
data{
    int N;
    int Kb;
```

```

int Kc;
matrix[N, Kb] Xb;
matrix[N, Kc] Xc;
int Y[N];
}
parameters{
    vector[Kc] beta;
    vector[Kb] gamma;
}
transformed parameters{
    vector[N] mu;
    vector[N] Pi;
    mu = exp(Xc * beta);
    for (i in 1:N) Pi[i] = inv_logit(Xb[i] * gamma);
}
model{
    real LL[N];
    for (i in 1:N) {
        if (Y[i] == 0) {
            LL[i] = log_sum_exp(bernoulli_lpmf(1|Pi[i]),
                                bernoulli_lpmf(0|Pi[i]) + poisson_lpmf(Y[i]|mu[i]));
        }else{
            LL[i] = bernoulli_lpmf(0|Pi[i] + poisson_lpmf(Y[i]|mu[i]));
        }
        target += LL;
    }
}
"""
# Run mcmc
fit = pystan.stan(model_code=stan_code, data=mydata, iter=5000, chains=3,
                    warmup=4000, n_jobs=3)

# Output
nlines = 9 # number of lines in screen output

output = str(fit).split('\n')
for item in output[:nlines]:
    print(item)
=====
      mean se_mean   sd  2.5%   25%   50%   75% 97.5% n_eff Rhat
beta[0]    1.01  1.2e-3  0.03  0.95  0.99  1.01  1.03  1.06 576.0  1.0
beta[1]     2.0   1.6e-3  0.04  1.93  1.97  2.0   2.02  2.07 570.0  1.0
gamma[0]    2.06  4.3e-3  0.11  1.85  1.99  2.06  2.13  2.27 619.0  1.0
gamma[1]   -5.15  8.3e-3  0.21 -5.55 -5.29 -5.14 -5.0  -4.75 623.0  1.0

```

Notice that our model is written in two parts, so that there is a probability `Pi` of drawing a zero and a probability `1 - Pi` of drawing from a Poisson distribution with mean `mu`. The expression `log_sum_exp(11, 12)` is a more arithmetically stable version of `log(exp(11) + exp(12))` (see the manual [Team Stan, 2016](#)).

7.1.2 Bayesian Zero-Inflated Negative Binomial Model

The logic of the zero-inflated negative binomial (ZINB) is the same as that of the zero-inflated Poisson (ZIP). The zero-inflated negative binomial model is preferred if there are in excess of 20% zero counts in the response term. The model can accommodate more extra correlation in the data than a ZIP model can. Of course, if the data is underdispersed, the ZINB model should not be used. Many statisticians, though, consider the ZINB to be the preferred count model when there are excessive zero counts in the data. Hurdle models can also be used effectively with models having excessive zero counts, and studies in the recent literature appear to bear this out. In addition, a negative binomial hurdle model can be used on models where there are too few zero counts on the basis of the distribution assumptions of the model. However, it is easier to interpret both the binary and count components in a zero-inflated model than in a hurdle model. We advise trying both models on zero-inflated count data and using the fit statistics to determine which model is to be preferred in

a given situation.

Below, we create synthetic zero-inflated negative binomial data using the `rzinegbin` function. The posterior parameter mean values assigned to the synthetic data are:

```
Binary: intercept=1; x1=2.0; x2=1.5
Count: intercept=2; x1=-5.0; x2=3.0
Alpha: 2 (indirect); 0.5 (direct)
note: the indirect is the inverse of the direct: alpha=1/size
```

Code 7.3 Zero-inflated negative binomial synthetic data in R.

```
=====
require(MASS)
require(R2jags)
require(VGAM)

set.seed(141)
nobs <- 1000
x1 <- runif(nobs)
x2 <- rbinom(nobs, size=1, 0.6)
xb <- 1 + 2.0*x1 + 1.5*x2
xc <- 2 - 5.0*x1 + 3*x2
exb <- exp(xb)
exc <- 1/(1 + exp(-xc))
alpha <- 2
zinby <- rzinegbin(n=nobs, mumb = exb, size=1/alpha, pstr0=exc)
zinbdata <- data.frame(zinby,x1,x2)
=====
```

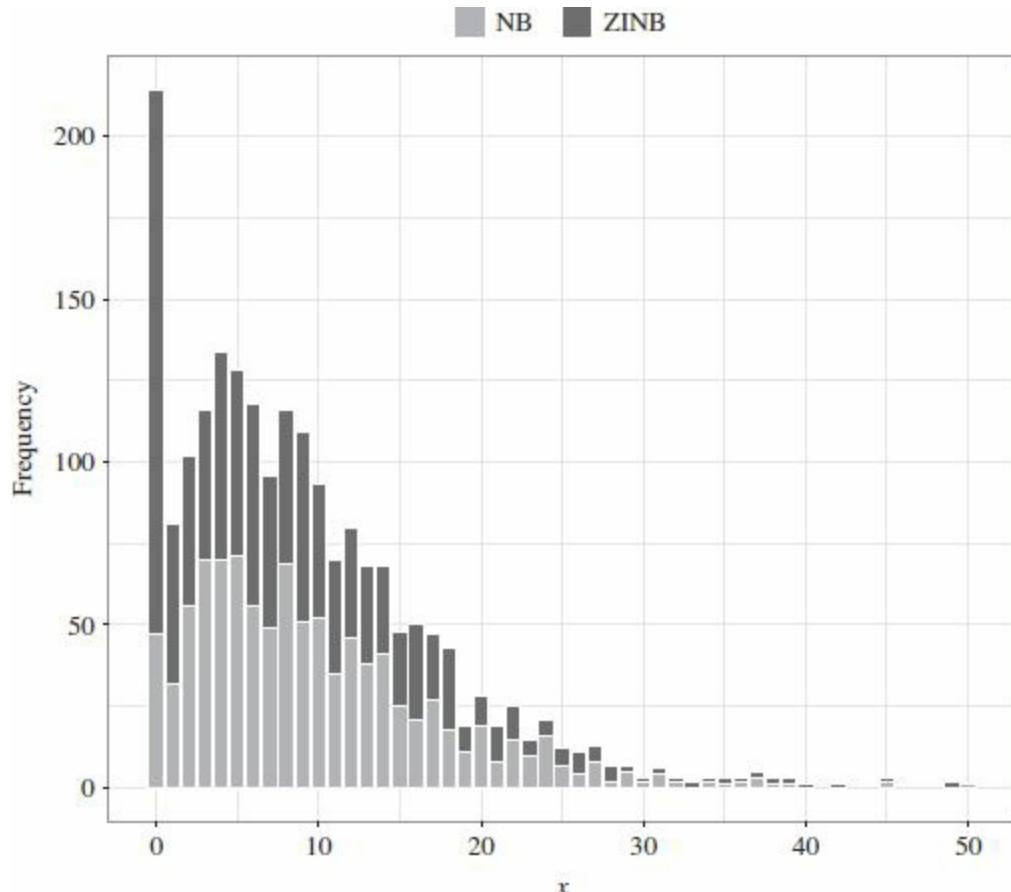


Figure 7.2 For comparison, random variables drawn from negative binomial (lighter) and zero-inflated negative binomial

(darker) distributions.

... in R using JAGS

The code for creating a zero-inflated negative binomial model is found below. We use the direct parameterization for the negative binomial dispersion parameter, but provide the code for converting the model to indirect parameterization below the model output.

Code 7.4 Bayesian zero-inflated negative binomial model using JAGS.

```
=====
Xc <- model.matrix(~ 1 + x1+x2, data=zinbdata)
Xb <- model.matrix(~ 1 + x1+x2, data=zinbdata)

Kc <- ncol(Xc)
Kb <- ncol(Xb)

ZI.data <- list(Y = zinbdata$zinby,           # response
                 Xc = Xc,                  # covariates
                 Kc = Kc,                  # number of betas
                 Xb = Xb,                  # covariates
                 Kb = Kb,                  # covariates
                 N = nrow(zinbdata))      # number of gammas

sink ("ZINB.txt")
cat("
model{
# Priors...
for(i in 1: kc...
model{
# Priors - count and binary components
for (i in 1:Kc) { beta[i] ~ dnorm(0, 0.0001)}
for (i in 1:Kb) { gamma[i] ~ dnorm(0, 0.0001)}
alpha ~ dunif(0.001, 5)
# Likelihood
for (i in 1:N) {
W[i] ~ dbern(1 - Pi[i])
Y[i] ~ dnegbin(p[i], 1/alpha)
p[i] <- 1/(1 + alpha * W[i] * mu[i])
# Y[i] ~ dnegbin(p[i],alpha) indirect
# p[i] <- alpha/(alpha + mueff[i]) indirect
mueff[i] <- W[i] * mu[i]
log(mu[i])   <- inprod(beta[], Xc[i,])
logit(Pi[i]) <- inprod(gamma[], Xb[i,])
}
}
",fill = TRUE)
sink()

W <- zinbdata$zinby
W[zinbdata$zinby > 0] <- 1

inits <- function () {
  list(beta  = rnorm(Kc, 0, 0.1),
       gamma = rnorm(Kb, 0, 0.1),
       W     = W)}
params <- c("beta", "gamma", "alpha")

ZINB   <- jags(data = model.data,
                inits = inits,
                parameters = params,
                model = textConnection(ZINB),
                n.thin = 1,
                n.chains = 3,
                n.burnin = 4000,
                n.iter = 5000)
print(ZINB, intervals=c(0.025, 0.975), digits=3)

# Figures for parameter trace plots and histogramss
source("CH-Figures.R")
out <- ZINB$BUGSoutput
MyBUGSHist(out,c(uNames("beta",Kc),"alpha", uNames("gamma",Kb)))
MyBUGSChains(out,c(uNames("beta",Kc),"alpha", uNames("gamma",Kb)))
```

	mu.vect	sd.vect	2.5%	97.5%	Rhat	n.eff
alpha	2.163	0.269	1.702	2.780	1.001	3000
beta[1]	0.988	0.300	0.441	1.624	1.007	630
beta[2]	2.016	0.393	1.179	2.765	1.019	560
beta[3]	1.751	0.184	1.409	2.126	1.003	890
gamma[1]	1.831	0.318	1.206	2.446	1.013	170
gamma[2]	-4.742	0.500	-5.741	-3.753	1.018	150
gamma[3]	2.812	0.266	2.314	3.381	1.010	220
deviance	2551.639	43.651	2471.052	2644.143	1.003	3000

pD = 953.0 and DIC = 3504.7

An indirect parameterization of the dispersion parameter may be obtained by substituting the two lines below for the corresponding lines in Code 7.4.

```
Y[i] ~ dnegbin(p[i], alpha)
p[i] <- alpha/(alpha + W[i]*mu[i])
```

Another run using the above lines results in the output below. Note that the only statistically substantial change is the inversion of the alpha parameter statistic.

	mu.vect	sd.vect	2.5%	97.5%	Rhat	n.eff
alpha	0.486	0.060	0.375	0.609	1.013	1500
beta[1]	1.078	0.284	0.511	1.614	1.007	380
beta[2]	1.911	0.371	1.213	2.659	1.011	320
beta[3]	1.739	0.192	1.371	2.138	1.007	320
gamma[1]	1.862	0.322	1.210	2.460	1.014	240
gamma[2]	-4.741	0.513	-5.705	-3.752	1.010	250
gamma[3]	2.794	0.285	2.243	3.371	1.020	240
deviance	2541.680	44.556	2460.229	2638.626	1.020	970

pD = 991.1 and DIC = 3532.8

...in Python using Stan

The equivalent Python code using Stan is displayed below. In working with any negative binomial model we emphasize that the reader should be careful to check the parameterizations available for this distribution. At the moment Stan provides three different parameterizations; here we use `neg_binomial_2` (Team Stan, 2016).

Code 7.5 Bayesian zero-inflated negative binomial model in Python using Stan.

```

x1 = uniform.rvs(size=nobs)
x2 = bernoulli.rvs(0.6, size=nobs)
xb = 1.0 + 2.0 * x1 + 1.5 * x2 # linear predictor
xc = 2.0 - 5.0 * x1 + 3.0 * x2

exb = np.exp(xb)
exc = 1 / (1 + np.exp(-xc))
alpha = 2

# Create y as adjusted
zinby = gen_zinegbinom(nobs, exb, exc, alpha)

X = np.column_stack((x1,x2))
X = sm.add_constant(X)

mydata = {} # build data dictionary
mydata['Y'] = zinby # response variable
mydata['N'] = nobs # sample size
mydata['Xb'] = X # predictors
mydata['Xc'] = X
mydata['Kb'] = X.shape[1] # number of coefficients
mydata['Kc'] = X.shape[1]

# Fit
stan_code = """
data{
    int N;
    int Kb;
    int Kc;
    matrix[N, Kb] Xb;
    matrix[N, Kc] Xc;
    int Y[N];
}
parameters{
    vector[Kc] beta;
    vector[Kb] gamma;
    real<lower=0> alpha;
}
transformed parameters{
    vector[N] mu;
    vector[N] Pi;

    mu = exp(Xc * beta);
    for (i in 1:N) Pi[i] = inv_logit(Xb[i] * gamma);
}
model{
    vector[N] LL;
    for (i in 1:N) {
        if (Y[i] == 0) {
            LL[i] = log_sum_exp(bernoulli_lpmf(1|Pi[i]),
                                bernoulli_lpmf(0|Pi[i]) +
                                neg_binomial_2_lpmf(Y[i]|mu[i], 1/alpha));
        } else {
            LL[i] = bernoulli_lpmf(0|Pi[i]) +
                    neg_binomial_2_lpmf(Y[i]| mu[i], 1/alpha);
        }
        target += LL;
    }
}
"""
"""

# Run mcmc
fit = pystan.stan(model_code=stan_code, data=mydata, iter=7000, chains=3,
                   warmup=3500, n_jobs=3)

# Output
nlines = 12 # number of lines in screen output

output = str(fit).split('\n')
for item in output[:nlines]:
    print(item)
=====
      mean se_mean sd 2.5% 25% 50% 75% 97.5% n_eff Rhat
beta[0]   1.15  3.6e-3 0.14  0.88 1.06 1.15 1.25 1.44 1595.0 1.0
beta[1]    1.8   4.7e-3 0.19  1.43 1.68 1.81 1.94 2.17 1651.0 1.0
beta[2]    1.44  2.3e-3 0.11  1.23 1.37 1.44 1.51 1.65 2138.0 1.0
gamma[0]   1.96  3.7e-3 0.16  1.64 1.86 1.96 2.07 2.27 1832.0 1.0
gamma[1]   -4.83  7.5e-3 0.31 -5.46 -5.04 -4.83 -4.62 -4.25 1702.0 1.0

```

```
gamma[2]    2.89 4.4e-3 0.18  2.55 2.77 2.89 3.02 3.26 1706.0 1.0
alpha       1.9  3.8e-3 0.16  1.62 1.79 1.9   2.01 2.25 1756.0 1.0
```

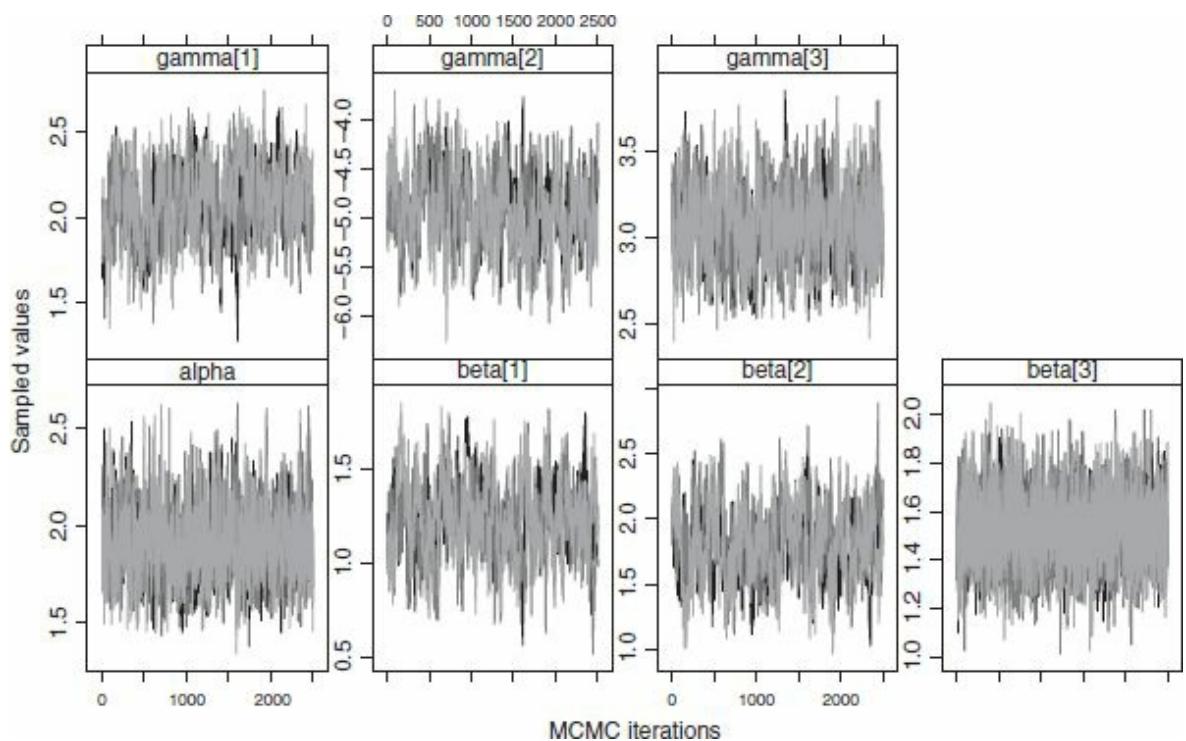


Figure 7.3 MCMC chains for the regression parameters of the zero-inflated negative binomial model.

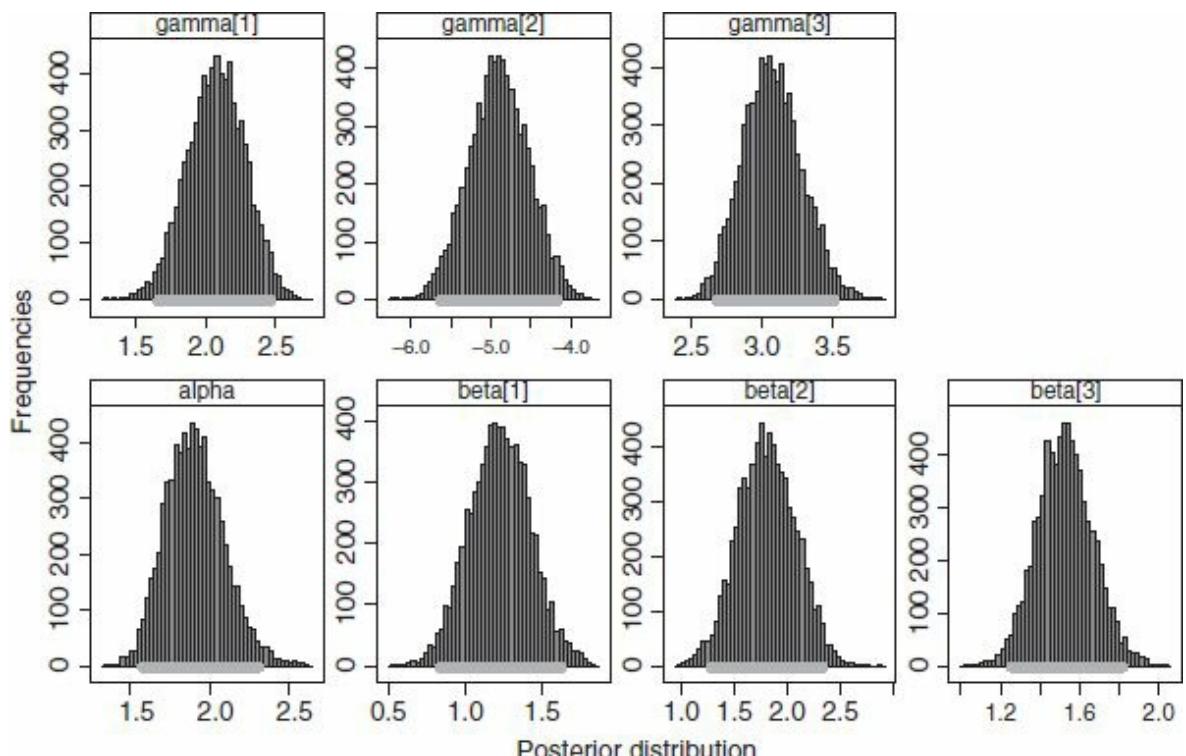


Figure 7.4 Histograms of the MCMC iterations for each parameter in the zero-inflated negative binomial model. The thick

line at the base of each histogram represents the 95% credible interval.

7.2 Bayesian Hurdle Models

In a hurdle model, or zero-altered model, it is assumed that two independent processes generate the data, but there is a difference from the zero-inflated models. Only the binary process generates zero counts, unlike in zero-inflated models where both the count and binary components generate zeros. Hurdle models are therefore referred to as two-part models, while zero-inflated models are considered as mixture models. Therefore, some process in nature may be thought to trigger the presence of a given phenomenon, for instance the minimum mass for halos to be able to host stars. The second count distribution will then model the non-zero component, for example, the star-formation–metallicity relation.

A hurdle model can be constructed for both discrete and continuous distributions. The binary component will generally be modeled using a Bernoulli distribution, while the other component can be cast as either a count or continuous distribution. In fact, though, the binary component can be modeled as being right censored at one count, but we shall not discuss this alternative here. In the following we provide non-exhaustive examples of hurdle models for both discrete and continuous data in the hope that readers will be able to adjust the templates discussed herein for their own needs.

7.2.1 Bayesian Poisson–Logit Hurdle Model

Hurdle count models are two-part models. Each part, or component, of the model can be estimated separately. The goal is to separate a model into two components – a binary component for modeling 0, 1 data and a count component modeling a zero-truncated count model. For the Poisson–logit hurdle model, the log-likelihood may be expressed as

$$\mathcal{L} = \ln(f(0)) + \{\ln[1 - f(0)] + \ln P(t)\}, \quad (7.4)$$

where $f(0)$ is the probability of a zero count and $P(t)$ is the probability of a positive count. The way to interpret the above equation is that the hurdle model log-likelihood is the natural log of the probability of $y = 0$, plus the log of $y = 1$, plus the log of the probability that y is a positive count ([Hilbe, 2014](#)).

The model is to be used when there are either too few or too many zero counts in the data on the basis of the probability distribution of the count model. This model may therefore be used on underdispersed data as well as on overdispersed data. Simulation studies have demonstrated that hurdle models, which are sometimes referred to as zero-altered models, may be preferable to zero-inflated models for dealing with overdispersion due to excessive zero counts in the data. This is the case for both maximum likelihood and Bayesian models. However, hurdle models are not always to be preferred in individual cases. A comparative test of DIC statistics should be made.

The two-part Poisson–logit hurdle model is generally regarded as the basic hurdle model. The binary component of the hurdle model is a logistic model for which the probability of a zero value is

$$f(0) = \frac{1}{1 + \exp(x\beta)}. \quad (7.5)$$

The probability of the value 1 is $1 - f(0)$:

$$f(1) = \frac{1}{1 + \exp(-x\beta)}. \quad (7.6)$$

For the count component, the distribution is truncated at zero. However, in order to retain its status as a probability distribution, the counts must sum to 1. This is done by subtracting the log of 1 minus the probability of a Poisson zero from the original Poisson log-likelihood. The probability of a Poisson count of zero is $\exp(-\mu)$. Since $\mu = \exp(x\beta)$, the log-likelihood of a zero-truncated Poisson distribution is

$$L(\beta; y) = \sum_{i=1}^n \{y_i \ln(x_i\beta) - \exp(x_i\beta) - \ln \Gamma(y_i + 1) - \ln[1 - \exp(-\exp(x_i\beta))]\}. \quad (7.7)$$

This value and the above probability of a logistic zero count constitute the terms of the Poisson–logit hurdle log-likelihood. This value is found in the JAGS model below.

... in R using JAGS

For the JAGS code below, the binary component is set up by creating a separate variable with value 1 for all counts greater than zero, and 0 for zero counts. The probability of the logistic component is based on this bifurcated data. The count component is a zero-truncated Poisson model. First we generate the data:

Code 7.6 Synthetic data for Poisson–logit hurdle zero-altered models.

```
=====
rztp <- function(N, lambda){
  p <- runif(N, dpois(0, lambda),1)
  ztp <- qpois(p, lambda)
  return(ztp)
}

# Sample size
nobs <- 1000

# Generate predictors, design matrix
x1 <- runif(nobs, -0.5, 2.5)
xb <- 0.75 + 1.5*x1

# Construct Poisson responses
exb <- exp(xb)
poy <- rztp(nobs, exb)
pdata <- data.frame(poy, x1)

# Generate predictors for binary part
xc <- -3 + 4.5*x1

# Construct filter
pi <- 1/(1+exp((xc)))
bern <- rbinom(nobs, size =1, prob=1-pi)

# Add structural zeros
pdata$poy <- pdata$poy*bern
=====
```

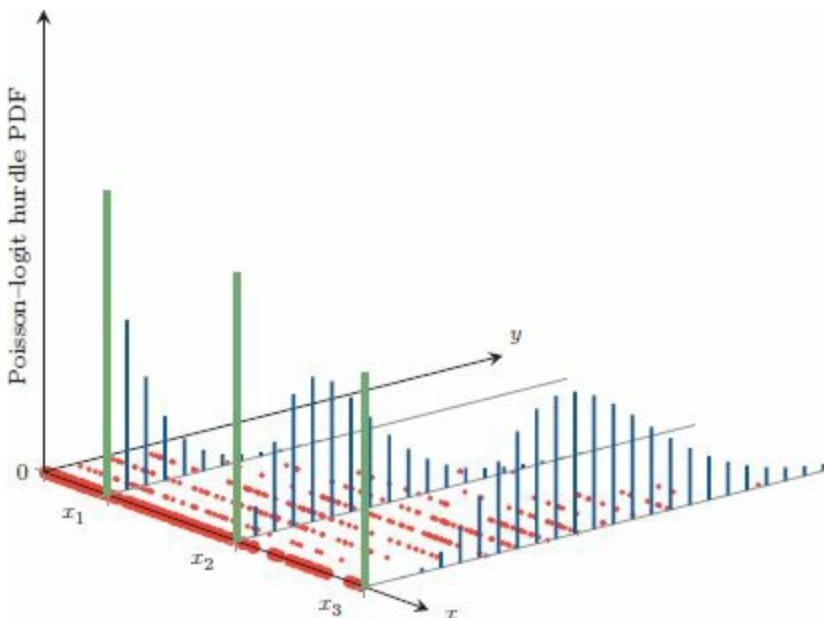


Figure 7.5 Illustration of Poisson–logit hurdle-distributed data.

Code 7.7 Bayesian Poisson–logit hurdle zero-altered models.

```
=====
require(R2jags)
Xc <- model.matrix(~ 1 + x1, data=pdata)
Xb <- model.matrix(~ 1 + x1, data=pdata)
Kc <- ncol(Xc)
Kb <- ncol(Xb)

model.data <- list(
  Y = pdata$poy, # response
```

```

Xc = Xc,           # covariates
Xb = Xb,           # covariates
Kc = Kc,           # number of betas
Kb = Kb,           # number of gammas
N = nrow(pdata), # sample size
Zeros = rep(0, nrow(pdata)))

sink("HPL.txt")
cat("
model{
  # Priors beta and gamma
  for (i in 1:Kc) {beta[i] ~ dnorm(0, 0.0001)}
  for (i in 1:Kb) {gamma[i] ~ dnorm(0, 0.0001)}

  # Likelihood using zero trick
  C <- 10000
  for (i in 1:N) {
    Zeros[i] ~ dpois(-ll[i] + C)
    LogTruncPois[i] <- log(Pi[i]) +
Y[i] * log(mu[i]) - mu[i] -(log(1 - exp(-mu[i])) +
loggam(Y[i] + 1) )
    z[i] <- step(Y[i] - 0.0001)
    l1[i] <- (1 - z[i]) * log(1 - Pi[i])
    l2[i] <- z[i] * (log(Pi[i]) + LogTruncPois[i])
    ll[i] <- l1[i] + l2[i]

    log(mu[i]) <- inprod(beta[], Xc[i,])
    logit(Pi[i]) <- inprod(gamma[], Xb[i,])
  }
}", fill = TRUE)
sink()

inits <- function () {
  list(beta = rnorm(Kc, 0, 0.1),
        gamma = rnorm(Kb, 0, 0.1))}
params <- c("beta", "gamma")

ZAP <- jags(data      = model.data,
             inits      = inits,
             parameters = params,
             model      = "HPL.txt",
             n.thin     = 1,
             n.chains   = 3,
             n.burnin   = 4000,
             n.iter     = 6000)
print(ZAP, intervals=c(0.025, 0.975), digits=3)
=====
          mu.vect sd.vect      2.5%     97.5%      Rhat  n.eff
beta[1]       0.737  0.032      0.674      0.800    1.005    820
beta[2]       1.503  0.016      1.472      1.534    1.005    670
gamma[1]      -2.764  0.212     -3.190     -2.354    1.004    650
gamma[2]       5.116  0.298      4.550      5.728    1.004    610
deviance  20004080.026  2.741 20004076.529 20004086.654  1.000      1

pD = 4.0 and DIC = 20004084.0

```

The beta parameters are for the Poisson component, with the intercept on the upper line. The logistic components are the lower gamma parameters.

It might be instructive to look at the results of a maximum likelihood Poisson–logit hurdle model on the same data as used for the above Bayesian model. This should inform us about the variability in parameter estimates based on the synthetic data. The code follows:

```

library(MASS)
library(pscl)

# Run hurdle synthetic data code above
hlpoi <- hurdle(poy ~ x1,
                  dist = "poisson",
                  zero.dist = "binomial",

                  link = "logit",
                  data = pdata)

```

```

summary(hlpoi)
      Estimate Std. Error   z value   Pr(>|z|)
(Intercept)  0.68905   0.03426   20.11 <2e-16 ***
x1          1.52845   0.01756   87.03 <2e-16 ***
Zero hurdle model coefficients (binomial with logit link):
      Estimate Std. Error   z value   Pr(>|z|)
(Intercept) -1.5993    0.1890   -8.463 <2e-16 ***
x1          4.2387    0.3327   12.739 <2e-16 ***
Log-likelihood: -1680 on 4 Df

```

The fact that we used only 750 observations in the model adds variability to the synthetic data as well as to the JAGS model, which is based on MCMC sampling. Increasing the sample size will increase the likelihood that the posterior means that we specified in the synthetic data will be reflected in the JAGS model output.

...in Python using Stan

Code 7.8 Bayesian Poisson–logit hurdle model in Python using Stan.

```

=====
import numpy as np
import pystan
import statsmodels.api as sm

from scipy.stats import uniform, bernoulli, poisson

def ztp(N, lambda_):

    """zero-truncated Poisson distribution"""
    temp = [poisson.pmf(0, item) for item in lambda_]
    p = [uniform.rvs(loc=item, scale=1-item) for item in temp]
    ztp = [int(poisson.ppf(p[i], lambda_[i])) for i in range(len(p))]

    return np.array(ztp)

# Data
np.random.seed(141)                         # set seed to replicate example
nobs = 750                                     # number of obs in model

x1 = uniform.rvs(size=nobs)

xb = 1.0 + 4.0 * x1                           # linear predictor
exb = np.exp(xb)
poy = ztp(nobs, exb)

xc = -1.0 + 3.5 * x1                          # construct filter
pi = 1.0/(1.0 + np.exp(xc))
bern = [bernoulli.rvs(1-pi[i]) for i in range(nobs)]

poy = [poy[i]*bern[i] for i in range(nobs)] # add structural zeros

X = np.transpose(x1)
X = sm.add_constant(X)

# Prepare data for Stan
mydata = {}                                     # build data dictionary
mydata['Y'] = poy                               # response variable
mydata['N'] = nobs                             # sample size
mydata['Xb'] = X                                # predictors
mydata['Xc'] = xc                              # number of coefficients
mydata['Kb'] = X.shape[1]                        # number of coefficients
mydata['Kc'] = X.shape[1]

# Fit
stan_code = """
data{
  int<lower=0> N;
  int<lower=0> Kb;
  int<lower=0> Kc;
  matrix[N, Kb] Xb;
  matrix[N, Kc] Xc;
  int<lower=0> Y[N];
}

```

```

parameters{
  vector[Kc] beta;
  vector[Kb] gamma;
  real<lower=0, upper=5.0> r;
}
transformed parameters{
  vector[N] mu;
  vector[N] Pi;

  mu = exp(Xc * beta);
  for (i in 1:N) Pi[i] = inv_logit(Xb[i] * gamma);
}
model{
  for (i in 1:N) {
    (Y[i] == 0) ~ bernoulli(1-Pi[i]);
    if (Y[i] > 0) Y[i] ~ poisson(mu[i]) T[1,];
  }
}
"""

# Run mcmc
fit = pystan.stan(model_code=stan_code, data=mydata, iter=7000, chains=3,
                    warmup=4000, n_jobs=3)

# Output
nlines = 10 # number of lines in screen output

output = str(fit).split('\n')
for item in output[:nlines]:
  print(item)
=====
      mean se_mean sd 2.5% 25% 50% 75% 97.5% n_eff Rhat
beta[0]  1.01  1.0e-3 0.03  0.94  0.99  1.01  1.03  1.07 913.0  1.0
beta[1]   4.0   1.3e-3 0.04  3.92  3.97  4.0   4.02  4.07 906.0  1.0
gamma[0] -1.07  5.3e-3 0.16 -1.39 -1.17 -1.06 -0.95 -0.77 904.0  1.01
gamma[1]  3.66  0.01  0.33  3.04  3.43  3.64  3.87  4.32 897.0  1.01
r        2.32  0.05  1.44  0.09  1.05  2.26  3.55  4.82 884.0  1.0

```

7.2.2 Bayesian Negative Binomial–Logit Hurdle Model

The negative binomial hurdle model is also known as a zero-altered negative binomial, or ZANB. Like the zero-altered Poisson, the binary component typically used is a Bernoulli logistic regression. Probit, complementary loglog or some variety of right-censored-at-one-count model can also be used for the binary component. The negative binomial hurdle model is used on data with either excessive zero counts in the response or at times too few zero counts. In the case where there are too few zero counts, a zero-inflated model cannot be used. The hurdle model is a good way to deal with such data. When the ZAP model is extra-dispersed and we have little idea of the origin of the extra dispersion, a ZANB should be used. Here we shall use the model on the synthetic ZAP data created in the previous section. Since the count component of the data is designed to be Poisson distributed, we expect that the zero-truncated negative binomial component has a value of alpha (a direct parameterization of the dispersion parameter) close to zero. The deviance statistic of the ZANB model is some 250 less than the ZAP model on the same data; α is 0.002, as expected. A Bayesian generalized Poisson hurdle model can be used when the data is underdispersed. The code for the ZAGP model is on the book's website.

...in R using JAGS

Code 7.9 Zero-altered negative binomial (ZANB) or NB hurdle model in R using JAGS.

```
=====
require(R2jags)
Xc <- model.matrix(~ 1 + x1, data = pdata)
Xb <- model.matrix(~ 1 + x1, data = pdata)
Kc <- ncol(Xc)
Kb <- ncol(Xb)
```

```

model.data <- list(
  Y = pdata$poy,
  Xc = Xc,
  Xb = Xb,
  Kc = Kc,           # number of betas - count
  Kb = Kb,           # number of gammas - binary
  N = nrow(pdata),
  Zeros = rep(0, nrow(pdata)))

sink("NBH.txt")
cat(""
model{
  # Priors beta and gamma
  for (i in 1:Kc) {beta[i] ~ dnorm(0, 0.0001)}
  for (i in 1:Kb) {gamma[i] ~ dnorm(0, 0.0001)}

  # Prior for alpha
  alpha ~ dunif(0.001, 5)

  # Likelihood using zero trick
  C <- 10000
  for (i in 1:N) {
    Zeros[i] ~ dpois(-ll[i] + C)
    LogTruncNB[i] <- 1/alpha * log(u[i]) +
      Y[i] * log(1 - u[i]) + loggam(Y[i] + 1/alpha) -
      loggam(1/alpha) - loggam(Y[i] + 1) -
      log(1 - (1 + alpha * mu[i])^(-1/alpha))
    z[i] <- step(Y[i] - 0.0001)
    l1[i] <- (1 - z[i]) * log(1 - Pi[i])
    l2[i] <- z[i] * (log(Pi[i]) + LogTruncNB[i])
    ll[i] <- l1[i] + l2[i]
    u[i] <- 1/(1 + alpha * mu[i])
    log(mu[i]) <- inprod(beta[], Xc[i,])
    logit(Pi[i]) <- inprod(gamma[], Xb[i,])
  }
}, fill = TRUE)
sink()

inits <- function () {
  list(beta = rnorm(Kc, 0, 0.1),
        gamma = rnorm(Kb, 0, 0.1),
        numS = rnorm(1, 0, 25),
        denomS = rnorm(1, 0, 1))}

params <- c("beta", "gamma", "alpha")
ZANB <- jags(data      = model.data,
              inits     = inits,
              parameters = params,
              model     = "NBH.txt",
              n.thin    = 1,
              n.chains  = 3,
              n.burnin  = 4000,
              n.iter    = 6000)
print(ZANB, intervals=c(0.025, 0.975), digits=3)
=====

mu.vect sd.vect   2.5%   97.5% Rhat n.eff
alpha      0.002  0.001   0.001   0.005 1.001  6000
beta[1]    0.704  0.037   0.626   0.775 1.008  260
beta[2]    1.524  0.019   1.488   1.563 1.009  260
gamma[1]   -1.829  0.199  -2.226  -1.451 1.002 1400
gamma[2]    4.277  0.335   3.665   4.956 1.003  990
deviance  15003236.535 3.417 15003231.948 15003245.359 1.000      1

pD = 5.8 and DIC = 15003242.4

```

...in Python using Stan

In what follows we refrain from repeating the import statements and definition of the `ztp` function, since they were already shown in Code 7.8.

Code 7.10 Zero-altered negative binomial (ZANB) or NB hurdle model in Python using Stan.

```
=====
# Data
```

```

np.random.seed(141)                      # set seed to replicate example
nobs= 750                                  # number of obs in model

x1 = uniform.rvs(size=nobs, loc=-0.5, scale=3.0)

xb = 0.75 + 1.5 * x1                      # linear predictor, xb
exb = np.exp(xb)
poy = ztp(nobs, exb)

xc = -2.0 + 4.5 * x1                      # construct filter
pi = 1.0/(1.0 + np.exp(xc))
bern = [bernoulli.rvs(1-pi[i]) for i in range(nobs)]

poy = [poy[i]*bern[i] for i in range(nobs)]    # add structural zeros

X = np.transpose(x1)
X = sm.add_constant(X)

# Prepare data for Stan
mydata = {}                                # build data dictionary
mydata['Y'] = poy                          # response variable
mydata['N'] = nobs                         # sample size
mydata['Xb'] = X                           # predictors
mydata['Xc'] = X                           # predictors
mydata['Kb'] = X.shape[1]                  # number of coefficients
mydata['Kc'] = X.shape[1]

stan_code = """
data{
    int<lower=0> N;
    int<lower=0> Kb;
    int<lower=0> Kc;
    matrix[N, Kb] Xb;
    matrix[N, Kc] Xc;
    int<lower=0> Y[N];
}
parameters{
    vector[Kc] beta;
    vector[Kb] gamma;
    real<lower=0, upper=5.0> alpha;
}
transformed parameters{
    vector[N] mu;
    vector[N] Pi;
    vector[N] temp;
    vector[N] u;
    mu = exp(Xc * beta);
    temp = Xb * gamma;
    for (i in 1:N) {
        Pi[i] = inv_logit(temp[i]);
        u[i] = 1.0/(1.0 + alpha * mu[i]);
    }
}
model{
    vector[N] LogTrunNB;
    vector[N] z;
    vector[N] l1;
    vector[N] l2;
    vector[N] ll;

    for (i in 1:Kc){
        beta[i] ~ normal(0, 100);
        gamma[i] ~ normal(0, 100);
    }
    for (i in 1:N) {
        LogTrunNB[i] = (1.0/alpha) * log(u[i]) + Y[i] * log(1 - u[i]) +
                        lgamma(Y[i] + 1.0/alpha) - lgamma(1.0/alpha) -
                        lgamma(Y[i] + 1) - log(1 - pow(u[i],1.0/alpha));
        z[i] = step(Y[i] - 0.0001);
        l1[i] = (1 - z[i]) * log(1 - Pi[i]);
        l2[i] = z[i] * (log(Pi[i]) + LogTrunNB[i]);
        ll[i] = l1[i] + l2[i];
    }
    target += ll;
}

```

```

}

"""

# Run mcmc
fit = pystan.stan(model_code=stan_code, data=mydata, iter=6000, chains=3,
                    warmup=4000, n_jobs=3)

# Output
nlines = 10                                # number of lines in screen output

output = str(fit).split('\n')
for item in output[:nlines]:
    print(item)

=====

```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
beta[0]	0.82	6.9e-4	0.04	0.75	0.8	0.82	0.84	0.89	2697.0	1.0
beta[1]	1.47	3.5e-4	0.02	1.43	1.46	1.47	1.48	1.51	2664.0	1.0
gamma[0]	-1.74	3.4e-3	0.19	-2.13	-1.87	-1.74	-1.61	-1.38	3246.0	1.0
gamma[1]	4.52	6.1e-3	0.36	3.86	4.27	4.51	4.74	5.24	3374.0	1.0
alpha	1.5e-3	1.8e-5	1.2e-3	5.9e-5	5.4e-4	1.2e-3	2.2e-3	4.7e-3	5006.0	1.0

7.2.3 Bayesian Gamma–Logit Hurdle Model

When a model is being considered where the response term is continuous and the values are zero and positive real, it may wise to use a gamma–logit model or, more specifically, a log-gamma–logit hurdle model, for modeling the data. Recall that gamma models do not use observations with zero-valued responses when estimating parameters or, for Bayesian models, when sampling for posterior parameters. However, the data is frequently in the form of zeros and positive real numbers. Negative values are assumed to be excluded. A normal or Gaussian model is best used on data having the possibility of both negative and positive values.

Recall from the earlier section on gamma models that the canonical or natural gamma model has a default reciprocal link. That is, the inverse or reciprocal link inverts the study data when calculating predicted or fitted values. There are times when this is exactly what one wants, but we suspect that usually researchers want to maintain the direction of the data being modeled. The solution given in Section 5.22 on gamma models was to use a log-gamma model, i.e., a gamma model with a log link function. This ensures that the data will remain positive and will not be inverted.

The log-gamma hurdle model has two components, as do other two-part hurdle models. The log-gamma component consists of a standard log-gamma model since there is no need to adjust the underlying PDF for missing zero values. The zeros are modeled using a binary component, for example, a logit, probit, complementary loglog, or other binary response model. A logit component is usually preferred, owing to the manner in which parameter values are interpreted. The binary component is constructed by regarding all values greater than zero as 1, with the zeros remaining as they are.

Note that a log link is used for almost all count models. It is the standard default link function with Poisson, negative binomial, generalized Poisson, and other count models. The manner in which posterior means can be interpreted for log-gamma models is similar to how Poisson–logit hurdle models are interpreted.

In the code below the data are separated into x_c and x_b components: x_b is the binary component

and x_c is now the continuous component, not the count component as for the Poisson hurdle model. Note that the gamma posterior means are for the binary component whereas the betas are reserved for the main distribution of interest, in this case the log-gamma estimates.

In the code below a synthetic model with predictors x_1 and x_2 is assumed. The response term is gy . These are stored in an R data frame called `gdata`. It is easy to amend the code for use with real data.

...in R using JAGS

```
set.seed(33559)
# Sample size
nobs <- 750

# Generate predictors, design matrix
x1 <- runif(nobs,0,4)
xc <- -1 + 0.75*x1
exc <- exp(xc)
phi <- 0.066
r <- 1/phi
y <- rgamma(nobs,shape=r, rate=r/exc)
LG <- data.frame(y, x1)

# Construct filter
xb <- -2 + 1.5*x1
pi <- 1/(1 + exp(-(xb)))
bern <- rbinom(nobs,size=1, prob=pi)

# Add structural zeros
LG$y <- LG$y*bern

Code 7.11 Bayesian log-gamma–logit hurdle model in R using JAGS.
=====
Xc <- model.matrix(~ 1 + x1, data=LG)
Xb <- model.matrix(~ 1 + x1, data=LG)
Kc <- ncol(Xc)
Kb <- ncol(Xb)

model.data <- list(
  Y = LG$y,          # response
  Xc = Xc,           # covariates from gamma component
  Xb = Xb,           # covariates from binary component
  Kc = Kc,           # number of betas
  Kb = Kb,           # number of gammas
  N = nrow(LG),      # sample size
  Zeros = rep(0, nrow(LG)))

load.module('glm')
sink("ZAGGLM.txt")
cat("
model{
  # Priors for both beta and gamma components
  for (i in 1:Kc) {beta[i] ~ dnorm(0, 0.0001)}
  for (i in 1:Kb) {gamma[i] ~ dnorm(0, 0.0001)}
  # Prior for scale parameter, r
  r ~ dgamma(1e-2, 1e-2)
  # Likelihood using the zero trick
  C <- 10000
  for (i in 1:N) {
    Zeros[i] ~ dpois(-ll[i] + C)

  # gamma log-likelihood
  lg1[i] <- - loggam(r) + r * log(r / mu[i])
  lg2[i] <- (r - 1) * log(Y[i]) - (Y[i] * r) / mu[i]
  LG[i] <- lg1[i] + lg2[i]

  z[i] <- step(Y[i] - 0.0001)
  l1[i] <- (1 - z[i]) * log(1 - Pi[i])
  l2[i] <- z[i] * ( log(Pi[i]) + LG[i])
  ll[i] <- l1[i] + l2[i]

  log(mu[i]) <- inprod(beta[], Xc[i,])
  logit(Pi[i]) <- inprod(gamma[], Xb[i,])
}
phi <- 1/r
}", fill = TRUE)
sink()

# Initial parameter values
inits <- function () {
  list(beta = rnorm(Kc, 0, 0.1),
       gamma = rnorm(Kb, 0, 0.1),
       r      = runif(1, 0,100) )}
```

```

# Parameter values to be displayed in output
params <- c("beta", "gamma", "phi")

# MCMC sampling
ZAG <- jags(data      = model.data,
             inits     = inits,
             parameters = params,
             model     = "ZAGGLM.txt",
             n.thin    = 1,
             n.chains  = 3,
             n.burnin  = 2500,
             n.iter    = 5000)

# Model results
print(ZAG, intervals = c(0.025, 0.975), digits=3)
=====
          mu.vect   sd.vect      2.5%    97.5%     Rhat  n.eff
beta[1]    -0.991    0.024    -1.035    -0.939    1.069    35
beta[2]     0.739    0.009     0.720     0.756    1.074    33
gamma[1]    -1.958    0.172    -2.295    -1.640    1.007   330
gamma[2]     1.462    0.096     1.282     1.652    1.008   290
phi        0.068    0.003     0.062     0.073    1.003   890
deviance 20001890.234   3.184 20001886.012 20001897.815   1.000    1

pD = 5.0 and DIC = 20001895.3

```

...in Python using Stan

Code 7.12 Bayesian log-gamma–logit hurdle model in Python using Stan.

```

=====
import numpy as np
import pystan
import statsmodels.api as sm

from scipy.stats import uniform, gamma, bernoulli

# Data
np.random.seed(33559)                      # set seed to replicate example
nobs = 1000                                    # number of obs in model

# Generate predictors, design matrix
x1 = uniform.rvs(loc=0, scale=4, size=nobs)
xc = -1 + 0.75 * x1
exc = np.exp(xc)
phi = 0.066
r = 1.0/phi
y = np.random.gamma(shape=exc, scale=r)

# Construct filter
xb = -2 + 1.5 * x1
pi = 1 / (1 + np.exp(-xb))
bern = bernoulli.rvs(1-pi)

gy = [y[i]*bern[i] for i in range(nobs)] # add structural zeros

X = np.transpose(x1)
X = sm.add_constant(X)

# Fit
mydata = {}                                     # build data dictionary
mydata['Y'] = gy                                # response variable
mydata['N'] = nobs                               # sample size
mydata['Xb'] = X                                 # predictors
mydata['Xc'] = x1
mydata['Kb'] = X.shape[1]                        # number of coefficients
mydata['Kc'] = X.shape[1]

stan_code = """
data{
int N;
int Kb;
int Kc;
matrix[N, Kb] Xb;
matrix[N, Kc] Xc;
real<lower=0> Y[N];
}
parameters{
```

```

vector[Kc] beta;
vector[Kb] gamma;
real<lower=0> phi;
}
model{
  vector[N] mu;
  vector[N] Pi;

  mu = exp(Xc * beta);
  for (i in 1:N) Pi[i] = inv_logit(Xb[i] * gamma);

  for (i in 1:N) {
    (Y[i] == 0) ~ bernoulli(Pi[i]);
    if (Y[i] > 0) Y[i] ~ gamma(mu[i], phi) T[0,];
  }
}
"""
# Run mcmc
fit = pystan.stan(model_code=stan_code, data=mydata, iter=6000, chains=3,
                   warmup=4000, n_jobs=3)

# Output
print (fit)
=====
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
beta[0]	-0.97	1.9e-3	0.08	-1.12	-1.02	-0.97	-0.92	-0.82	1536.0	1.0
beta[1]	0.79	1.e-3	0.04	0.71	0.77	0.8	0.82	0.88	1527.0	1.0
gamma[0]	-1.9	4.6e-3	0.17	-2.24	-2.02	-1.9	-1.79	-1.57	1374.0	1.0
gamma[1]	1.5	2.6e-3	0.1	1.31	1.43	1.5	1.56	1.69	1381.0	1.0
phi	0.07	1.7e-4	6.4e-3	0.06	0.07	0.07	0.08	0.08	1459.0	1.0

7.2.4 Bayesian Lognormal–Logit Hurdle Model

In what follows we show how straightforward it is to change from a gamma to a lognormal model. Both models are useful for modeling the continuous positive component of the hurdle model, and the choice of one or the other depends on the desired interpretation from the researcher, as discussed in previous chapters. Figure 7.6 displays an example of lognormal–logit hurdle-distributed data.

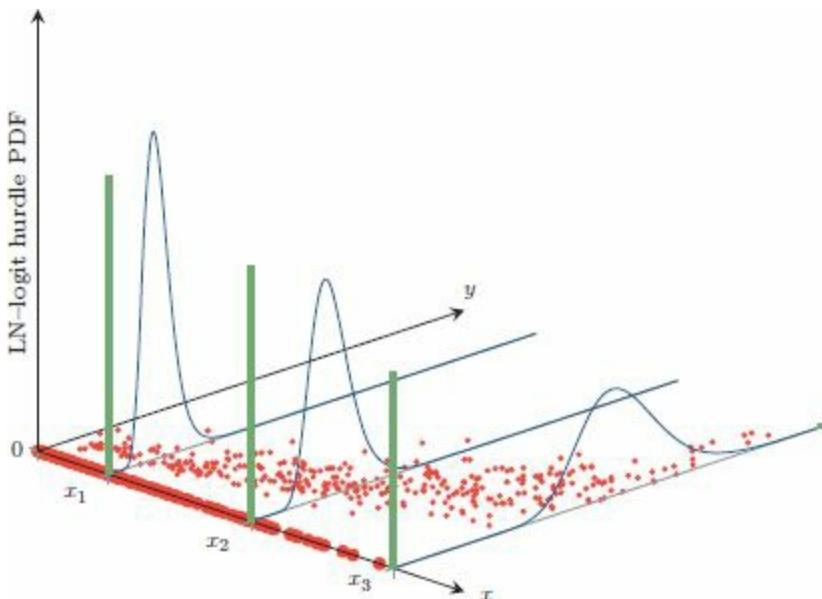


Figure 7.6 Illustration of lognormal–logit hurdle-distributed data.

...in R using JAGS

Code 7.13 Bayesian lognormal–logit hurdle using JAGS.

```
=====
set.seed(33559)
# Sample size
nobs <- 1000

# Generate predictors, design matrix
x1 <- runif(nobs,0,2.5)
xc <- 0.6 + 1.25*x1
y <- rlnorm(nobs, xc, sdlog=0.4)
lndata <- data.frame(y, x1)

# Construct filter
xb <- -3 + 4.5*x1
pi <- 1/(1+exp(-(xb)))
bern <- rbinom(nobs,size=1, prob=pi)

# Add structural zeros
lndatas$y <- lndatas$y*bern

Xc <- model.matrix(~ 1 + x1,data = lndata)
Xb <- model.matrix(~ 1 + x1,data = lndata)
Kc <- ncol(Xc)
Kb <- ncol(Xb)

JAGS.data <- list(
  Y = lndatas$y, # response
  Xc = Xc, # covariates
  Xb = Xb, # covariates
  Kc = Kc, # number of betas
  Kb = Kb, # number of gammas
  N = nrow(lndata), # sample size
  Zeros = rep(0, nrow(lndata)))

load.module('glm')
sink("ZALN.txt")
cat("
model{
  # Priors for both beta and gamma components
  for (i in 1:Kc) {beta[i] ~ dnorm(0, 0.0001)}
  for (i in 1:Kb) {gamma[i] ~ dnorm(0, 0.0001)}

  # Prior for sigma
  sigmaLN ~ dgamma(1e-3, 1e-3)

  # Likelihood using the zero trick
  C <- 10000
  for (i in 1:N) {
    Zeros[i] ~ dpois(-ll[i] + C)

    # LN log-likelihood
    ln1[i] <- -(log(Y[i]) + log(sigmaLN) + log(sqrt(2 * sigmaLN)))
    ln2[i] <- -0.5 * pow((log(Y[i]) - mu[i]),2)/(sigmaLN * sigmaLN)
    LN[i] <- ln1[i] + ln2[i]
    z[i] <- step(Y[i] - 1e-5)
    l1[i] <- (1 - z[i]) * log(1 - Pi[i])
    l2[i] <- z[i] * ( log(Pi[i]) + LN[i])
    ll[i] <- l1[i] + l2[i]

    mu[i] <- inprod(beta[], xc[i,])
    logit(Pi[i]) <- inprod(gamma[], Xb[i,])
  }
}", fill = TRUE)
sink()

# Initial parameter values
inits <- function () {
  list(beta = rnorm(Kc, 0, 0.1),
       gamma = rnorm(Kb, 0, 0.1),
       sigmaLN = runif(1, 0, 10))}

# Parameter values to be displayed in output
params <- c("beta", "gamma", "sigmaLN")

# MCMC sampling
ZALN <- jags(data      = JAGS.data,
              inits     = inits,
```

```

parameters = params,
model      = "ZALN.txt",
n.thin     = 1,
n.chains   = 3,
n.burnin   = 2500,
n.iter     = 5000)

# Model results
print(ZALN, intervals = c(0.025, 0.975), digits=3)
=====
mu.vect sd.vect    2.5%    97.5% Rhat n.eff
beta[1]  0.703  0.041    0.621    0.782 1.003 1700
beta[2]  1.238  0.025    1.190    1.287 1.002 1800
gamma[1] -3.063  0.235   -3.534   -2.615 1.008 300
gamma[2]  4.626  0.299    4.054    5.225 1.007 320
deviance 20004905.556  3.168 20004901.358 20004913.309 1.000      1

pD = 5.0 and DIC = 20004910.6

```

The values of the posteriors for both the count and the Bernoulli components are close to what we specified in the synthetic data. The values are nearly identical to the maximum likelihood estimation of the same data.

...in Python using Stan

The differences in parameterization of the Bernoulli parameter in generating the data and in fitting the model are due to the underlying parameterizations used by Stan and `scipy.stats`.

Code 7.14 Bayesian lognormal–logit hurdle model in Python using Stan.

```

=====
import numpy as np
import pystan
import statsmodels.api as sm

from scipy.stats import uniform, bernoulli

# Data
np.random.seed(33559)                      # set seed to replicate example
nobs = 2000                                    # number of obs in model

x1 = uniform.rvs(loc=0, scale=2.5, size=nobs)
xc = 0.6 + 1.25 * x1                         # linear predictor
y = np.random.lognormal(sigma=0.4, mean=np.exp(xc))

xb = -3.0 + 4.5 * x1                         # construct filter
pi = 1.0/(1.0 + np.exp(-xb))
bern = [bernoulli.rvs(1-pi[i]) for i in range(nobs)]

ly = [y[i]*bern[i] for i in range(nobs)] # add structural zeros

X = np.transpose(x1)
X = sm.add_constant(X)

mydata = {}                                     # build data dictionary
mydata['Y'] = ly                               # response variable
mydata['N'] = nobs                            # sample size
mydata['Xb'] = X                               # predictors
mydata['Xc'] = x1                             # number of coefficients
mydata['Kb'] = X.shape[1]
mydata['Kc'] = X.shape[0]

# Fit
stan_code = """
data{
  int<lower=0> N;
  int<lower=0> Kb;
  int<lower=0> Kc;
  matrix[N, Kb] Xb;
  matrix[N, Kc] Xc;
}
```

```

    real<lower=0> Y[N];
}
parameters{
  vector[Kc] beta;
  vector[Kb] gamma;
  real<lower=0> sigmaLN;
}
model{
  vector[N] mu;
  vector[N] Pi;

  mu = exp(Xc * beta);
  for (i in 1:N) Pi[i] = inv_logit(Xb[i] * gamma);

  for (i in 1:N) {
    (Y[i] == 0) ~ bernoulli(Pi[i]);
    if (Y[i] > 0) Y[i] ~ lognormal(mu[i], sigmaLN);
  }
}
"""

# Run mcmc
fit = pyStan.stan(model_code=stan_code, data=mydata, iter=7000, chains=3,
                    warmup=4000, n_jobs=3)

# Output
print(fit)
=====

      mean   se_mean     sd   2.5%   25%   50%   75%   97.5%   n_eff   Rhat
beta[0]  0.59   2.0e-4  9.2e-3  0.57  0.58  0.59   0.6   0.61  2036.0   1.0
beta[1]  1.26   2.1e-4  9.3e-3  1.24  1.25  1.26   1.26  1.28  2045.0   1.0
gamma[0] -3.09   4.1e-3   0.18  -3.44  -3.2  -3.09  -2.97  -2.74  1903.0   1.0
gamma[1]  4.55   5.1e-3   0.22   4.12   4.4   4.54   4.7   5.0   1905.0   1.0
sigmaLN  0.42   2.6e-4   0.01   0.39   0.41   0.42   0.43   0.44  2237.0   1.0

```

Further Reading

- Cameron, E. (2011). "On the estimation of confidence intervals for binomial population proportions in astronomy: the simplicity and superiority of the Bayesian approach." *Publ. Astronom. Soc. Australia* 28, 128–139. DOI: [10.1071/AS10046](https://doi.org/10.1071/AS10046). arXiv:1012.0566 [astro-ph.IM].
- de Souza, R. S., E. Cameron, M. Killedar, J. M. Hilbe, R. Vilalta, U. Maio, V. Biffi *et al.* (2015). "The overlooked potential of generalized linear models in astronomy, I: Binomial regression." *Astron. Comput.* 12, 21–32. DOI: <http://dx.doi.org/10.1016/j.ascom.2015.04.002>.
- Elliott, J., R. S. de Souza, A. Krone-Martins, E. Cameron, E. O. Ishida, and J. M. Hilbe (2015). "The overlooked potential of generalized linear models in astronomy, II: gamma regression and photometric redshifts." *Astron. Comput.* 10, 61–72. DOI: [10.1016/j.ascom.2015.01.002](https://doi.org/10.1016/j.ascom.2015.01.002). arXiv: 1409.7699 [astro-ph.IM].
- Hardin, J. W. and J. M. Hilbe (2012). *Generalized Linear Models and Extensions, Third Edition*. Taylor & Francis.
- Hilbe, J. M. (2011). *Negative Binomial Regression, Second Edition*. Cambridge University Press.
- Hilbe, J. M. (2014). *Modeling Count Data*. Cambridge University Press.
- Hilbe, J. M. (2015). *Practical Guide to Logistic Regression*. Taylor & Francis.
- McElreath, R. (2016). *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*. Chapman & Hall/CRC Texts in Statistical Science. CRC Press.
- Smithson, M. and E. C. Merkle (2013). *Generalized Linear Models for Categorical and Continuous Limited Dependent Variables*. Chapman & Hall/CRC Statistics in the Social and Behavioral Sciences. Taylor & Francis.
- Zuur, A. F., J. M. Hilbe, and E. N. Ieno (2013). *A Beginner's Guide to GLM and GLMM with R: A Frequentist and Bayesian Perspective for Ecologists*. Highland Statistics.

8 Hierarchical GLMMs

8.1 Overview of Bayesian Hierarchical Models/GLMMs

The Bayesian models we have discussed thus far in the book have been based on a likelihood, and the mixture of a likelihood and prior distribution. The product of a model likelihood, or log-likelihood, and a prior is called a posterior distribution. One of the key assumptions of a likelihood distribution is that each component observation in the distribution is independent of the other observations. This assumption goes back to the probability distribution from which a likelihood is derived. The terms or observations described by a probability distribution are assumed to be independent. This criterion is essential to creating and interpreting the statistical models we addressed in the last chapter.

When there is correlation or time series autocorrelation in the data caused by clustered, nested, or panel structured data, statisticians must make adjustments to the model in order to avoid bias in the interpretation of the parameters, especially the standard deviations. In maximum likelihood estimation this is a foremost problem when developing models, and an entire area of statistics is devoted to dealing with excess correlation in data.

In many cases the data being modeled is correlated by its structure, or is said to be structurally correlated. For instance, suppose we have data that is collected by individual observations over time or data that belongs to different sources or clusters, e.g., data in which the observations are nested into levels and so is hierarchically structured. Example data is provided in the tables. Table 8.1 gives an example of longitudinal data. Note that time periods (`Period`) are nested within each observation (`id`). Table 8.2 gives an example of hierarchical, grouped, or clustered data. This type of data is also referred to as cross sectional data. It is the data structure most often used for random intercept data. We pool the data if the grouping variable, `grp`, is ignored. However, since it may be the case that values within groups are more highly correlated than are the observations when the data is pooled, it may be necessary to adjust the model for the grouping effect. If there is more correlation within groups, the data is likely to be overdispersed. A random intercept model adjusts the correlation effect by having separate intercepts for each group in the data. We will describe how this works in what follows.

Table 8.1 Example of longitudinal data.

i

j

id	Period	X	Y
1	1	9	45
1	2	10	39
1	3	13	40
1	4	15	43
2	1	7	29
2	2	11	34
2	3	15	44
2	4	19	48
3	1	8	30
3	2	10	38
3	3	12	44
3	4	17	49
4	1	15	31
4	2	17	39
4	3	18	47
4	4	20	49
5	1	14	29
:	:	:	:
:	:	:	:

Table 8.2 Example of clustered data.

grp	i	X	Y
1	1	23	1
1	1	12	1
1	1	18	0
1	1	20	1
1	1	16	0
1	1	14	1
2	2	17	1
2	2	15	0
2	2	10	0
2	2	10	1
3	3	11	0
3	3	17	0
3	3	15	1
4	4	12	0
4	4	13	0
4	4	23	1
5	5	20	0
:	:	:	:
:	:	:	:

Let us first give an example from an imagined relationship between a student's grade point average (GPA) and test score. It may help to clarify the basic relationships involved with modeling this type of data structure by using a random intercept or random-intercept-random-slopes (abbreviated to random intercept-slopes) model. Be aware, though, that this explanation is foremost

based on frequency-based modeling. Models of this class are alternatively called random effect models, panel models, hierarchical models, or multilevel models. When Bayesian methods are used, these models are usually referred to as hierarchical or generalized linear mixed models (GLMM).

So, we suppose that a study is made of student scores on the Graduate Record Examination (GRE) subject-area test in physics. The scores are taken from applicants for PhD programs in astronomy and astrophysics. We wish to model the test scores on the basis of applicants' cumulative GPAs on undergraduate courses taken in mathematics and physics. The model can appear as

$$Score = \beta_0 + (\beta_1 \times GPA) + \varepsilon, \quad (8.1)$$

where β_0 and β_1 are parameters and ε is an error term (see below). What may concern us, though, is that the GPA results could differ depending on where the applicant did their undergraduate work. Students taking courses at one university may differ from students at another university. Moreover, the type of courses and expectations of what must be learned to achieve a given grade may differ from university to university. In other words, GPA results within a university may be more correlated than results between universities.

There are a number of ways in which statisticians have adjusted for this panel effect. The most basic is referred to as a random intercept model. Recall the error term ε in Equation 8.1. It represents the error that exists between the true GRE physics scores obtained and the score values predicted on the basis of the model. Since the GPA results between students are assumed to be independent of one another, the errors should be normally distributed as $N(0, \sigma^2)$.

The random intercept model conceptually divides the data into separate models for each university that is part of the study. The coefficients are assumed to remain the same, but the intercepts vary between universities. If the mean GPA results within each university are nearly the same as the overall mean GPA results, then the universities will differ little in their results. If they differ considerably, however, we know that there is a university effect.

It is important to have a sufficient number of groups, or universities, in the model to be able to have a meaningful variance statistic to distinguish between groups; this is standard statistical practice for this class of models. For frequency-based models most simulation studies require at least 10 groups with a minimum of 20 observations in each group. The advantage of using a Bayesian random intercept hierarchical model, though, is that fewer groups may be sufficient for a well-fitted model (see the astronomical examples in Sections 10.2 and 10.3). It depends on the fit of each parameter in the model.

The general formula for a random intercept model for a single predictor can be given as

$$y_{ij} = \beta_0 + \beta_1 X_{ij} + \xi_j + \varepsilon_{ij}. \quad (8.2)$$

Here β_0 represents the overall mean of y across all groups; $\beta_1 X_{ij}$ represents the vector of

coefficients in the model acting on the matrix of the model data. For frequentist-based models, the coefficients are considered as fixed. In Bayesian methodology they are random; each predictor parameter is estimated using MCMC sampling. The quantities ξ_j are the random intercepts, which are the same within each group j in the model. Finally, as mentioned earlier, ε_{ij} are the residual errors, indicating the difference between the predicted and actual model values for each observation in the model. The fit is evaluated by comparing the mean value of each intercept with the pooled mean. The predicted values for each intercept can be obtained in the usual way. That is, there are as many mean predicted values as intercepts. The predictor values differ only by the differences in the transformed intercept values. The inverse link function of the model transforms the various intercepts to the fitted values. Of course, the parameters and data common to all groups are added to the calculations.

Random or varying intercept models are also known as partially pooled models. The reason for this is that the model provides a more accurate estimate of the component group or cluster means than if the data were divided into the component models and each mean were estimated separately, or if the data were pooled across groups with the clustering effect ignored.

A second model, called a random coefficients or random slopes model, allows the coefficients to vary across groups. For our example, the coefficient of GPA can vary across schools. Usually, though, if we wish the coefficients to vary then we vary them in conjunction with the intercepts. Only rarely does a researcher attempt to design a model with random slopes only. We therefore will consider a model with both random slopes, or coefficients, and intercepts.

The general formula for the combined random intercept and slopes model can be expressed as

$$y_{ij} = \beta_0 + \beta_1 X_{ij} + \xi_{0j} + \xi_{1j} X_{ij} + \varepsilon_{ij}, \quad (8.3)$$

where an extra term, $\xi_{1j} X_{ij}$, has been added to the random intercept model (Equation 8.2). This term represents the random coefficient, providing us with the difference between each group's slope and $\beta_1 X_{ij}$.

At times statisticians construct the equation for the random slopes (and intercept) model in such a way that the fixed and random components of the model are combined:

$$y_{ij} = \beta_0 + \xi_{0j} + \beta_1 X_{1ij} + \xi_{1j} X_{ij} + \varepsilon_{ij}; \quad (8.4)$$

here the first two terms on the right-hand side of the equation are the random intercepts and the second two terms are the random slope components.

Again, there are a large variety of random effect models. In astronomy it is likely that the grouping variables, which represent a second level when the model is thought of as hierarchical, take the form of clusters. A second-level group can also be longitudinal. However, in longitudinal models the time periods are regarded as the level-1 variables and the individuals as the level-2

variables. Thus one may track changes in luminosity for a class of stars over time. The individual stars, j , are the grouping variables; the changes in measurement, i , within individual stars are the level-1 observations. Care must be taken when assigning level-1 and level-2 status with respect to cross sectional nested or clustered data and longitudinal data.

We should note as well that the symbolization of this class of models varies with individual authors. There is no truly standard symbolization, although the above formulas are commonly used. This is particularly the case for Bayesian hierarchical models, which incorporate models varying by intercept as well as models varying by both parameter mean and intercept. Moreover, when we focus on Bayesian models, the coefficients are no longer slopes; they are posterior distribution means. They are also already random, and therefore differ from frequentist-based models where the coefficients are assumed to be fixed. We shall observe how these models are constructed for Gaussian, logistic, Poisson, and negative binomial hierarchical models. For the Poisson distribution we shall provide examples for both a random intercept model and a combined random-intercept–random-slopes (or coefficients) model. The logic used for the Poisson random intercept–slopes model will be the same as for other distributions.

The reader can compare the code for creating a basic Bayesian model with the code we show for both random intercept and random intercept–slope models. The same logic that we use to develop random intercept models, for instance, from the code for basic Bayesian models can be applied to random intercept or random intercept–slope models that we do not address in this chapter. This is the case for beta and beta–binomial random intercept models as well as zero-inflated and three-parameter NB-P models.

8.2 Bayesian Gaussian or Normal GLMMs

8.2.1 Random Intercept Gaussian Data

The random intercept Gaussian or normal model is the most basic generalized linear mixed model, which, as we have observed, is also referred to as a hierarchical or multilevel model. The model should be treated as a standard normal model but with separate intercepts for each cluster or group. The goal is to determine whether there is more correlation or within-group variance in the data than there is between-group variance in the data. When the groups or clusters are not taken into consideration for modeling purposes, we refer to the data used for the model as pooled. In other words, we model the data using a standard Bayesian model. In using a random intercepts model, we obtain a mean value for the correlation ϕ of the within-groups data using the formula

$$\phi = \frac{\sigma_{\text{groups}}^2}{\sigma_{\text{groups}}^2 + \sigma_{\text{pooled}}^2}. \quad (8.5)$$

As for the example models in Chapter 5, we create random intercept or random-intercept–random-slopes data for each model discussed in this section. Code 8.1 creates 4500 observations divided equally into 20 groups with 225 observations in each. Again, it is important to have a sufficient

number of groups in the data in order to develop a meaningful within-group variance as well as a between-group variance. In frequentist models the recommendation is for a minimum of 10 groups with some 20 observations in each group. Slightly smaller numbers for both can still produce meaningful variance statistics, but we advise not reducing the numbers too much. It is also important to remember that the data is assumed to be a sample from a greater population of data, i.e., the model data is not itself the population data.

For the Bayesian models we are discussing here, recommendations based on simulation studies state that there should be at least five groups with 20 or more observations in each. Bayesian models, like frequentist models, still use random intercept models to address problems with overdispersion caused by the clustering of data into groups or panels.

In Code 8.1, observe that the line that creates the number of groups and the number of observations in each starts with `Groups`. The following line puts the normally distributed values in each group, with mean zero and standard deviation 0.5. The random effect is therefore signified as the variable `a`. This code is common for all the synthetic random intercept models we develop in this chapter. Other distributions could be used for the random intercepts, but generally a good reason must be given to do so. The line beginning with `y` specifies the family distribution of the model being created. Here it is a normal or Gaussian model – hence the use of the pseudo-random number generator `rnorm`.

Code 8.1 Random intercept Gaussian data generated in R.

```
=====
set.seed(1656)
N <- 4500
NGroups <- 20
x1 <- runif(N)
x2 <- runif(N)

Groups <- rep(1:20, each = 225)      # 20 groups, each with 225 observations
a <- rnorm(NGroups, mean = 0, sd = 0.5)
print(a,2)
[1]  0.579 -0.115 -0.125  0.169 -0.500  -1.429 -1.171 -0.205  0.193
    0.041 -0.917 -0.353 -1.197  1.044  1.084 -0.085 -0.886 -0.352
   -1.398  0.350

mu <- 1 + 0.2 * x1 - 0.75 * x2 + a[Groups]
y <- rnorm(N, mean=mu, sd=2)

normr <- data.frame(
  y = y,
  x1 = x1,
  x2 = x2,
  Groups = Groups,
  RE = a[Groups]
)
=====
```

8.2.2 Bayesian Random Intercept Gaussian Model in R using JAGS

The `MCMCglmm` function of R allows estimation of several important Bayesian random intercept models. The function is part of the `MCMCglmm` package on CRAN. The PDF document which comes with the software describes its options and limitations. We primarily employ the default options with `MCMCglmm`. These are 13 000 samples, or iterations of the MCMC algorithm, with 3000 burn-in samples and a thinning rate of 10. That is, every tenth sample is used to create the posterior distribution of each parameter in the model. Ten thousand samples are actually used to develop the

posterior. These are small numbers for most real-life analyses but are generally fine for synthetic data. However, for this normal model we used the `burnin` option to specify 10 000 burn-in samples and the `nitt` option to have 20 000 samples for the posterior. We could have used `thin` to change the default `thin` rate value, but there is no appearance of a problem with autocorrelation. When there is, changing the `thin` rate can frequently ameliorate the problem.

The most important post-estimation statistics for the `MCMCglmm` function are `plot(model)`, `summary(model)`, and `autocorr(model$vcv)`. We show `summary` results below, but not `plot` or `autocorr` results. The `plot` option provides trace graphs and histograms for each parameter. The mean of the posterior distribution is the default parameter value, but the median or mode may also be used. These are interpreted in the same way as we interpreted the models in Chapter 5 and are vital in assessing the worth of a parameter. For space reasons we do not display them, neither do we display the autocorrelation statistics.

The random intercept normal or Gaussian model can be estimated using the code below. The `verbose=FALSE` option specifies that an iteration log of partial results is not displayed to the screen. If this option is not specified, pages of partial results can appear on your screen.

```
> library(MCMCglmm)
> bnglmm <- MCMCglmm(y ~ x1 + x2, random=~Groups,
+ family="gaussian", data=normr, verbose=FALSE,
+ nitt=20000, burnin=10000)
> summary(bnglmm)
```

The output of the above `MCMCglmm` normal model is:

```
DIC: 19001.4
G-structure: ~Groups
  post.mean l-95% CI u-95% CI eff.samp
Groups    0.6054   0.2396   1.041     1000
R-structure: ~units
  post.mean l-95% CI u-95% CI eff.samp
units     3.976    3.817    4.133     848.5
Location effects: y ~ x1 + x2
  post.mean    l-95% CI    u-95% CI    eff.samp      pMCMC
(Intercept)  0.713572  0.320457  1.062875    1000  <0.001 ***
x1          0.202560 -0.009952  0.386730    1081   0.048 *
x2         -0.691357 -0.877759 -0.481239    1000  <0.001 ***
```

The G-structure results relate to the variance of the random effects or intercepts of the model. The variance of `Groups` has posterior mean value 0.6054 and 95% credible interval 0.2396 to 1.041. This is the estimated mean variance for `Groups` (intercepts). It is a parameter for this class of Bayesian models.

The R structure relates to the variance of the model residuals, which has posterior mean 3.976. The `eff.samp` statistic is the effective sample size, which is a measure of the autocorrelation within the sampling distribution of the parameter. Ideally it should be close to the MCMC sample size or alternatively it should be approximately 1000 or more. The `pMCMC` statistics tests whether the

parameter is significantly different from zero.

Intercept, x_1 , and x_2 are fixed effects with mean posterior statistics 0.7136, 0.2026, and -0.6913 respectively. These are close to the values specified in Code 8.1. The error involved has its source in the random code used to create the data and in the randomness of the sampling to develop the mean parameter.

We next turn to using JAGS from within R to calculate a Bayesian random intercept normal or Gaussian model. The advantage of using JAGS is, of course, that it is not limited to the models built into `MCMCglmm`. For instance, the Bayesian GLMM negative binomial is not an option with `MCMCglmm`, but it is not difficult to code it using JAGS once we know the standard Bayesian negative binomial.

8.2.3 Bayesian Random Intercept Normal Model in R using JAGS

We use the data created in Code 8.1 for the JAGS code in Code 8.2 below. Note that the data is called `normr`.

Code 8.2 Random intercept normal model in R using JAGS.

```
=====
library(R2jags)
# Data
X <- model.matrix(~ x1 + x2, data = normr)
K <- ncol(X)
re <- as.numeric(normr$Groups)
Nre <- length(unique(normr$Groups))

model.data <- list(
  Y = normr$y,           # response
  X = X,                 # covariates
  N = nrow(normr),       # rows in model
  re = re,                # random effect
  b0 = rep(0,K),          # parameter priors with initial 0
                         # value
  B0 = diag(0.0001, K),  # priors for V-C matrix
  a0 = rep(0,Nre),        # priors for scale parameters
  A0 = diag(Nre))         # hyperpriors for scale parameters

# Fit
sink("lmm.txt")
cat("
model {
  # Diffuse normal priors for regression parameters
  beta ~ dmnorm(b0[], B0[,])

  # Priors for random intercept groups
  a ~ dmnorm(a0, tau.plot * A0[,])

  # Priors for the two sigmas and taus
  tau.plot <- 1 / (sigma.plot * sigma.plot)
  tau.eps <- 1 / (sigma.eps * sigma.eps)
  sigma.plot ~ dunif(0.001, 10)
  sigma.eps ~ dunif(0.001, 10)

  # Likelihood
  for (i in 1:N) {
    Y[i] ~ dnorm(mu[i], tau.eps)
    mu[i] <- eta[i]
    eta[i] <- inprod(beta[], X[i,]) + a[re[i]]
  }
}
", fill = TRUE)
sink()

inits <- function () {
  list(beta = rnorm(K, 0, 0.01),
```

```

a = rnorm(Nre, 0, 1),
sigma.eps = runif(1, 0.001, 10),
sigma.plot = runif(1, 0.001, 10)
)}

params <- c("beta", "a", "sigma.plot", "sigma.eps")

NORM0 <- jags(data = model.data,
                 inits = inits,
                 parameters = params,
                 model.file = "lmm.txt",
                 n.thin = 10,
                 n.chains = 3,
                 n.burnin = 6000,
                 n.iter = 10000)

# Output
print(NORM0, intervals=c(0.025, 0.975), digits=3)
=====
Inference for Bugs model at "lmm.txt", fit using jags,
  3 chains, each with 10000 iterations (first 6000 discarded), n.thin = 10
  n.sims = 1200 iterations saved
      mu.vect    sd.vect    2.5%    97.5%     Rhat   n.eff
a[1]      0.855    0.217    0.436    1.266    1.000   1200
a[2]      0.346    0.219   -0.083    0.791    1.001   1100
a[3]      0.028    0.221   -0.401    0.462    1.000   1200
a[4]      0.397    0.220   -0.007    0.836    1.002   890
a[5]     -0.209    0.213   -0.632    0.199    1.002   1200
a[6]     -1.106    0.216   -1.524   -0.682    1.003   620
a[7]     -0.946    0.215   -1.386   -0.536    1.000   1200
a[8]     -0.041    0.221   -0.487    0.398    1.000   1200
a[9]      0.453    0.219    0.017    0.871    1.002   1200
a[10]     0.347    0.215   -0.081    0.768    1.000   1200
a[11]     -0.557    0.210   -0.988   -0.170    1.004   910
a[12]     -0.013    0.217   -0.429    0.415    1.001   1200
a[13]     -1.011    0.216   -1.438   -0.589    1.002   1000
a[14]      1.230    0.221    0.804    1.676    1.000   1200
a[15]      1.268    0.222    0.820    1.731    1.000   1200
a[16]      0.057    0.218   -0.362    0.475    1.000   1200
a[17]     -0.404    0.219   -0.834    0.029    1.000   1200
a[18]     -0.009    0.214   -0.435    0.425    1.002   1200
a[19]     -1.297    0.217   -1.738   -0.887    1.001   1200
a[20]      0.603    0.220    0.165    1.020    1.001   1200
beta[1]     0.720    0.189    0.346    1.095    1.002   1200
beta[2]     0.203    0.102   -0.011    0.403    1.000   1200
beta[3]     -0.698    0.101   -0.893   -0.497    1.005   420
sigma.eps    1.994    0.022    1.950    2.036    1.006   350
sigma.plot    0.800    0.143    0.576    1.136    1.001   1200
deviance  18978.863   6.994 18967.033 18994.194    1.000   1200

```

For each parameter, n.eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor (at convergence, Rhat=1).

DIC info (using the rule, pD = var(deviance)/2)
pD = 24.5 and DIC = 19003.3

There are 20 “a” statistics, one for each intercept in the model. An analyst should look for consistency of the slopes. Also note that the estimated beta values, which are the means of the posterior distributions for the model intercept and predictors, are consistent with what we estimated using the MCMCglmm software. Figure 8.1 illustrates this, showing the posteriors for each “a” (the vertical lines) and their fiducial values (the crosses).

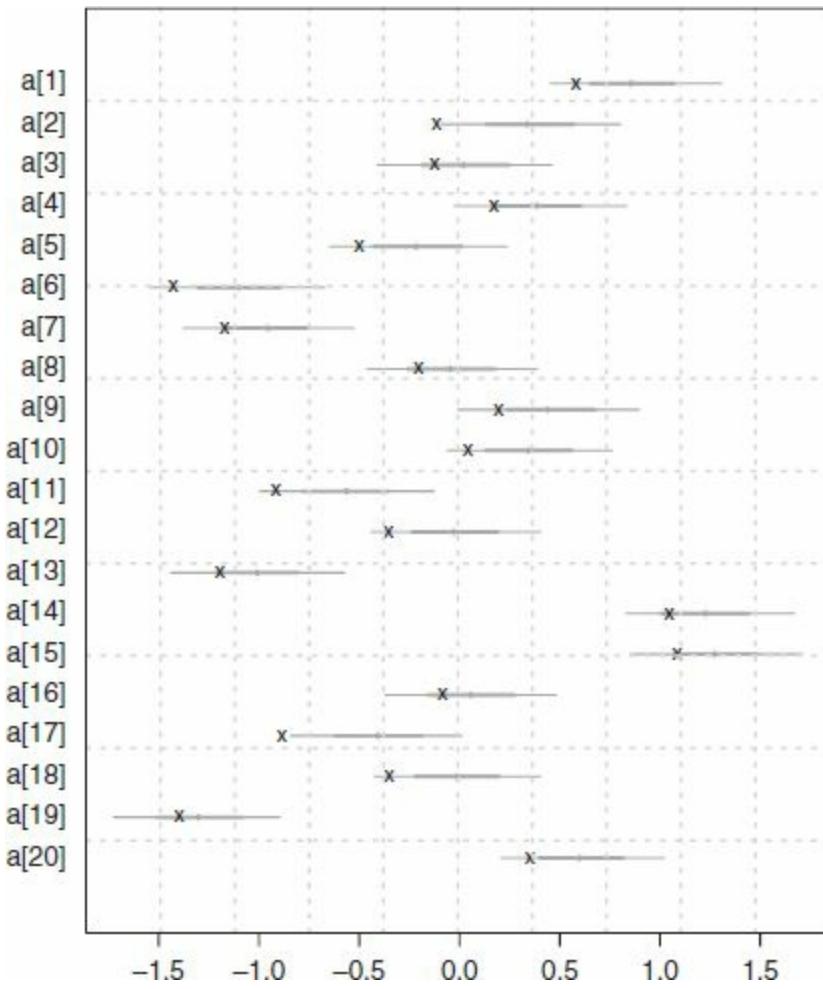


Figure 8.1 Results from the random intercept model. The horizontal lines represent the posterior distributions for each intercept a and the crosses represent their fiducial value.

The DIC statistic is comparative, like the AIC or BIC statistic in maximum likelihood estimation. The lower the DIC between two models on the same data, the better the fit. There are a number of post-estimation fit tests that can be employed to evaluate the model. We do not address these tests here. However, `n.eff` and `Rhat` statistics are provided in the displayed results of almost all Bayesian software. We should address them at this point. The effective sample size is `n.eff`, the same statistic as the `MCMCglmm eff.samp` statistic. When the `n.eff` statistic is considerably lower than the number of samples used to form the posterior distribution following burn-in, this is evidence that the chains used in estimation are inefficient. This does not mean that the statistics based on the posterior are mistaken; it simply means that there may be more efficient ways to arrive at acceptable results. For most models, a better-fitted parameter should have an `n.eff` of 1000 or more.

`Rhat` is the Gelman–Rubin convergence statistic, which is also symbolized simply as `R`. Values of the statistic above 1.0 indicate that there is a problem with convergence, in particular that a chain has not yet converged. Usually the solution is to raise the number of samples used to determine the posterior of the parameter in question. However, this solution should not be held as an absolute: values above 1.0 may be a genuine cause for concern, and posterior statistics should be only tentatively accepted, especially if other `Rhat` statistics in the model are also over 1.0.

Note in the JAGS code the `tau.plot` and `tau.eps` functions are defined in terms of the inverse of `sigma.plot` squared and the inverse of `sigma.eps` squared, where `sigma.plot` is the standard deviation of the mean intercepts and its square is the variance. Likewise, `sigma.eps` is the standard deviation of the pooled mean intercepts, and its square is the variance. The functions `tau.plot` and `tau.eps` are the precision statistics for each parameter mean.

There may be times when an analyst simply wants to create data with a specified number of groups or clusters, without considering any random effects, or he or she may want to use an entirely different type of model on the data. The example shown in Code 8.3 provides data for the same 20 groups of 225 observations in each group.

Code 8.3 Code for clustered normal data.

```
=====
N <- 4500
NGroups <- 20
x1 <- runif(N)
x2 <- runif(N)

Groups <- rep(1:20, each = 225) # 20 groups, each with 225 observations
mu <- 1 + 0.2 * x1 - 0.75 * x2
y <- rnorm(N, mean=mu, sd=2)
ndata <- data.frame(y = y, x1 = x1, x2 = x2, Groups = Groups)
=====
```

8.2.4 Bayesian Random Intercept Normal Model in Python using Stan

For the Stan implementation, we can take advantage of the built-in Cauchy distribution in order to define the priors over the scale parameters. A half-Cauchy distribution is implicitly determined through the constraints imposed in the parameters block (this is analogous to the discussion presented in Section 2.5).

Code 8.4 Random intercept normal model in Python using Stan.

```
=====
import numpy as np
import statsmodels.api as sm
import pystan

from scipy.stats import norm, uniform

# Data
np.random.seed(1656)          # set seed to replicate example
N = 4500                      # number of obs in model
NGroups = 20

x1 = uniform.rvs(size=N)
x2 = uniform.rvs(size=N)
Groups = np.array([225 * [i] for i in range(20)]).flatten()

# Use this if you want random values for a
# a = norm.rvs(loc=0, scale=0.5, size=NGroups)

# Use values from code 8.1
a = np.array([0.579, -0.115, -0.125, 0.169, -0.500, -1.429, -1.171, -0.205,
0.193, 0.041, -0.917, -0.353, -1.197, 1.044, 1.084, -0.085, -0.886, -0.352,
-1.398, 0.350])

mu = 1 + 0.2 * x1 - 0.75 * x2 + a[Groups]
y = norm.rvs(loc=mu, scale=2, size=N)

X = sm.add_constant(np.column_stack((x1,x2)))
K = X.shape[1]
```

```

re = Groups
Nre = NGroups

model_data = {}
model_data['Y'] = y
model_data['X'] = X
model_data['K'] = K
model_data['N'] = N
model_data['NGroups'] = NGroups
model_data['re'] = re
model_data['b0'] = np.repeat(0, K)
model_data['B0'] = np.diag(np.repeat(100, K))
model_data['a0'] = np.repeat(0, Nre)
model_data['A0'] = np.diag(np.repeat(1, Nre))

# Fit
stan_code = """
data{
    int<lower=0> N;
    int<lower=0> K;
    int<lower=0> NGroups;
    matrix[N, K] X;
    real Y[N];
    int re[N];
    vector[K] b0;
    matrix[K, K] B0;
    vector[NGroups] a0;
    matrix[NGroups, NGroups] A0;
}
parameters{
    vector[K] beta;
    vector[NGroups] a;
    real<lower=0> sigma_plot;
    real<lower=0> sigma_eps;
}
transformed parameters{
    vector[N] eta;
    vector[N] mu;

    eta = X * beta;
    for (i in 1:N){
        mu[i] = eta[i] + a[re[i]] + 1;
    }
}
model{
    sigma_plot ~ cauchy(0, 25);
    sigma_eps ~ cauchy(0, 25);

    beta ~ multi_normal(b0, B0);
    a ~ multi_normal(a0, sigma_plot * A0);

    Y ~ normal(mu, sigma_eps);
}
"""

fit = pystan.stan(model_code=stan_code, data=model_data, iter=10000,
                   chains=3, thin=10, warmup=6000, n_jobs=3)

# Output
nlines = 30                                # number of lines in screen output

output = str(fit).split('\n')
for item in output[:nlines]:
    print(item)
=====
      mean se_mean   sd   2.5%   25%   50%   75% 97.5%  n_eff Rhat
beta[0]    0.81  7.8e-3  0.2   0.39   0.68   0.81   0.94  1.22  661.0  1.0
beta[1]    0.14  3.0e-3  0.1  -0.06   0.07   0.13   0.21   0.34 1160.0  1.0
beta[2]   -0.77  3.2e-3  0.1  -0.98  -0.84  -0.77  -0.7  -0.56 1022.0  1.0
a[0]       1.01  8.3e-3  0.22   0.58   0.85   1.0   1.16   1.45  730.0  1.0
a[1]       0.15  8.3e-3  0.23  -0.29  9.9e-3  0.16   0.3    0.6  742.0  1.0
...
a[18]     -1.2  8.2e-3  0.22  -1.65  -1.35  -1.2  -1.05  -0.76  728.0  1.0
a[19]      0.55  9.1e-3  0.23   0.1     0.4    0.55   0.71  1.02  652.0  1.0
sigma_plot 0.73  8.2e-3  0.28   0.36   0.55   0.68   0.87  1.43 1151.0  1.0
sigma_eps  1.98  6.1e-4  0.02   1.94   1.97   1.98    2.0   2.03 1200.0  1.0

```

We have omitted part of the output in order to save space, but the reader can check for the extract shown above that the results are consistent with those obtained with JAGS.

8.3 Bayesian Binary Logistic GLMMs

The data used for the random intercept binary logistic model are similar to those we used for the normal model. In fact, only two lines of code need to be changed to switch to synthetic model data. These are the lines beginning with `mu` and `y`. For the normal model `mu` and `eta` are identical, but that is not the case for the other models we discuss. We therefore have to add another line, transforming the linear predictor `eta` to `mu`. For the binary logistic model we do this using the inverse logit function, $\mu = 1/(1 + \exp(-\eta))$.

8.3.1 Random Intercept Binary Logistic Data

Code 8.5 Simulated random intercept binary logistic data.

```
=====
N <- 4000 #20 groups, each with 200 observations
NGroups <- 20
x1 <- runif(N)
x2 <- runif(N)

Groups <- rep(1:20, each = 200)
a <- rnorm(NGroups, mean = 0, sd = 0.5)
eta <- 1 + 0.2 * x1 - 0.75 * x2 + a[Groups]
mu <- 1/(1+exp(-eta))
y <- rbinom(N, prob=mu, size=1)
logitr <- data.frame(
  y = y,
  x1 = x1,
  x2 = x2,
  Groups = Groups,
  RE = a[Groups]
)
=====

> table(y)
y
  0    1
1604 2396

> head(logitr)
   y      x1      x2 Groups      RE
1 1 0.5210797 0.1304233     1 -0.4939037
2 1 0.2336428 0.1469366     1 -0.4939037
3 1 0.2688640 0.8611638     1 -0.4939037
4 0 0.0931903 0.8900367     1 -0.4939037
5 1 0.4125358 0.7587841     1 -0.4939037
6 0 0.4206252 0.6262765     1 -0.4939037
```

8.3.2 Bayesian Random Intercept Binary Logistic Model with R

For modeling a binary random intercept logistic model, `MCMCglmm` uses the ordinal family. This family allows the estimation of logistic models with two through some eight levels of response. The standard binary logistic model is actually ordered, in that level 1 is considered higher than level 0. This is unlike multinomial or unordered logistic models, where the response levels are considered as independent. Here they are ordered and can therefore be modeled using an ordered logistic algorithm.

Code 8.6 Bayesian Random intercept binary model in R.

```
=====
library(MCMCglmm)

BayLogitRI <- MCMCglmm(y ~ x1 + x2, random= ~Groups,
                         family="ordinal", data=logitr,
                         verbose=FALSE, burnin=10000, nitt=20000)
summary(BayLogitRI)
DIC: 3420.422

G-structure: ~Groups
  post.mean l-95% CI u-95% CI eff.samp
Groups     1.184    0.2851   2.489    22.48

R-structure: ~units
  post.mean l-95% CI u-95% CI eff.samp
units      8.827    3.383   17.12    6.252

Location effects: y ~ x1 + x2

  post.mean  l-95% CI  u-95% CI  eff.samp      pMCMC
(Intercept)  0.97526  0.30427  1.64171  60.49  <0.001 ***
x1          0.49351  0.05572  1.00037  69.19   0.032 *
x2         -0.81374 -1.41232 -0.33242  39.72   0.002 **
```

We give a caveat on using this particular model while relying on `MCMCglmm` outputs. There is typically a great deal of variation in the results, which is not found when modeling the other families we consider in this chapter. In general, though, the more sampling and more groups with larger within-group observations, the more stable the results from run to run. JAGS or Stan models, however, are usually quite close to what we specify for the `intercept` and `x1,x2` coefficient (mean parameter) values.

8.3.3 Bayesian Random Intercept Binary Logistic Model with Python

Modeling a random intercept in pure Python is more complicated, since we are not aware of an available GLMM package. However, it is possible to construct the complete model using `pymc3` in an exercise analogous to the one presented in Code 8.6.

Code 8.7 Bayesian random intercept binary logistic model in Python using `pymc3`.

```
=====
import numpy as np
import pymc3 as pm

from scipy.stats import norm, uniform
from scipy.stats import bernoulli
# Data
np.random.seed(13531)                      # set seed to replicate example
N = 4000                                     # number of obs in model
NGroups = 20

x1 = uniform.rvs(size=N)
x2 = uniform.rvs(size=N)
Groups = np.array([200 * [i] for i in range(20)]).flatten()

a = norm.rvs(loc=0, scale=0.5, size=NGroups)
eta = 1 + 0.2 * x1 - 0.75 * x2 + a[list(Groups)]
mu = 1.0/(1.0 + np.exp(-eta))
y = bernoulli.rvs(mu, size=N)

with pm.Model() as model:
    # Define priors
    sigma = pm.Uniform('sigma', 0, 100)
    sigma_a = pm.Uniform('sigma_a', 0, 10)
    beta1 = pm.Normal('beta1', 0, sd=100)
    beta2 = pm.Normal('beta2', 0, sd=100)
    beta3 = pm.Normal('beta3', 0, sd=100)

    # Priors for random intercept (RI) parameters
    a_param = pm.Normal('a_param',
                        np.repeat(0, NGroups),           # mean
                        sd=np.repeat(sigma_a, NGroups),  # standard
                        deviation                      # number of RI
                        shape=NGroups                  # parameters

    eta = beta1 + beta2*x1 + beta3*x2 + a_param[Groups]

    # Define likelihood
    y = pm.Normal('y', mu=1.0/(1.0 + np.exp(-eta)), sd=sigma, observed=y)
# Fit
start = pm.find_MAP()                         # Find starting value by optimization
step = pm.NUTS(state=start)                   # Initiate sampling
trace = pm.sample(7000, step, start=start)

# Print summary to screen
pm.summary(trace)
=====

beta1:
  Mean        SD      MC Error     95% HPD interval
-----+-----+-----+-----+-----+
  1.140      0.160      0.004      [0.824,  1.450]

  Posterior quantiles:
   2.5          25          50          75         97.5
  |-----+-----+-----+-----+-----|
  0.831      1.032      1.137      1.245      1.461

beta2:
  Mean        SD      MC Error     95% HPD interval
-----+-----+-----+-----+-----+
  0.223      0.121      0.001      [-0.015,  0.459]
```

```

Posterior quantiles:
 2.5          25          50          75         97.5
 |-----|=====|=====|-----|
 -0.014      0.143      0.224      0.305      0.461

beta3:
  Mean        SD       MC Error     95% HPD interval
  -----|-----|-----|-----|-----|
 -0.861    0.126      0.001      [-1.125, -0.629]

Posterior quantiles:
 2.5          25          50          75         97.5
 |-----|=====|=====|-----|
 -1.112    -0.945     -0.860     -0.776     -0.612

a_param:
  Mean        SD       MC Error     95% HPD interval
  -----|-----|-----|-----|-----|
 -0.343    0.183      0.004      [-0.690, 0.024]
 -0.295    0.183      0.004      [-0.671, 0.042]
 -0.192    0.187      0.004      [-0.552, 0.181]
 -0.341    0.184      0.004      [-0.710, 0.014]
 ...
 0.109    0.198      0.004      [-0.266, 0.510]
 0.632    0.224      0.004      [0.201, 1.088]
 0.859    0.249      0.005      [0.398, 1.377]
 0.197    0.199      0.004      [-0.172, 0.601]

Posterior quantiles:
 2.5          25          50          75         97.5
 |-----|=====|=====|-----|
 -0.698    -0.464     -0.343     -0.220      0.018
 -0.656    -0.418     -0.292     -0.170      0.062
 -0.560    -0.320     -0.192     -0.065      0.175
 ...
 0.206    0.480      0.625      0.778      1.100
 0.401    0.688      0.845      1.017      1.385
 -0.191   0.062      0.198      0.331      0.588

sigma:
  Mean        SD       MC Error     95% HPD interval
  -----|-----|-----|-----|-----|
 0.451    0.005      0.000      [0.441, 0.461]

Posterior quantiles:
 2.5          25          50          75         97.5
 |-----|=====|=====|-----|
 0.441    0.447      0.451      0.455      0.461

sigma_a:
  Mean        SD       MC Error     95% HPD interval
  -----|-----|-----|-----|-----|
 0.561    0.112      0.007      [0.353, 0.777]

Posterior quantiles:
 2.5          25          50          75         97.5
 |-----|=====|=====|-----|
 0.376    0.485      0.549      0.619      0.833

```

8.3.4 Bayesian Random Intercept Binary Logistic Model in R using JAGS

Code 8.8 Bayesian random intercept binary logistic model in R using JAGS.

```

=====
library(R2jags)

X <- model.matrix(~ x1 + x2, data = logitr)
K <- ncol(X)
re <- as.numeric(logitr$Groups)
Nre <- length(unique(logitr$Groups))

model.data <- list(
  Y = logitr$y,                                # Response
  X = X,                                         # Covariates
  Nre = Nre,
  K = K,
  re = re)

```

```

K = K,                                # Num. betas
N = nrow(logitr),                      # Sample size
re = logitr$Groups,                     # Random effects
b0 = rep(0,K),
B0 = diag(0.0001, K),
a0 = rep(0,Nre),
A0 = diag(Nre))

sink("GLMM.txt")
cat(")
model {
  # Diffuse normal priors for regression parameters
  beta ~ dmnorm(b0[], B0[,])

  # Priors for random effect group
  a ~ dmnorm(a0, tau.re * A0[,])
  num ~ dnorm(0, 0.0016)
  denom ~ dnorm(0, 1)
  sigma.re <- abs(num / denom)
  tau.re <- 1 / (sigma.re * sigma.re)

  # Likelihood
  for (i in 1:N) {
    Y[i] ~ dbern(p[i])
    logit(p[i]) <- max(-20, min(20, eta[i]))
    eta[i] <- inprod(beta[], X[i,]) + a[re[i]]
  }
}", fill = TRUE)
sink()

inits <- function () {
  list(beta = rnorm(K, 0, 0.01),
       a = rnorm(Nre, 0, 1),
       num = runif(1, 0, 25),
       denom = runif(1, 0, 1))}

params <- c("beta", "a", "sigma.re", "tau.re")

LRI0 <- jags(data = model.data,
               inits = inits,
               parameters = params,
               model.file = "GLMM.txt",
               n.thin = 10,
               n.chains = 3,
               n.burnin = 5000,
               n.iter = 7000)
print(LRI0, intervals=c(0.025, 0.975), digits=3)
=====

      mu.vect sd.vect   2.5%   97.5% Rhat n.eff
a[1]     0.701  0.192   0.346   1.087 1.183   15
a[2]     0.037  0.176  -0.297   0.402 1.093   27
a[3]     0.570  0.186   0.185   0.882 1.049   270
a[4]     0.095  0.168  -0.220   0.410 1.101   24
a[5]     1.331  0.225   0.920   1.796 1.068   36
a[16]    -0.745  0.182  -1.123  -0.389 1.026   80
a[17]    0.002  0.149  -0.258   0.306 1.192   17
a[18]    0.334  0.168  -0.045   0.640 1.237   15
a[19]    0.158  0.163  -0.142   0.444 1.149   17
a[20]    0.230  0.179  -0.093   0.575 1.179   15
beta[1]   1.004  0.135   0.748   1.284 1.169   16
beta[2]   0.264  0.118   0.027   0.488 1.003   540
beta[3]   -0.821  0.118  -1.035  -0.587 1.018   140
sigma.re  0.588  0.111   0.411   0.844 1.004   450
tau.re   3.192  1.142   1.404   5.916 1.004   450
deviance 5139.344  5.409 5130.391 5149.540 1.022   110

pD = 13.8 and DIC = 4761.7

```

The reader can compare the model's results with its original values for `a` by typing `> print(a)` in the R console. It may be instructive for the reader to know why we used `num` and `denom` in the code:

```

num ~ dnorm(0, 0.0016)
denom ~ dnorm(0, 1)
sigma.re <- abs(num / denom)
tau.re <- 1 / (sigma.re * sigma.re)

```

Here `sigma.re` is the mean of the intercept standard deviations. The square of `sigma.re` is the mean variance of the intercepts. As a parameter, it requires a prior. [Gelman \(2006\)](#), [Marley and Wand \(2010\)](#), and [Zuur et al. \(2013\)](#) recommend that a half Cauchy(25) prior be used for the standard deviation parameter, σ_{Groups} . The half Cauchy, as well as the half Cauchy(25), is not a built-in prior in JAGS. Neither is it a built-in prior in OpenBUGS, Stata, or other Bayesian packages. One way to define the half Cauchy(25) prior is as the absolute value of a variable from a $Normal(0, 625)$ distribution divided by a variable with a $Normal(0, 1)$ distribution. That is,

$$\sigma_{Groups} \sim \text{half Cauchy}(25) = \left| \frac{X_1 \sim Normal(0, 625)}{X_2 \sim Normal(0, 1)} \right|. \quad (8.6)$$

The distribution of `num` is $Normal(0, 0.0016)$, where 0.0016 is the precision and 625 is the variance ($1/0.0016$). The denominator is the standard normal distribution. The absolute value of `num/denom` is σ . The inverse of σ^2 is τ . The remainder of the code should be clear.

8.3.5 Bayesian Random Intercept Binary Logistic Model in Python using Stan

This implementation has a lot in common with the one presented in [Code 8.4](#). Modifications as follows need to be made in the `# Data` section,

```
from scipy.stats import bernoulli

a = norm.rvs(loc=0, scale=0.5, size=NGroups)
eta = 1 + 0.2 * x1 - 0.75 * x2 + a[list(Groups)]
mu = 1.0/(1.0 + np.exp(-eta))
y = bernoulli.rvs(mu, size=N)
```

and within the Stan model, where it is necessary to change the response variable declaration to `int` and rewrite the likelihood using a Bernoulli distribution. Given that the Bernoulli distribution uses only one parameter, it is also necessary to suppress the scale parameters.

In what follows we eliminate both `sigma_plot` and `sigma_eps`, and include the parameter `tau_re` in order to make the comparison with [Code 8.8](#) easier. In the same line of thought, the parameter `mu` is given the traditional nomenclature `p` in the context of a Bernoulli distribution.

Notice that it is not necessary to define the logit transformation explicitly since there is a built-in `bernoulli_logit` distribution in Stan:

```
data{
    int<lower=0> Y[N];
}
parameters{
    # remove parameters sigma_plot and sigma_eps from Code 8.4 and add tau_re
    real<lower=0> tau_re;
}
transformed parameters{
    vector[N] p;
```

```

    for (i in 1:N){
      p[i] = eta[i] + a[re[i]+1];
    }
}
model{
  # remove priors for sigma_plot and sigma_eps from Code 8.4
  # add prior for tau_re
  tau_re ~ cauchy(0, 25);
  # rewrite prior for a
  a ~ multi_normal (a0, tau_re * A0);
  Y ~ bernoulli_logit(p);
}

```

This will lead to the following output on the screen:

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
beta[0]	1.13	5.3e-3	0.16	0.82	1.02	1.13	1.23	1.44	963.0	1.0
beta[1]	0.22	3.5e-3	0.12	-6.2e-3	0.14	0.22	0.3	0.46	1193.0	1.0
beta[2]	-0.83	3.5e-3	0.12	-1.06	-0.92	-0.84	-0.76	-0.6	1200.0	1.0
a[0]	-0.36	6.3e-3	0.19	-0.73	-0.49	-0.36	-0.23	8.3e-3	879.0	1.0
a[1]	-0.29	5.8e-3	0.19	-0.67	-0.42	-0.29	-0.18	0.08	1091.0	1.0
a[2]	-0.21	6.2e-3	0.19	-0.57	-0.34	-0.2	-0.08	0.15	957.0	1.0
...										
a[17]	0.62	6.8e-3	0.22	0.19	0.47	0.62	0.77	1.03	1015.0	1.0
a[18]	0.9	7.6e-3	0.22	0.49	0.75	0.89	1.05	1.34	858.0	1.0
a[19]	0.17	6.6e-3	0.2	-0.22	0.03	0.16	0.3	0.57	968.0	1.0
sigma_re	0.35	4.5e-3	0.15	0.16	0.25	0.32	0.43	0.72	1137.0	1.0

8.4 Bayesian Binomial Logistic GLMMs

As discussed in Chapter 5, binomial logistic models are grouped models. The format of the response term is numerator/denominator, where the numerator is the number of successes (for which $y = 1$) with respect to its associated denominator. The denominator consists of separate predictors having the same values. If the predictors x_1 , x_2 , and x_3 have values 1, 0, and 5 respectively, this is a unique denominator, m . If there are 10 observations in the data having that same profile of values, and four of the observations also have a response term y with values of 1 then six observations will have zero for y . In a grouped context, this is one observation: $y = 4$, $m = 10$, $x_1 = 1$, $x_2 = 0$, and $x_3 = 5$. Code 8.9 has 20 groups with values as displayed below. The `Groups` variable may have a clustering effect on the data, which we test using a Bayesian logistic random effects model.

Note that a probit model may also be constructed, using Code 8.10. The only difference is that we substitute

```
probit(p[i]) <- eta[i]
```

```
for
```

```
logit(p[i]) <- eta[i]
```

To reiterate our discussion in Chapter 5 regarding Bayesian probit models, statisticians use a probit model foremost when the binary response term has been bifurcated or dichotomized from a normally distributed continuous variable. In fact, though, most statisticians use a probit model when any non-multimodal continuous variable is dichotomized. If a physicist desires, for example, to model the visible wavelength and the dichotomized wavelength in such a way that $y = 1$ if the light source is greater or equal to 550 nm and $y = 0$ if the source is less than 550 nm then, rather than using a

logistic model for the data, it is likely that a probit model would be selected.

8.4.1 Random Intercept Binomial Logistic Data

Code 8.9 Random intercept binomial logistic data in R.

```
=====
y <- c(6,11,9,13,17,21,8,10,15,19,7,12,8,5,13,17,5,12,9,10)
m <- c(45,54,39,47,29,44,36,57,62,55,66,48,49,39,28,35,39,43,50,36)
x1 <- c(1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0)
x2 <- c(1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0)
Groups <- c(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20)
logitr <- data.frame(y,m,x1,x2,Groups)
=====
```

If we use the R `head` function, the structure of the data may be more easily observed:

```
> head(logitr)
  y  m x1 x2 Groups
1 6 45  1  1      1
2 11 54  1  0      2
3 9 39  1  1      3
4 13 47  1  0      4
5 17 29  1  1      5
6 21 44  1  0      6
```

The random intercept variable `Groups` simply identifies each cluster of data; `m` specifies the number of observations in the respective group. Group 1 has 45 observations, Group 2 has 54 and Group 3 has 39. R does not have a Bayesian random intercept logistic model at present, so we shall advance directly to JAGS.

8.4.2 Bayesian Random Intercept Binomial Logistic Model in R using JAGS

Code 8.10 Random intercept binomial logistic model in R using JAGS.

```
=====
library(R2jags)

X <- model.matrix(~ x1 + x2, data = logitr)
K <- ncol(X)

re <- length(unique(logitr$Groups))
Nre <- length(unique(Groups))

model.data <- list(
  Y = logitr$y,          # response
  X = X,                # covariates
  m = m,                # binomial denominator
  N = nrow(logitr),     # sample size
  re = logitr$Groups,   # random effects
  b0 = rep(0,K),
  B0 = diag(0.0001, K),
  a0 = rep(0,Nre),
  A0 = diag(Nre))

sink("GLMM.txt")
cat(")
model{
  # Diffuse normal priors for regression parameters
  beta ~ dmnorm(b0[], B0[,])

  # Priors for random effect group
  a ~ dmnorm(a0, tau * A0[,])
  num ~ dnorm(0, 0.0016)
  denom ~ dnorm(0, 1)
  sigma <- abs(num / denom)
  tau <- 1 / (sigma * sigma)
```

```

# Likelihood function
for (i in 1:N){
  Y[i] ~ dbin(p[i], m[i])
  logit(p[i]) <- eta[i]
  eta[i] <- inprod(beta[], X[i,]) + a[re[i]]
}
",fill = TRUE)
sink()

inits <- function () {
  list(
    beta = rnorm(K, 0, 0.1),
    a = rnorm(Nre, 0, 0.1),
    num = rnorm(1, 0, 25),
    denom = rnorm(1, 0, 1))}

params <- c("beta", "a", "sigma")

LOGITO <- jags(data = model.data,
                 inits = inits,
                 parameters = params,
                 model.file = "GLMM.txt",
                 n.thin = 10,
                 n.chains = 3,
                 n.burnin = 4000,
                 n.iter = 5000)

print(LOGITO, intervals=c(0.025, 0.975), digits=3)
=====
      mu.vect sd.vect 2.5% 97.5% Rhat n.eff
a[1]    -0.520   0.407 -1.364   0.174 1.021   120
a[2]    -0.415   0.378 -1.224   0.273 1.015   120
a[3]    -0.096   0.422 -0.976   0.632 1.004   300

a[20]     0.083   0.377 -0.567   0.864 1.009   270
beta[1]   -1.113   0.294 -1.704  -0.579 1.009   300
beta[2]    0.270   0.336 -0.429   0.901 1.010   180
beta[3]   -0.292   0.335 -0.921   0.372 1.007   230
sigma      0.645   0.165  0.394   1.097 1.017   100
deviance  97.808   6.425 87.439 111.230 1.037    55

pD = 20.0 and DIC = 117.8

```

We did not use synthetic or simulated data with specified parameter values for this model. We can determine whether the beta values are appropriate, though, by running a maximum likelihood binomial logistic model and comparing parameter values.

```
> cases <- m - y
> mygrpl <- glm(cbind(y, cases) ~ x1 + x2, family=binomial, data=logitr)
> summary(mygrpl)
```

Coefficients:	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	-1.0612	0.1381	-7.683	1.56e-14 ***
x1	0.2365	0.1554	1.522	0.1281
x2	-0.3271	0.1554	-2.105	0.0353 *

Given that the GLM parameter estimates are fairly close to the beta parameter values of the Bayesian model, we can be confident that the code is correct. Remember that the GLM model does not adjust for any grouping effect, nor is it estimated as a Bayesian model. Of course, if we had informative priors to place on any of the Bayesian parameters then the result would differ more. We shall discuss priors later.

8.4.3 Bayesian Random Intercept Binomial Logistic Model in Python using Stan

The Stan model is shown below. As in Subsection 8.3.5 we can avoid the explicit logit transformation by using the Stan built-in `binomial_logit` distribution.

Code 8.11 Random intercept binomial logistic model in Python using Stan.

```

vector[K] b0;
matrix[K, K] B0;
vector[N] a0;
matrix[N, N] A0;
}
parameters{
  vector[K] beta;
  vector[N] a;
  real<lower=0> sigma;
}
transformed parameters{
  vector[N] eta;
  vector[N] p;

  eta = x * beta;
  for (i in 1:N){
    p[i] = eta[i] + a[re[i]+1];
  }
}
model{
  sigma ~ cauchy(0, 25);

  beta ~ multi_normal(b0, B0);
  a ~ multi_normal(a0, sigma * A0);

  Y ~ binomial_logit(m, p);
}
"""

fit = pystan.stan(model_code=stan_code, data=model_data, iter=5000,
chains=3, thin=10, warmup=4000, n_jobs=3)

# Output
nlines = 29 # number of lines in screen output

output = str(fit).split('\n')
for item in output[:nlines]:
  print(item)
=====
      mean se_mean   sd  2.5%   25%   50%   75% 97.5% n_eff Rhat
beta[0] -1.11    0.02  0.33 -1.79 -1.34 -1.11 -0.87 -0.51 300.0  0.99
beta[1]  0.28    0.02  0.38 -0.45  0.03  0.3  0.51  1.03 300.0  0.99
beta[2] -0.32    0.02  0.38 -1.11 -0.55 -0.32 -0.05  0.39 286.0  1.0
a[0]     -0.59    0.03  0.48 -1.57 -0.89 -0.54 -0.29  0.29 261.0  1.0
a[1]     -0.44    0.02  0.4  -1.38 -0.68 -0.45 -0.19  0.29 300.0  1.0
a[2]     -0.07    0.03  0.47 -0.99 -0.36 -0.07  0.2  0.83 300.0  1.0
...
a[17]    0.08    0.02  0.4  -0.73 -0.2   0.1  0.38  0.82 300.0  1.01
a[18]    -0.1    0.03  0.44 -0.9  -0.39 -0.07  0.18  0.78 300.0  0.99
a[19]    0.1     0.02  0.41 -0.76 -0.16  0.09  0.38  0.95 300.0  1.0
sigma    0.54    0.02  0.3  0.16  0.33  0.48  0.66  1.37 269.0  1.0

```

8.5 Bayesian Poisson GLMMs

As discussed in Chapter 5, the Poisson and negative binomial models are foremost in modeling count data. In fact, if one has discrete count data to model then a Poisson model should be estimated first. After that the model should be evaluated to determine whether it is extra-dispersed, i.e. over or underdispersed. This can be evaluated easily by checking the Poisson dispersion statistic, which is calculated as the Pearson χ^2 statistic divided by the residual number of degrees of freedom. If the resulting dispersion statistic is greater than 1, the model is likely to be overdispersed; if under 1, it is likely to be underdispersed. The `P_disp` function in the COUNT package in CRAN can quickly provide the appropriate statistics.

If the Poisson model is overdispersed, the analyst usually employs a negative binomial model on

the data. However, another tactic is first to attempt to determine the likely source of overdispersion. For the binomial and count models, when the data is clustered or is in longitudinal form, the data is nearly always overdispersed. If the data within groups is more highly correlated than is the data between groups, this gives rise to overdispersion. When this is the case the most appropriate tactic is first to model the data using a random intercept Poisson model. If we find that there is variability in coefficients or parameters as well as within groups, it may be necessary to employ a combined random-intercept-random-slopes model. We discuss both these models in this section.

Of course, if there are other sources of excess variation in the data that cannot be accounted for by using a random intercept model, or a combined random intercept–slopes model, then a negative binomial model should be tried. In addition, if the data is not grouped but is still overdispersed then using a negative binomial model is wise. However, it is always vital to attempt to identify the source of extra-dispersion before selecting a count model to use subsequent to Poisson modeling.

If the data to be modeled is underdispersed, modeling the data as a Bayesian random intercept or combined random intercept–slopes might be able to adjust for the underdispersion. A negative binomial model cannot be used with underdispersed data. The best alternative in this case is a Bayesian generalized Poisson model or, better, a Bayesian hierarchical generalized Poisson model.

We begin this section with providing code and an example of a Bayesian Poisson random intercept model.

8.5.1 Random Intercept Poisson Data

Unlike the synthetic normal and binary logistic models created earlier in this chapter, here we only have 10 groups of 200 observations each. However, we retain the same parameter values for the intercept and predictor parameter means as before.

Code 8.12 Random intercept Poisson data in R.

```
=====
N <- 2000                                # 10 groups, each with 200 observations
NGroups <- 10
x1 <- runif(N)
x2 <- runif(N)

Groups <- rep(1:10, each = 200)
a <- rnorm(NGroups, mean = 0, sd = 0.5)
eta <- 1 + 0.2 * x1 - 0.75 * x2 + a[Groups]
mu <- exp(eta)
y <- rpois(N, lambda = mu)
poir <- data.frame(
  y = y,
  x1 = x1,
  x2 = x2,
  Groups = Groups,
  RE = a[Groups])
=====
```

8.5.2 Bayesian Random Intercept Poisson Model with R

The `MCMCglmm` function will be used to estimate random intercept Poisson parameters. We remind the

reader that the results of the pseudo-random number generator can vary considerably. When we specify intercept and predictor parameter mean values we do expect the resulting random data to reflect those values. We did not use a seed in Code 8.12, so each time it is run different values will be produced.

```
> library(MCMCglmm)
> bpglmm <- MCMCglmm(y ~ x1 + x2, random= ~Groups,
+   family="poisson", data=poir, verbose=FALSE,
+   burnin=10000, nitt=20000)
> summary(bpglmm)
DIC: 6980.156

G-structure: ~Groups
post.mean l-95% CI u-95% CI eff.samp
Groups    0.4514    0.1157    1.005     1000

R-structure: ~units
post.mean l-95% CI u-95% CI eff.samp
units   0.006479  0.001411  0.01738    6.452

Location effects: y ~ x1 + x2
post.mean l-95% CI u-95% CI eff.samp      pMCMC
(Intercept) 1.0733  0.6093  1.4626  1000.00 <0.001 ***
x1          0.2303  0.1489  0.3121  28.37 <0.001 ***
x2         -0.7861 -0.8602 -0.7046  36.51 <0.001 ***
```

The intercept and predictor posterior means are fairly close to the values we specified in the code to create the data. For a Bayesian model the disparity may be due to either the randomness of the data or the randomness of the sampling used in the modeling process.

8.5.3 Bayesian Random Intercept Poisson Model in Python

Code 8.13 Bayesian random intercept Poisson model in Python

```
=====
import numpy as np
import pymc3 as pm

from scipy.stats import norm, uniform, poisson

# Data
np.random.seed(1656)                      # set seed to replicate example
N = 2000                                     # number of obs in model
NGroups = 10

x1 = uniform.rvs(size=N)
x2 = uniform.rvs(size=N)

Groups = np.array([200 * [i] for i in range(NGroups)]).flatten()
a = norm.rvs(loc=0, scale=0.5, size=NGroups)
eta = 1 + 0.2 * x1 - 0.75 * x2 + a[list(Groups)]
mu = np.exp(eta)

y = poisson.rvs(mu, size=N)

with pm.Model() as model:
    # Define priors
    sigma_a = pm.Uniform('sigma_a', 0, 100)
    beta1 = pm.Normal('beta1', 0, sd=100)
    beta2 = pm.Normal('beta2', 0, sd=100)
    beta3 = pm.Normal('beta3', 0, sd=100)

    # Priors for random intercept (RI) parameters
    a_param = pm.Normal('a_param',
                        np.repeat(0, NGroups),           # mean
                        sd=np.repeat(sigma_a, NGroups),  # standard deviation
                        shape=NGroups)                 # number of RI parameters
```

```

eta = beta1 + beta2*x1 + beta3*x2 + a_param[Groups]

# Define likelihood
y = pm.Poisson('y', mu=np.exp(eta), observed=y)

# Fit
start = pm.find_MAP()           # Find starting value by optimization
step = pm.NUTS(state=start)     # Initiate sampling
trace = pm.sample(20000, step, start=start)

# Print summary to screen
pm.summary(trace)
=====

beta1:
  Mean        SD      MC Error    95% HPD interval
-----+-----+-----+-----+-----+
  0.966      0.188      0.006      [0.603, 1.336]

Posterior quantiles:
  2.5       25       50       75       97.5
|-----+-----+-----+-----+-----|
  0.605      0.847      0.965      1.083      1.339

beta2:
  Mean        SD      MC Error    95% HPD interval
-----+-----+-----+-----+-----+
  0.214      0.052      0.001      [0.111, 0.316]

Posterior quantiles:
  2.5       25       50       75       97.5
|-----+-----+-----+-----+-----|
  0.111      0.179      0.214      0.249      0.317

beta3:
  Mean        SD      MC Error    95% HPD interval
-----+-----+-----+-----+-----+
 -0.743      0.054      0.001      [-0.846, -0.636]

Posterior quantiles:
  2.5       25       50       75       97.5
|-----+-----+-----+-----+-----|
 -0.848     -0.780     -0.743     -0.707     -0.638

a_param:
  Mean        SD      MC Error    95% HPD interval
-----+-----+-----+-----+-----+
  0.374      0.188      0.005      [-0.011, 0.725]
  0.347      0.189      0.005      [-0.020, 0.722]
  ...
  0.123      0.189      0.005      [-0.262, 0.479]
 -0.446      0.193      0.005      [-0.841, -0.078]

Posterior quantiles:
  2.5       25       50       75       97.5
|-----+-----+-----+-----+-----|
 -0.003     0.256     0.375     0.495     0.737
 -0.032     0.230     0.348     0.466     0.712
  ...
 -0.254     0.004     0.124     0.242     0.488
 -0.839     -0.567    -0.444    -0.322    -0.076

sigma_a:
  Mean        SD      MC Error    95% HPD interval
-----+-----+-----+-----+-----+
  0.569      0.170      0.002      [0.304, 0.905]

Posterior quantiles:
  2.5       25       50       75       97.5
|-----+-----+-----+-----+-----|
  0.339      0.453      0.537      0.648      0.994

```

8.5.4 Bayesian Random Intercept Poisson Model in R using JAGS

Code 8.14 Bayesian random intercept Poisson model in R using JAGS.

```
=====
library(R2jags)
```

```

X <- model.matrix(~ x1 + x2, data=poir)
K <- ncol(X)
re <- as.numeric(poir$Groups)
Nre <- length(unique(poir$Groups))

model.data <- list(
  Y = poir$y, # response
  X = X, # covariates
  K = K, # num. betas
  N = nrow(poir), # sample size
  re = poir$Groups, # random effects
  b0 = rep(0,K),
  B0 = diag(0.0001, K),
  a0 = rep(0,Nre),
  A0 = diag(Nre))

sink("GLMM.txt")
cat("
model {
  # Diffuse normal priors for regression parameters
  beta ~ dmnorm(b0[], B0[,])

  # Priors for random effect group
  a ~ dmnorm(a0, tau.re * A0[,])
  num ~ dnorm(0, 0.0016)
  denom ~ dnorm(0, 1)
  sigma.re <- abs(num / denom)
  tau.re <- 1 / (sigma.re * sigma.re)

  # Likelihood
  for (i in 1:N) {
    Y[i] ~ dpois(mu[i])
    log(mu[i])<- eta[i]
    eta[i] <- inprod(beta[], X[i,]) + a[re[i]]
  }
}
", fill = TRUE)
sink()

inits <- function () {
  list(beta = rnorm(K, 0, 0.01),
       a = rnorm(Nre, 0, 1),
       num = runif(1, 0, 25),
       denom = runif(1, 0, 1))}

# Identify parameters
params <- c("beta", "a", "sigma.re", "tau.re")

# Run MCMC
PRI <- jags(data = model.data,
             inits = inits,
             parameters = params,
             model.file = "GLMM.txt",
             n.thin = 10,
             n.chains = 3,
             n.burnin = 4000,
             n.iter = 5000)
print(PRI, intervals=c(0.025, 0.975), digits=3)
=====

mu.vect sd.vect 2.5% 97.5% Rhat n.eff
a[1]      1.086  0.240   0.609   1.579 1.002 300
a[2]     -0.593  0.246  -1.081  -0.124 1.008 170
a[3]     -0.535  0.247  -1.047  -0.077 1.005 220
a[4]      0.054  0.242  -0.418   0.509 1.009 200
a[5]     -0.670  0.245  -1.175  -0.182 1.004 300
a[6]     -0.528  0.246  -1.004  -0.060 1.006 210
a[7]      0.211  0.243  -0.294   0.685 1.000 300
a[8]     -0.043  0.243  -0.564   0.442 1.004 270
a[9]      0.717  0.241   0.198   1.213 1.009 300
a[10]     0.280  0.239  -0.209   0.743 1.003 300
beta[1]    1.073  0.243   0.581   1.546 1.000 300
beta[2]    0.223  0.045   0.145   0.311 1.011 190
beta[3]   -0.771  0.051  -0.865  -0.669 1.014 150
sigma.re   0.708  0.202   0.427   1.170 1.004 300
tau.re     2.446  1.265   0.730   5.484 1.004 300
deviance 6967.403 4.966 6960.198 6978.731 1.013 300

```

pD = 12.4 and DIC = 6979.8

The results of the intercept and predictor mean posteriors are nearly identical. The square of the standard deviation, `sigma.re`, is close to the variance of the mean intercepts as posted in the `MCMCglmm` output.

8.5.5 Bayesian Random Intercept Poisson Model in Python using Stan

Code 8.15 Bayesian random intercept Poisson model in Python using Stan.

```
=====
import numpy as np
import pystan
import statsmodels.api as sm

from scipy.stats import norm, uniform, poisson

# Data
np.random.seed(1656)                      # set seed to replicate example
N = 2000                                     # number of obs in model
NGroups = 10

x1 = uniform.rvs(size=N)
x2 = uniform.rvs(size=N)

Groups = np.array([200 * [i] for i in range(NGroups)]).flatten()
a = norm.rvs(loc=0, scale=0.5, size=NGroups)
eta = 1 + 0.2 * x1 - 0.75 * x2 + a[list(Groups)]
mu = np.exp(eta)

y = poisson.rvs(mu)

X = sm.add_constant(np.column_stack((x1,x2)))
K = X.shape[1]
Nre = NGroups

model_data = {}
model_data['Y'] = y
model_data['X'] = X
model_data['K'] = K
model_data['N'] = N
model_data['NGroups'] = NGroups
model_data['re'] = Groups
model_data['b0'] = np.repeat(0, K)
model_data['B0'] = np.diag(np.repeat(100, K))
model_data['a0'] = np.repeat(0, Nre)
model_data['A0'] = np.diag(np.repeat(1, Nre))

# Fit
stan_code = """
data{
    int<lower=0> N;
    int<lower=0> K;
    int<lower=0> NGroups;
    matrix[N, K] X;
    int Y[N];
    int re[N];
    vector[K] b0;
    matrix[K, K] B0;
    vector[NGroups] a0;
    matrix[NGroups, NGroups] A0;
}
parameters{
    vector[K] beta;
    vector[NGroups] a;
    real<lower=0, upper=10> sigma_re;
}
transformed parameters{
    vector[N] eta;
    vector[N] mu;

    eta = X * beta;
    for (i in 1:N){
        mu[i] = exp(eta[i] + a[re[i]+1]);
    }
}
```

```

}
model{
  sigma_re ~ cauchy(0, 25);
  beta ~ multi_normal(b0, B0);
  a ~ multi_normal(a0, sigma_re * A0);

  Y ~ poisson(mu);
}
"""

fit = pystan.stan(model_code=stan_code, data=model_data, iter=5000,
                    chains=3, thin=10, warmup=4000, n_jobs=3)

# Output
nlines = 19                                # number of lines in screen output

output = str(fit).split('\n')

for item in output[:nlines]:
  print(item)
=====
      mean se_mean    sd   2.5%   25%   50%   75% 97.5% n_eff Rhat
beta[0]  0.96  0.01  0.21  0.57  0.82  0.95  1.1  1.44  285  1.0
beta[1]  0.22  3.1e-3 0.05  0.12  0.18  0.22  0.25  0.32  247  1.0
beta[2] -0.74  3.0e-3 0.05 -0.86 -0.78 -0.74 -0.71 -0.65  300  1.0
a[0]    0.37  0.01  0.22 -0.11  0.23  0.38  0.52  0.77  274  1.0
a[1]    0.35  0.01  0.21 -0.13  0.22  0.36  0.49  0.74  268  1.0
a[2]    0.15  0.01  0.22 -0.31  0.02  0.16  0.29  0.58  270  1.0
a[3]   -1.12  0.01  0.22 -1.62 -1.24 -1.1  -0.99 -0.67  239  1.0
a[4]    0.16  0.01  0.22 -0.32  0.03  0.16  0.29  0.54  287  1.0
a[5]   -0.27  0.01  0.22 -0.75 -0.41 -0.25 -0.12  0.18  278  1.0
a[6]    0.19  0.01  0.22 -0.3  0.05  0.21  0.35  0.59  270  1.0
a[7]    0.47  0.01  0.21 -2.5e-3 0.35  0.48  0.61  0.89  278  1.0
a[8]    0.12  0.01  0.22 -0.38 -9.9e-3 0.13  0.27  0.55  268  1.0
a[9]   -0.45  0.01  0.22 -0.93 -0.59 -0.44 -0.3  -0.05  274  1.0
sigma_re  0.4   0.01  0.25  0.15  0.24  0.33  0.48  0.99  300  1.0

```

8.5.6 Bayesian Random-Intercept–Random-Slopes Poisson Model

For maximum likelihood models, the random-intercept–random-slopes model assumes that the intercepts of the model vary, as do one or more coefficients or slopes. Of course, in Bayesian modeling all parameters are assumed to be random. Here we quantify the randomness around the x_1 parameter with respect to the random intercepts of $NGroups$. In setting up the data, x_1 is multiplied by the random intercept but the result has a value for the standard deviation that is different from that of the frequentist random intercept–slopes model.

Code 8.16 Random-intercept–random-slopes Poisson data in R.

```

=====
N <- 5000          # 10 groups, each with 500 observations
NGroups <- 10
x1 <- runif(N)
x2 <- ifelse( x1<=0.500, 0, NA)
x2 <- ifelse(x1> 0.500, 1, x2)

Groups <- rep(1:10, each = 500)
a <- rnorm(NGroups, mean = 0, sd = 0.1)
b <- rnorm(NGroups, mean = 0, sd = 0.35)
eta <- 2 + 4 * x1 - 7 * x2 + a[Groups] + b[Groups]*x1
mu <- exp(eta)
y <- rpois(N, lambda = mu)
pric <- data.frame(
  y = y,
  x1 = x1,
  x2 = x2,
  Groups = Groups)
=====
```

We now have data in 10 groups with 500 observations in each. The random intercept and slope

are defined on the `eta` line. The specified parameter predictor means for the model are 2 for the intercept, 4 for `x1`, and `-7` for `x2`. The random intercept is `Groups` as with the previous models, with the random slopes as `x1`.

Code 8.17 Random-intercept–random-slopes Poisson model in R using JAGS.

```
=====
library(R2jags)

X <- model.matrix(~ x1 + x2, data = pric)
K <- ncol(X)
re <- as.numeric(pric$Groups)
NGroups <- length(unique(pric$Groups))

model.data <- list(Y = pric$y,
                     X = X,
                     N = nrow(pric),
                     K = K,
                     b0 = rep(0, K),
                     B0 = diag(0.0001, K),
                     re = re,
                     a0 = rep(0, NGroups),
                     A0 = diag(1, NGroups))

sink("RICGLMM.txt")
cat("
model{
#Priors
beta ~ dmnorm(b0[], B0[,])
a ~ dmnorm(a0[], tau.ri * A0[,])
b ~ dmnorm(a0[], tau.rs * A0[,])
tau.ri ~ dgamma( 0.01, 0.01 )
tau.rs ~ dgamma( 0.01, 0.01 )
sigma.ri <- pow(tau.ri,-0.5)
sigma.rs <- pow(tau.rs,-0.5)

# Likelihood
for (i in 1:N){
    Y[i] ~ dpois(mu[i])
    log(mu[i])<- eta[i]
    eta[i] <- inprod(beta[], X[i,]) + a[re[i]] + b[re[i]] * X[i,2]
}
",
fill = TRUE)
sink()

# Initial values
inits <- function () {
list(
    beta = rnorm(K, 0, 0.01),
    tau = 1,
    a = rnorm(NGroups, 0, 0.1),
    b = rnorm(NGroups, 0, 0.1)
) }

# Identify parameters
params <- c("beta", "sigma.ri", "sigma.rs", "a", "b")

# Run MCMC
PRIRS <- jags(data = model.data,
                 inits = inits,
                 parameters = params,
                 model.file = "RICGLMM.txt",
                 n.thin = 10,
                 n.chains = 3,
                 n.burnin = 3000,
                 n.iter     = 4000)

print(PRIRS, intervals=c(0.025, 0.975), digits=3)
=====

          mu.vect sd.vect    2.5%   97.5%   Rhat n.eff
a[1]      0.073  0.057   -0.027    0.197  0.999  300
a[2]      0.118  0.051    0.027    0.216  1.004  300
a[3]      0.085  0.053   -0.011    0.205  1.003  300
```

a[4]	-0.060	0.053	-0.161	0.049	1.003	300
a[5]	0.045	0.053	-0.054	0.161	1.010	240
a[6]	0.075	0.053	-0.024	0.179	1.003	300
a[7]	-0.042	0.053	-0.138	0.066	1.006	300
a[8]	-0.179	0.052	-0.276	-0.078	1.001	300
a[9]	-0.155	0.055	-0.270	-0.051	1.001	300
a[10]	0.038	0.050	-0.058	0.144	1.003	300
b[1]	-0.276	0.173	-0.611	0.035	1.004	270
b[2]	0.425	0.164	0.099	0.712	0.998	300
b[3]	0.314	0.174	-0.017	0.617	1.005	300
b[4]	-0.352	0.164	-0.691	-0.082	1.000	300
b[5]	-0.447	0.163	-0.764	-0.159	1.002	300
b[6]	0.671	0.157	0.363	0.971	1.034	270
b[7]	-0.274	0.167	-0.589	0.051	1.007	270
b[8]	-0.505	0.167	-0.831	-0.177	1.011	250
b[9]	0.144	0.175	-0.204	0.469	0.999	300
b[10]	0.250	0.165	-0.090	0.566	0.997	300
beta[1]	2.016	0.045	1.918	2.107	1.002	300
beta[2]	3.982	0.146	3.732	4.271	1.005	300
beta[3]	-7.016	0.047	-7.103	-6.923	1.078	37
sigma.ri	0.128	0.034	0.077	0.222	1.020	85
sigma.rs	0.457	0.125	0.284	0.741	1.004	250
deviance	16570.999	6.893	6559.449	16586.785	1.011	150

```
DIC info (using the rule, pD = var(deviance)/2)
pD = 23.6 and DIC = 16594.6
```

As defined earlier, the “a” statistics are the model random intercepts. Each intercept is a separate parameter for which posterior distributions are calculated. The “b” statistics are the random slopes on x_1 , each of which is a posterior. Recall that the synthetic data was specified to have an intercept parameter of 2, with $x_1 = 4$ and $x_2 = -7$. The quantities `sigma.ri` are the square roots of the random intercepts, whereas `sigma.rs` are the standard deviations of the random slopes on x_1 . The model posterior means closely approximate these values. The mean intercept parameter value is 0.128 and the mean random slopes parameter (for x_1) is 0.457.

The equivalent model in Python using Stan is shown below.

Code 8.18 Random-intercept-random-slopes Poisson model in Python using Stan.

```
=====
import numpy as np
import pystan
import statsmodels.api as sm

from scipy.stats import norm, uniform, poisson

# Data
np.random.seed(1656)                      # set seed to replicate example
N = 5000                                     # number of obs in model
NGroups = 10

x1 = uniform.rvs(size=N)
x2 = np.array([0 if item <= 0.5 else 1 for item in x1])

Groups = np.array([500 * [i] for i in range(NGroups)]).flatten()
a = norm.rvs(loc=0, scale=0.1, size=NGroups)
b = norm.rvs(loc=0, scale=0.35, size=NGroups)
eta = 1 + 4 * x1 - 7 * x2 + a[list(Groups)] + b[list(Groups)] * x1
mu = np.exp(eta)

y = poisson.rvs(mu)

X = sm.add_constant(np.column_stack((x1,x2)))
K = X.shape[1]

model_data = {}
model_data['Y'] = y
model_data['X'] = X
model_data['K'] = K
model_data['N'] = N
```

```

model_data['NGroups'] = NGroups
model_data['re'] = Groups
model_data['b0'] = np.repeat(0, K)
model_data['B0'] = np.diag(np.repeat(100, K))
model_data['a0'] = np.repeat(0, NGroups)
model_data['A0'] = np.diag(np.repeat(1, NGroups))

# Fit
stan_code = """
data{
    int<lower=0> N;
    int<lower=0> K;
    int<lower=0> NGroups;
    matrix[N, K] X;
    int Y[N];
    int re[N];
    vector[K] b0;
    matrix[K, K] B0;
    vector[NGroups] a0;
    matrix[NGroups, NGroups] A0;
}
parameters{
    vector[K] beta;
    vector[NGroups] a;
    vector[NGroups] b;
    real<lower=0> sigma_ri;
    real<lower=0> sigma_rs;
}
transformed parameters{
    vector[N] eta;
    vector[N] mu;

    eta = X * beta;
    for (i in 1:N){
        mu[i] = exp(eta[i] + a[re[i]+1] + b[re[i] + 1] * X[i,2]);
    }
}
model{
    sigma_ri ~ gamma(0.01, 0.01);
    sigma_rs ~ gamma(0.01, 0.01);

    beta ~ multi_normal(b0, B0);
    a ~ multi_normal(a0, sigma_ri * A0);
    b ~ multi_normal(a0, sigma_rs * A0);

    Y ~ poisson(mu);
}
"""

fit = pystan.stan(model_code=stan_code, data=model_data, iter=4000,
                   chains=3, thin=10, warmup=3000, n_jobs=3)

# Output
nlines = 30 # number of lines in screen output
output = str(fit).split('\n')

for item in output[:nlines]:
    print(item)

=====
      mean se_mean   sd  2.5%   25%   50%   75% 97.5% n_eff Rhat
beta[0]  1.04  1.9e-3  0.03  0.97  1.02  1.04  1.06  1.1  300  1.01
beta[1]  4.12  8.0e-3  0.13  3.84  4.03  4.13  4.2  4.34  261  1.01
beta[2] -6.88  4.3e-3  0.08 -7.02 -6.93 -6.88 -6.83 -6.73  300  1.0
a[0]     0.06  2.9e-3  0.05 -0.04  0.03  0.06  0.09  0.17  300  1.0
a[1]     0.04  3.0e-3  0.05 -0.06  3.2e-3  0.04  0.07  0.14  300  1.0
...
a[8]     0.05  2.9e-3  0.05 -0.05  0.01  0.05  0.08  0.16  300  0.99
a[9]    -0.03  2.9e-3  0.05 -0.12 -0.06 -0.03  6.7e-3  0.06  300  1.0
b[0]     0.48  9.5e-3  0.16  0.17  0.36  0.47  0.58  0.83  300  1.0
b[1]     0.28  0.01   0.16 -0.04  0.16  0.28  0.37  0.61  225  1.01
...
b[8]    -0.29   0.01   0.17 -0.61  -0.4  -0.29  -0.19  7.9e-3  254  1.01
b[9]    -0.24  9.9e-3  0.16 -0.54  -0.35  -0.24  -0.14  0.09  275  1.0
sigma_ri 9.2e-3  4.5e-4  7.6e-3  1.1e-3  4.2e-3  7.2e-3  0.01  0.02  294  1.0
sigma_rs  0.13  5.4e-3  0.08  0.04   0.08  0.11  0.16  0.38  240  1.0

```

8.6 Bayesian Negative Binomial GLMMs

The negative binomial model is nearly always used to model overdispersed Poisson count data. The data are Poisson when the mean and variance of the model counts are equal. If the variance is greater than the mean, the data is said to be overdispersed; if the variance is less than the mean, the data is underdispersed. We may test for this relationship by checking the Pearson dispersion statistic, which is defined as the value of the model's Pearson χ^2 statistic divided by the residual number of degrees of freedom. If the dispersion statistic is greater than 1, the data is likely to be overdispersed; if the statistic is less than 1, the data is likely to be underdispersed.

As we saw in the previous chapter, however, there are a variety of reasons why count data can be overdispersed: the counts in the model may be structurally missing a zero or there may be far more zero counts than allowed by the Poisson distribution for a given mean. There are a number of other causes of overdispersion, but we do not need to detail them here. However, if you are interested in the ways in which data can be overdispersed and how to ameliorate the problem then see [Hilbe \(2011\)](#) and [Hilbe \(2014\)](#). If a specific remedy for overdispersion in our model data is not known then the negative binomial model should be used, but it needs to be evaluated to determine whether it fits the data properly. There are alternatives to the negative binomial model, though, when dealing with overdispersion, e.g., the generalized Poisson, Poisson inverse Gaussian, and NB-P models. But the negative binomial model usually turns out to be the best model to deal with overdispersion.

Probably the foremost reason for count data being Poisson overdispersed is that the data exhibits a clustering effect. That is, many count data situations entail that the data is nested in some manner or is longitudinal in nature. This generally gives rise to overdispersion. We have given examples of clustered data throughout this chapter. But, like the Poisson model, the negative binomial model can itself be overdispersed – or negative binomial overdispersed – in the sense that there is more correlation in the data than is theoretically expected for a negative binomial with a given mean and dispersion parameter. If the negative binomial Pearson dispersion is greater than 1 then the negative binomial model may be said to be overdispersed or correlated.

The same logic as used in selecting a maximum likelihood model can be used in selecting the appropriate Bayesian model. This logic has been implicit in our discussion of the various Bayesian models addressed in this text. The reason is that the likelihood that is central to the maximum likelihood model is also central to understanding the model data before mixing it with a prior distribution; the structure of the data is reflected in the likelihood. Therefore, when there still exists extra correlation in a hierachial or random intercept Poisson model, which itself is designed to adjust for overdispersion resulting from a clustering variable or variables, then a hierarchical random intercept negative binomial model should be used to model the data.

8.6.1 Random Intercept Negative Binomial Data

Synthetic random intercept data for the negative binomial is set up as for other random intercept

models.

Code 8.19 Random intercept negative binomial data in R.

```
=====
library(MASS)

N <- 2000           # 10 groups, each with 200 observations
NGroups <- 10
x1 <- runif(N)
x2 <- runif(N)

Groups <- rep(1:10, each = 200)
a <- rnorm(NGroups, mean = 0, sd = 0.5)
eta <- 1 + 0.2 * x1 - 0.75 * x2 + a[Groups]
mu <- exp(eta)
y <- rnbinom(mu, size=2)
nbri <- data.frame(
  y = y,
  x1 = x1,
  x2 = x2,
  Groups = Groups,
  RE = a[Groups]
)
=====
```

8.6.2 Random Intercept Negative Binomial MLE Model using R

At the time of writing, there is no code available in R for modeling Bayesian random intercept data. We therefore provide a maximum likelihood model for the same data, comparing it to the JAGS model developed next.

Code 8.20 Random intercept negative binomial mixed model in R.

```
=====
library(gamlss.mx)

nbrani <- gamlssNP(y ~ x1 + x2,
                     data = nbri,
                     random = ~ 1 | Groups,
                     family = NBI,
                     mixture = "gq", k = 20)
summary(nbrani)

Mu link function: log
Mu Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.00436   0.05872 17.104 <2e-16 ***
x1          0.05189   0.07604  0.682  0.495
x2         -0.65532   0.07883 -8.313 <2e-16 ***
z            0.26227   0.01417 18.505 <2e-16 ***

-----
Sigma link function: log
Sigma Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.70860   0.07061 -10.04 <2e-16 ***

-----
No. of observations in the fit: 8000
Degrees of Freedom for the fit: 5
Residual Deg. of Freedom: 1995
at cycle: 1

Global Deviance: 7521.191
AIC: 7531.191
SBC: 7559.195
=====
```

8.6.3 Bayesian Random Intercept Negative Binomial Model using Python

In Python, we are able to use `pymc3` to build a Bayesian model as desired.

Code 8.21 Bayesian random intercept negative binomial mixed model in Python using `pymc3`.

```
=====
import numpy as np
import pymc3 as pm
import statsmodels.api as sm

from scipy.stats import norm, uniform, nbinom

# Data
np.random.seed(1656)                      # set seed to replicate example
N = 2000                                     # number of obs in model
NGroups = 10

x1 = uniform.rvs(size=N)
x2 = uniform.rvs(size=N)

Groups = np.array([200 * [i] for i in range(NGroups)]).flatten()
a = norm.rvs(loc=0, scale=0.5, size=NGroups)
eta = 1 + 0.2 * x1 - 0.75 * x2 + a[Groups]
mu = np.exp(eta)

y = nbinom.rvs(mu, 0.5)

with pm.Model() as model:
    # Define priors
    alpha = pm.Uniform('sigma', 0, 100)
    sigma_a = pm.Uniform('sigma_a', 0, 10)
    beta1 = pm.Normal('beta1', 0, sd=100)
    beta2 = pm.Normal('beta2', 0, sd=100)
    beta3 = pm.Normal('beta3', 0, sd=100)

    # priors for random intercept (RI) parameters
    a_param = pm.Normal('a_param',
                         np.repeat(0, NGroups),           # mean
                         sd=np.repeat(sigma_a, NGroups),  # standard deviation
                         shape=NGroups)                 # number of RI
                                                # parameters

    eta = beta1 + beta2*x1 + beta3*x2 + a_param[Groups]

    # Define likelihood
    y = pm.NegativeBinomial('y', mu=np.exp(eta), alpha=alpha, observed=y)

    # Fit
    start = pm.find_MAP()              # Find starting value by optimization
    step = pm.NUTS(state=start)        # Initiate sampling
    trace = pm.sample(7000, step, start=start)

# Print summary to screen
pm.summary(trace)
=====

beta1:
  Mean          SD       MC Error      95% HPD interval
-----+-----+-----+-----+-----+
  1.049        0.180      0.006      [0.704, 1.417]

  Posterior quantiles:
  2.5          25          50          75          97.5
  |-----+-----+-----+-----+-----|
  0.693        0.939      1.041      1.164      1.409

beta2:
  Mean          SD       MC Error      95% HPD interval
-----+-----+-----+-----+-----+
  0.217        0.072      0.001      [0.078, 0.356]

  Posterior quantiles:
  2.5          25          50          75          97.5
  |-----+-----+-----+-----+-----|
  0.075        0.168      0.217      0.267      0.355

beta3:
  Mean          SD       MC Error      95% HPD interval
-----+-----+-----+-----+-----+
```

```

-0.824          0.075          0.001          [-0.973, -0.684]
Posterior quantiles:
 2.5           25            50            75           97.5
 |-----|=====|=====|=====|-----|
-0.970       -0.873       -0.825       -0.773       -0.679

a_param:
  Mean        SD      MC Error    95% HPD interval
  -----
 0.342        0.180      0.005      [-0.002, 0.715]
 0.235        0.182      0.005      [-0.133, 0.589]
  ...
sigma_a:
  Mean        SD      MC Error    95% HPD interval
  -----
 0.507        0.148      0.003      [0.286, 0.811]

Posterior quantiles:
 2.5           25            50            75           97.5
 |-----|=====|=====|=====|-----|
 0.304       0.406       0.478       0.575       0.872

```

8.6.4 Bayesian Random Intercept Negative Binomial Model in R using JAGS

It should be recalled that, as was the case when modeling the Bayesian negative binomial data in Chapter 5, this model has an extra dispersion parameter. The parameter is nearly always used to adjust for Poisson overdispersed data. We use a half Cauchy(25) prior on the dispersion parameter, which is called `alpha`.

Code 8.22 Bayesian random intercept negative binomial in R using JAGS.

```

=====
library(R2jags)

X <- model.matrix(~ x1 + x2, data = nbri)
K <- ncol(X)
Nre <- length(unique(nbriGroups))

model.data <- list(
  Y = nbriy,          # response
  X = X,              # covariates
  K = K,              # num. betas
  N = nrow(nbri),     # sample size
  re = nbriGroups,    # random effects
  b0 = rep(0,K),
  B0 = diag(0.0001, K),
  a0 = rep(0,Nre),
  A0 = diag(Nre))

sink("GLMM_NB.txt")
cat(")
model {
  # Diffuse normal priors for regression parameters
  beta ~ dmnorm(b0[], B0[,])

  # Priors for random effect group
  a ~ dnorm(a0, tau.re * A0[,])
  num ~ dnorm(0, 0.0016)
  denom ~ dnorm(0, 1)
  sigma.re <- abs(num / denom)
  tau.re <- 1 / (sigma.re * sigma.re)

  # Prior for alpha
  numS ~ dnorm(0, 0.0016)
  denomS ~ dnorm(0, 1)
  alpha <- abs(numS / denomS)

  # Likelihood
  for (i in 1:N) {
    Y[i] ~ dnegbin(p[i], 1/alpha)
    p[i] <- 1 / (1 + alpha * mu[i])
    log(mu[i]) <- eta[i]
  }
}
```

```

        eta[i] <- inprod(beta[], X[i,]) + a[re[i]]
    }
}
", fill = TRUE)
sink()

# Define initial values
inits <- function () {
  list(beta = rnorm(K, 0, 0.1),
       a = rnorm(Nre, 0, 1),
       num = rnorm(1, 0, 25),
       denom = rnorm(1, 0, 1),
       numS = rnorm(1, 0, 25),
       denomS = rnorm(1, 0, 1))}
}

# Identify parameters
params <- c("beta", "a", "sigma.re", "tau.re", "alpha")

NBI <- jags(data = model.data,
              inits = inits,
              parameters = params,
              model.file = "GLMM_NB.txt",
              n.thin = 10,
              n.chains = 3,
              n.burnin = 4000,
              n.iter = 5000)
print(NBI, intervals=c(0.025, 0.975), digits=3)
=====

      mu.vect  sd.vect    2.5%   97.5%   Rhat  n.eff
a[1]     0.587  0.113   0.349   0.787  1.303   10
a[2]     0.749  0.120   0.481   0.962  1.228   13
a[3]    -0.396  0.134  -0.660  -0.144  1.264   11
a[4]    -0.065  0.121  -0.340   0.140  1.285   11
a[5]     0.293  0.136  -0.003   0.518  1.114   22
a[6]    -0.548  0.128  -0.807  -0.320  1.297   11
a[7]    -0.188  0.127  -0.411   0.059  1.277   11
a[8]    -0.262  0.126  -0.506  -0.044  1.171   16
a[9]    -0.179  0.120  -0.413   0.037  1.225   13
a[10]    0.275  0.118   0.012   0.498  1.109   24
alpha    0.493  0.035   0.422   0.562  1.013  130
beta[1]   0.942  0.120   0.755   1.215  1.357    9
beta[2]   0.054  0.068  -0.082   0.179  1.007  300
beta[3]   -0.659  0.080  -0.820  -0.506  1.021   91
sigma.re   0.499  0.149   0.299   0.849  1.022  240
tau.re     4.954  2.507   1.386  11.207  1.022  240
deviance  7488.491  4.999  7480.699  7500.039  1.057   47

pD = 12.0 and DIC = 7500.5

```

The `betas` are similar to the `gamlss` results, as is the value for `alpha`. The `gamlss` negative binomial dispersion parameter is given in log form as `sigma` intercept. The value displayed is **-0.70860**. Exponentiating the value produces the actual dispersion statistic, 0.492 333. This compares nicely with the above Bayesian model result, 0.493.

Recall that the `gamlss` function, which can be downloaded from CRAN, parameterizes the negative binomial dispersion parameter `alpha` in a direct manner. That is, the more variability there is in the data, the greater the value of `alpha`. There is a direct relationship between `alpha` and `mu`. If `alpha` is zero, or close to zero, the model is Poisson.

The `glm` function of R uses an indirect parameterization, calling the dispersion parameter `theta`, where $\alpha = 1/\theta$. Some statisticians who use R prefer to view the dispersion parameter as $1/\theta$, which is in fact the same as `alpha`. Note that all commercial statistical software use a direct parameterization for negative binomial models. You will find, however, many new books on the market that use R in examples follow R's `glm` in inverting the dispersion parameter. Care must be taken, when interpreting negative binomial models, regarding these differing software conventions,

and even when using different functions within the same package. For those readers who prefer to use an indirect parameterization, simply substitute the lines of code marked # in the `Likelihood` section in place of the similar lines above.

Also note that in the above Bayesian output, `sigma.re`, which is the mean standard deviation of the model random intercepts, has the value 0.499. This is very close to the value of `z` in the `gamlss` model output above, which is in variance form: squaring `sigma.re` results in the `z` statistic from the `gamlss` output.

8.6.5 Bayesian Random Intercept Negative Binomial Model in Python using Stan

We show below the same model in Python using Stan. The synthetic data used here was generated using Code 8.21. The user should be aware of the difference between parameterizations built in `scipy` and Stan, which need to be taken into account in the likelihood definition.

Code 8.23 Bayesian random intercept negative binomial in Python using Stan.

```
=====
import pyStan

X = sm.add_constant(np.column_stack((x1,x2)))
K = X.shape[1]

model_data = {}
model_data['Y'] = y
model_data['X'] = X
model_data['K'] = K
model_data['N'] = N
model_data['NGroups'] = NGroups
model_data['re'] = Groups
model_data['b0'] = np.repeat(0, K)
model_data['B0'] = np.diag(np.repeat(100, K))
model_data['a0'] = np.repeat(0, NGroups)
model_data['A0'] = np.diag(np.repeat(1, NGroups))

# Fit
stan_code = """
data{
    int<lower=0> N;
    int<lower=0> K;

    int<lower=0> NGroups;
    matrix[N, K] X;
    int Y[N];
    int re[N];
    vector[K] b0;
    matrix[K, K] B0;
    vector[NGroups] a0;
    matrix[NGroups, NGroups] A0;
}
parameters{
    vector[K] beta;
    vector[NGroups] a;
    real<lower=0> sigma_re;
    real<lower=0> alpha;
}
transformed parameters{
    vector[N] eta;
    vector[N] mu;

    eta = X * beta;
    for (i in 1:N){
        mu[i] = exp(eta[i] + a[re[i]+1]);
    }
}
```

```

model{
  sigma_re ~ cauchy(0, 25);
  alpha ~ cauchy(0, 25);

  beta ~ multi_normal(b0, B0);
  a ~ multi_normal(a0, sigma_re * A0);

  Y ~ neg_binomial(mu, alpha/(1.0 - alpha));
}
"""
fit = pystan.stan(model_code=stan_code, data=model_data, iter=5000,
                   chains=3, thin=10, warmup=4000, n_jobs=3)

# Output
nlines = 20                                # number of lines in screen output

output = str(fit).split('\n')

for item in output[:nlines]:
  print(item)
=====
      mean se_mean    sd  2.5%   25%   50%   75% 97.5% n_eff Rhat
beta[0]  1.08    0.01  0.18  0.77  0.96  1.07  1.19  1.43  300  0.99
beta[1]  0.19  3.8e-3  0.06  0.06  0.15  0.18  0.23  0.32  271  1.01
beta[2] -0.82  4.8e-3  0.07 -0.97 -0.87 -0.82 -0.77 -0.68  226  1.0
a[0]     0.33    0.01  0.17 -0.03  0.23  0.33  0.45  0.64  300  1.0
...
a[9]    -0.31    0.01  0.18 -0.71 -0.43 -0.31 -0.21  0.06  300  1.0
sigma_re 0.31    0.02  0.28  0.09  0.17  0.23  0.38  0.99  300  1.0
alpha    0.5   1.0e-3  0.02  0.47  0.49   0.5   0.52  0.54  300  0.99

```

Further Reading

- de Souza, R. S., J. M. Hilbe, B. Buelens, J. D. Riggs, E. Cameron, E. O. Ishida *et al.* (2015). “The overlooked potential of generalized linear models in astronomy III. Bayesian negative binomial regression and globular cluster populations”. *Mon. Not. Roy. Astronom. Soc.* 453, 1928–1940. DOI: [10.1093/mnras/stv1825](https://doi.org/10.1093/mnras/stv1825). arXiv: [1506.04792 \[astro-ph.IM\]](https://arxiv.org/abs/1506.04792).
- Finch, W. H., J. E. Bolin, and K. Kelley (2014). *Multilevel Modeling Using R*. Chapman & Hall/CRC Statistics in the Social and Behavioral Sciences. Taylor & Francis.
- Zuur, A. F., J. M. Hilbe, and E. N. Ieno (2013). *A Beginner’s Guide to GLM and GLMM with R: A Frequentist and Bayesian Perspective for Ecologists*. Highland Statistics.

9 Model Selection

9.1 Information Criteria Tests for Model Selection

9.1.1 Frequentist and Bayesian Information Criteria

Model selection is of foremost concern in both frequentist and Bayesian modeling. Selection is nearly always based on comparative tests, i.e. two or more models are evaluated and one is determined as having the better fit. This started out as a way to determine whether a particular model with predictors x_1 , x_2 , and x_3 was superior to the same model using only predictors x_1 and x_2 . The goal was to determine whether x_3 significantly contributed to the model. For frequentist-based models, the likelihood ratio test and deviance test were standard ways to determine which model gave the better fit in comparison with alternative models. The likelihood ratio test is currently a very popular method to test predictors as to the best model.

The likelihood ratio test and deviance test require that the models being compared are nested, i.e. one model is nested within a model with more predictors. Broadly speaking, these tests are not appropriate for comparing non-nested models. It was not until the development of information criteria tests that non-nested models could be compared in a meaningful manner. The problem with information criteria tests, though, is that there is no clear way to determine whether one model gives a significantly better fit than another. This problem still exists for information criteria tests.

In the frequentist tradition, in which the majority of models use some type of maximum likelihood algorithm for the estimation of parameters, the most used tests are the AIC and BIC tests and a host of variations. The AIC test is an acronym for the Akaike information criterion, named after Hirotugu Akaike (1927–2009), the Japanese statistician who first developed the test in 1973. Not published until 1974, the statistic has remained the most used test for model selection. Another information test called the Bayesian information criterion test (BIC) was developed by Gideon E. Schwarz (1933–2007) in 1978; this test is commonly referred to as the Schwarz criterion.

The AIC test is given as

$$AIC = -2\mathcal{L} + 2k = -2(\mathcal{L} - k) \quad (9.1)$$

where \mathcal{L} is the log-likelihood of the model and k is the number of predictors and parameters, including the model intercept. If the model predictors and intercept are regarded as parameters then k is simply the number of parameters in the model. This constitutes an adjustment to the likelihood for additional model parameters, which biases the statistic. The more parameters or predictors in a model, the better fitted the model becomes. So, the above adjustment is used to balance the likelihood for models with more parameters. The model with a lower AIC statistic is regarded as the better fitted model. The AIC statistic has been amended when comparing models with different numbers of observations n by dividing the statistic by n . Both these forms of AIC are found in statistical outputs.

The Bayesian information criterion, which does not concern Bayesian models within the frequentist modeling framework, was originally based on the model deviance function rather than on the likelihood. But the test has been converted to log-likelihood form as

$$BIC = -2\mathcal{L} + k \log(n). \quad (9.2)$$

The penalty term is the number of parameters times the log of the number of model observations. This is important, especially when comparing models with different numbers of observations. Again, as with the AIC statistic, a model is considered a better fit if its BIC statistic is lower than that from another model.

For both the AIC and BIC statistics it is assumed that the models being compared come from the same family of probability distributions and that there are significantly more observations in the model than parameters. It is also assumed that the data are independent, i.e., that they are not correlated or clustered. For model comparisons between clustered or panel-level models, no commonly accepted statistic is available (see the book [Hardin and Hilbe, 2012](#)). This is the case for the AIC and BIC statistics as well. A number of alternatives have been developed in the literature to account for the shortcomings of using AIC and BIC.

We should also define the deviance statistic, since it has been used for goodness-of-fit assessment:

$$\text{Deviance} = -2(\mathcal{L}(y, y) - \mathcal{L}(\mu, y)). \quad (9.3)$$

Thus the deviance is minus twice the difference between the “saturated” log-likelihood and the model log-likelihood. The “saturated” likelihood corresponds to an overparameterized model, which represents no more than an interpolation of the data. It is obtained by substituting a y for every μ in the log-likelihood formula for the model. Lower values indicate a smaller difference between the observed and predicted model values.

Remember that in a maximum likelihood model the parameters that are being estimated are fixed. They are unknown parameters of the probability distribution giving rise to the data being modeled. The model is in fact an attempt to determine the most unbiased estimate of the unknown,

but fixed, parameters on the basis of the given data. Bayesian modeling, however, views parameters in an entirely different way. They are random distributions and are not at all fixed. In Bayesian modeling we attempt to determine the posterior distribution for each parameter in the model. The goal is to calculate the mean (or median or mode, depending on the type of distribution involved), standard deviation, and credible intervals of each posterior distribution in the model. This typically requires engaging in an MCMC type of sampling to determine the appropriate posterior distributions given that they are all products of the likelihood and associated prior distributions. This means that the deviance and information criteria for Bayesian models must be based on posterior distributions instead of fixed parameters. We address this concern and interpretation in the following subsections.

9.1.2 Bayesian Deviance Statistic

The log-likelihood is an essential statistic in the estimation of most Bayesian models. For most single-level models, which were discussed in Chapter 5, the log-likelihood statistic is stored in either `L[i]`, `LL[i]`, or `ll[i]`, depending on the model (for R code snippets). The symbols we have used are arbitrary. However, we strongly recommend that you use some symbol that is identical, or similar to what we have used, to represent the log-likelihood in the code. It will be much easier to remember if you need to come back to alter or amend the code, or for anyone else reading your code.

In any case, to obtain the actual model log-likelihood statistic, which is the sum of the individual observation log-likelihoods, we need only to use the R code

```
logLike <- sum(LL[1:N])
```

following the brace closing the calculation of the log-likelihood for each observation in the model. Of course, we have assumed here that `LL[i]` is the symbol for the log-likelihood being used. The code for summing the individual log-likelihoods is calculated directly following the braces defining the log-likelihood module. We can see this better by showing the following JAGS snippet:

```
model{
  # Priors

  # Likelihood
  for (I in 1:N) {
    # any preparatory code

    LL[i] <- calculation of log-likelihood
  }
  LogLike <- sum(LL[1:N])
}
```

We refer to the JAGS Code 5.22 for the binary probit model in Section 5.3.2 for how these statistics appear in a complete model code. The Bayesian deviance statistic is determined by multiplying the model log-likelihood by -2 . Observe the output for the binary probit model mentioned above: the log-likelihood value is displayed using the code we have just explained. Multiply that value, which is displayed in the output as `-814.954`, by -2 and the result is `1629.908`, which is the value displayed in the output for the deviance. Note that this varies from the frequentist definition of deviance. The value is important since it can be used as a rough fit statistic for nested models, with lower values indicating a better fit. The deviance is also used in the creation of the pD and deviance information criterion (DIC) statistic, which is analogous to the AIC statistic for maximum likelihood models:

```
Deviance = -2 * loglikelihood
```

The deviance can be calculated within the JAGS code. One simply adds the following line directly under the log-likelihood function in the model module,

```
dev[i] <- -2 * LLi[i]
```

then summarize it under the summary of the log-likelihood function,

```
Dev <- sum(dev[1:N])
```

Then you need to add `Dev` to the `params` line. We can amend the binary probit model we have been referring to in such a way that the deviance is calculated using the following JAGS code:

```
model{
  # Priors
  beta ~ dmmnorm(b0[], B0[,])

  # Likelihood
  for (i in 1:N){
    Y[i] ~ dbern(p[i])
    probit(p[i]) <- max(-20, min(20, eta[i]))
    eta[i] <- inprod(beta[], X[i,])
    LLi[i] <- Y[i] * log(p[i]) +
      (1 - Y[i]) * log(1 - p[i])
    dev[i] <- -2 * LLi[i]
  }
  LogL <- sum(LLi[1:N])
  Dev <- sum(dev[1:N])
  AIC <- -2 * LogL + 2 * K
  BIC <- -2 * LogL + LogN * K
}
```

Notice that we have retained the AIC and BIC statistics that were calculated in the probit code. These are frequentist versions of AIC and BIC, but with diffuse priors they are satisfactory.

9.1.3 *pD and Deviance Information Criteria (DIC)*

JAGS automatically produces three fit statistics following each model estimation: the deviance, DIC, and pD. We discussed the deviance in Section 9.1.2 and saw how we ourselves can calculate the statistic within the JAGS environment. We now turn to the other two fit test statistics, the pD and DIC statistics. They are nearly always recommended over the AIC and BIC when one is evaluating a Bayesian model, particularly if the model uses informative priors.

The pD statistic specifies the number of effective parameters in the model. The DIC statistic is calculated as the deviance statistic plus pD, which serves as a penalty term, as does $2K$ for the AIC statistic and as does $\log(N)K$ for the BIC statistic. Recall that, for a given model, adding extra parameters to the model generally results in a better model fit. To adjust for this effect, statisticians have designed various penalty terms to add on to the log-likelihood when it is used as a comparative fit statistic. We have provided you with the two most well used penalties, which comprise the standard AIC and BIC statistics. The DIC statistic, or deviance information criterion, employs the pD statistic as its penalty term. The pD statistic evaluates the number of Bayesian parameters that are actually used in the model. Bayesian models, and in particular hierarchical models, can have a large number of parameters, which may be adjusted to reflect their respective

contribution to the model.

The pD statistic can be regarded as a measure of model complexity and is typically calculated as the posterior mean deviance minus the deviance of the posterior means. This is not a simple statistic to calculate, but there is a shortcut: it may also be determined by taking one-half the deviance variance. In JAGS language, this appears as

```
pD <- var(Dev)/2
```

In Code 5.22, the variance of the deviance is not provided in the output but the standard deviation is given as 2.376. The standard deviation results in the variance. Half that then is the pD statistic:

```
> (2.376^2)/2
[1] 2.822688
```

The JAGS output of Code 5.22 displays $pD = 2.8$.

As mentioned before, to obtain the DIC statistic we add the pD term (2.8) we calculated to the deviance (1629.908):

```
DIC <- Dev + pD
```

The DIC statistic as reported by JAGS is 1632.7.

As with all comparative information criterion tests, the models with smaller DIC values are better supported by the data. Usually a difference of about 4 is thought to be meaningful. The expression term “significance difference” makes little statistical sense here. There is no specific test of significance for the comparison of models using any of these information criterion statistics.

9.2 Model Selection with Indicator Functions

One simple approach for variable selection is to define an indicator function I_j that sets a probability for each variable to enter the model:

$$\begin{aligned}\theta &= \theta_0 + \theta_1 x_1 + \cdots + \theta_P x_P, \\ \theta_j &= I_j \times \beta_j, \\ I_j &\in \{0, 1\}.\end{aligned}\tag{9.4}$$

Thus, the indicator function I_j is either 0 or 1 depending on whether the covariate x_j has an influence on the response variable (y). The model is fitted in a Bayesian framework, assuming I_j as another parameter to be determined in such a way that the posterior probability of a variable in the model is determined by the mean value of I_j . In the following we discuss one approach that implements the selection method developed by [Kuo and Mallick \(1998\)](#), but we refer the reader to

O'Hara and Sillanpää (2009) for other examples.

Kuo and Mallick Selection Method

The method sets $\theta_j = I_j \beta_j$, with independent priors placed on each I_j and β_j . The model does not require any *a priori* tuning, but the level of shrinkage can be adapted by placing an informative prior for the probability π for the variable to be included in the model. For a normal model the method can be written as follows:

$$\begin{aligned}
y &\sim \text{Normal}(\mu_i, \sigma^2) \\
\mu_i &= \theta_0 + \theta_1 x_1 + \cdots + \theta_p x_p \\
\theta_j &= I_j \beta_j \\
I_j &\sim \text{Bernoulli}(\pi) \\
\beta_j &\sim \text{Normal}(0, \tau^{-1}) \\
\tau &\sim \text{Uniform}(0, 10) \\
\sigma &\sim \text{Gamma}(10^{-3}, 10^{-3}) \\
i &= 1, \dots, N \\
j &= 1, \dots, P
\end{aligned} \tag{9.5}$$

The addition of the extra parameter τ in β_j helps to avoid the poor mixing of chains when $I_0 = 0$. This happens because, since the priors for I_0 and β are independent, any value produced by the MCMC algorithm will have no effect in the likelihood when $I_0 = 0$. The probability π can be adjusted depending on the level of desired shrinkage. For instance, if we set $\pi = 0.5$, we are expecting on average 50% of the variables to remain in the model; if we set $\pi = 0.1$, we expect 10%; and so forth. An alternative is to add an informative beta prior on π itself, $\pi \sim \text{Beta}(a, b)$. In the following we show how to implement the method in JAGS given in Code 9.3.

First we generate some synthetic multivariate data with an average correlation of 0.6 between the predictors:

Code 9.1 Synthetic multivariate data in R.

```
=====
require(MASS)

# Data
set.seed(1056)
nobs  <- 500                      # number of samples
nvar   <- 15                         # number of predictors
rho    <- 0.6                         # correlation between predictors
p     <- rbinom(nvar, 1, 0.2)
beta  <- round(p * rnorm(nvar, 0, 5), 2)
```

```

# Check the coefficients
print(beta,2)
[1] -2.43 1.60 0.00 5.01 4.12 0.00 0.00
[2] 0.00 -0.89 0.00 -2.31 0.00 0.00 0.00

# Covariance matrix
d      <- length(beta)
Sigma <- toeplitz(c(1, rep(rho, d - 1)))
Mu    <- c(rep(0,d))

# Multivariate sampling
M      <- mvtnorm(nobs, mu = Mu, Sigma = Sigma )
xb    <- M %*% beta
=====

# Dependent variable
y <- rnorm(nobs, xb, sd = 2)
=====
```

To visualize the correlation matrix for our data, we can use the package `corrplot`:

```
require(corrplot)
corrplot(cor(M), method="number", type="upper")
```

with the output shown in Figure 9.1. As we expected, the average correlation between all variables is around 0.6 (see the article [de Souza and Ciardi, 2015](#), for other examples of the visualization of high-dimensional data).

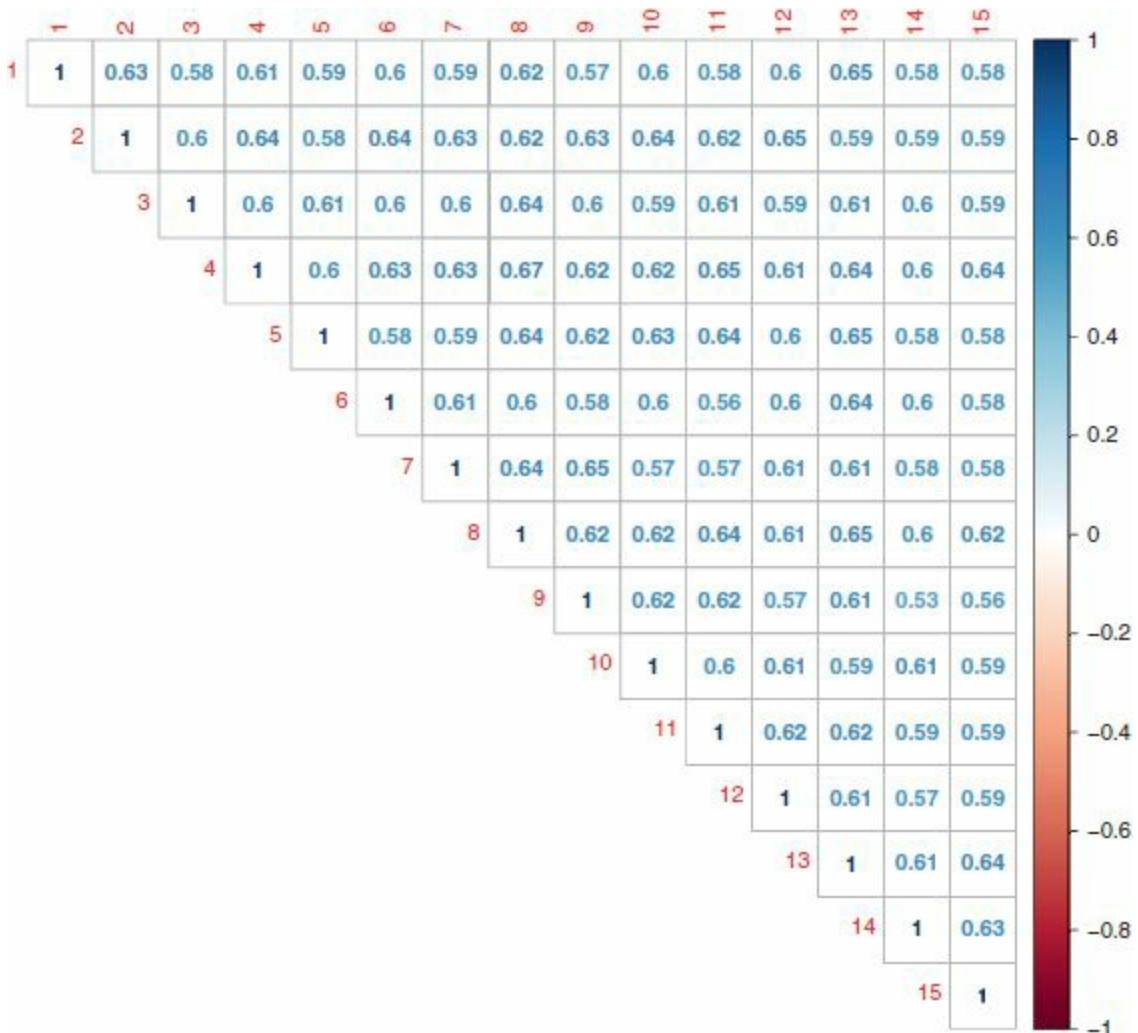


Figure 9.1 Correlation matrix for the synthetic multivariate normal data set.

Now we will fit the data with a normal model to check how well we can recover the coefficients.

Code 9.2 Normal model applied to the multivariate synthetic data from Code 9.1.

```
=====
require(R2jags)

# Prepare data for JAGS
X <- model.matrix(~ M-1)
K <- ncol(X)

jags_data <- list(Y = y,
                    X = X,
                    K = K,
                    N = nobs)

# Fit
NORM <-" model{
  # Diffuse normal priors for predictors
  for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001) }

  # Uniform prior for standard deviation
  tau <- pow(sigma, -2)          # precision
  sigma ~ dgamma(1e-3, 1e-3)      # standard deviation
```

```

# Likelihood function
for (i in 1:N){
  Y[i]~dnorm(mu[i],tau)
  mu[i] <- eta[i]
  eta[i] <- inprod(beta[], X[i,])
}
}"

# Determine initial values
inits <- function () {
  list(beta = rnorm(K, 0, 0.01))
}

# Parameters to display and save
params <- c("beta", "sigma")

# MCMC
NORM_fit <- jags(data = jags_data,
  inits = inits,
  parameters = params,
  model = textConnection(NORM),
  n.chains = 3,
  n.iter = 5000,
  n.thin = 1,
  n.burnin = 2500)
=====
```

In order to visualize how close the coefficients are to the original values, we show a caterpillar plot in Figure 9.2, which can be achieved with the following code:

```

require(mcmcplots)

caterplot(NORM_fit,"beta",denstrip = T,
  greek = T,style = "plain", horizontal = F,
  reorder = F,cex.labels = 1.25,col="gray35")
caterpoints(beta, horizontal=F, pch="x", col="cyan")
```

Note that, although the normal model recovers the coefficients well, it does not set any of them to exactly zero, so all the coefficients remain in the model. Next we show how to implement the variable selection model in JAGS, placing an informative prior of 0.2 in the probability for a given predictor to stay in the model:

Code 9.3 K and M model applied to multivariate synthetic data from Code 9.1.

```

=====
NORM_Bin <-" model{
  # Diffuse normal priors for predictors
  tauBeta <- pow(sdBeta,-2);
  sdBeta ~ dgamma(0.01,0.01)
  PInd <- 0.2

  for (i in 1:K){
    Ind[i] ~ dbern(PInd)
    betat[i] ~ dnorm(0,tauBeta)
    beta[i] <- Ind[i]*betat[i]
  }
  # Uniform prior for standard deviation
  tau <- pow(sigma, -2)          # precision
  sigma ~ dgamma(0.01,0.01)      # standard deviation

  # Likelihood function
  for (i in 1:N){
    Y[i]~dnorm(mu[i],tau)
    mu[i] <- eta[i]
    eta[i] <- inprod(beta[], X[i,])
  }
}"
=====
```

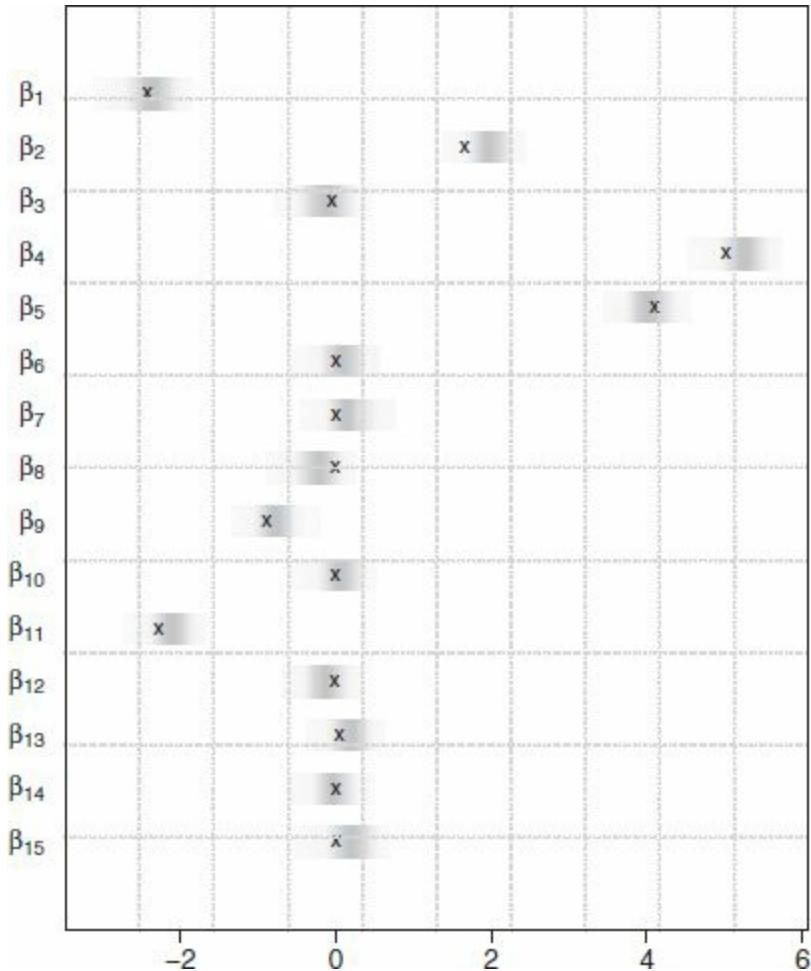


Figure 9.2 Visualization of results from the normal model fit to synthetic multivariate data from Code 9.1. The gray regions correspond to the posteriors of each parameter and the crosses to their true values.

Figure 9.3 shows the estimated variables for the K and M model. Note that now the model actually set the coefficients to zero, so we can find a sparse solution for it.

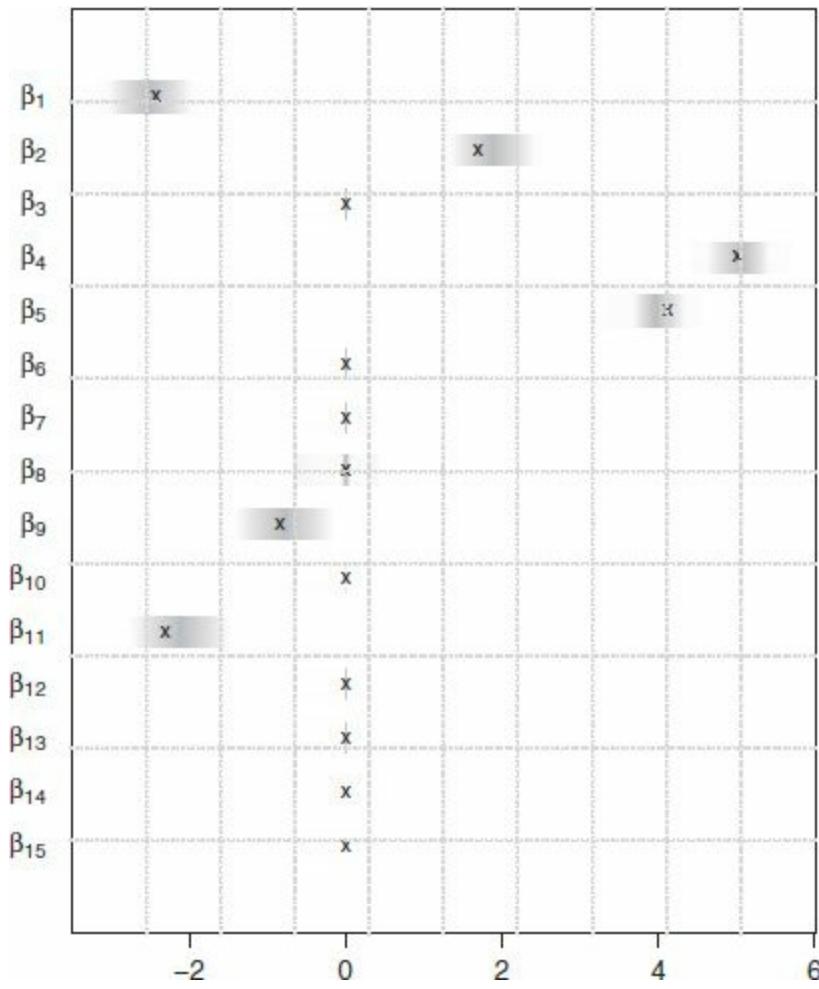


Figure 9.3 Visualization of results from the K and M model fit to synthetic multivariate data from Code 9.1. The gray regions correspond to the posteriors of each parameter and the crosses to their true values.

The equivalent code in Python using Stan is displayed below. Notice that, given the different parameterizations for the Gaussian distribution built in JAGS and Stan, the exponents for `tau` and `beta` have opposite signs.

Code 9.4 K and M model in Python using Stan.

```
=====
import numpy as np
import pystan

from scipy.linalg import toeplitz
from scipy.stats import norm, uniform, nbinom, multivariate_normal, bernoulli

# Data
np.random.seed(1056)

nobs = 500
nvar = 15
rho = 0.6
p = bernoulli.rvs(0.2, size=nvar)

# Use these to generate new values of beta
# beta1 = p * norm.rvs(loc=0, scale=5.0, size=nvar)
# beta = np.array([round(item, 2) for item in beta1])

# Use same beta values as Code 9.1
```

```

beta = np.array([-2.43, 1.60, 0.00, 5.01, 4.12, 0.00, 0.00, 0.00,
                 -0.89, 0.00, -2.31, 0.00, 0.00, 0.00, 0.00])

# Covariance matrix
d = beta.shape[0]
Sigma = toeplitz(np.insert(np.repeat(rho, d-1), 0, 1))

# Multivariate sampling - default mean is zero
M = multivariate_normal.rvs(cov=Sigma, size=nobs)
xb = np.dot(M, beta)

y = [norm.rvs(loc=xb[i], scale=2.0) for i in range(xb.shape[0])]

# Fit
mydata = {}
mydata['X'] = M - 1.0
mydata['K'] = mydata['X'].shape[1]
mydata['Y'] = y
mydata['N'] = nobs
mydata['Ind'] = bernoulli.rvs(0.2, size = nvar)

stan_model = '''
data{
    int<lower=0> N;
    int<lower=0> K;
    matrix[N, K] X;
    vector[N] Y;
    int Ind[K];
}
parameters{
    vector[K] beta;
    real<lower=0> sigma;
    real<lower=0> sdBeta;
}
transformed parameters{
    vector[N] mu;
    real<lower=0> tau;
    real<lower=0> tauBeta;
    mu = X * beta;
    tau = pow(sigma, 2);
    tauBeta = pow(sdBeta, 2);
}
model{
    sdBeta ~ gamma(0.01, 0.01);

    for (i in 1:K){
        if (Ind[i] > 0) beta[i] ~ normal(0, tauBeta);
    }
    sigma ~ gamma(0.01, 0.01);

    Y ~ normal(mu, tau);
}'''

fit = pystan.stan(model_code=stan_model, data=mydata, iter=5000, chains=3,
                   thin=1, warmup=2500, n_jobs=3)

# Output
nlines = 21                                # number of lines in screen output

output = str(fit).split('\n')

for item in output[:nlines]:
    print(item)
=====

          mean se_mean   sd  2.5%   25%   50%   75% 97.5% n_eff Rhat
beta[0]  -2.53  6.7e-3  0.26 -3.04  -2.71  -2.53  -2.36 -2.04  1483  1.0
beta[1]   1.11  7.2e-3  0.26  0.62   0.94   1.11   1.28  1.62  1275  1.0
beta[2]  -0.45  6.8e-3  0.27 -0.98  -0.63  -0.46  -0.27  0.07  1571  1.0
beta[3]   4.75  6.9e-3  0.26  4.25   4.58   4.74   4.92  5.27  1417  1.0
beta[4]   4.29  6.0e-3  0.25  3.8    4.11   4.29   4.45  4.78  1743  1.0
beta[5]  -0.32  6.5e-3  0.26 -0.84  -0.5   -0.32  -0.15  0.17  1593  1.0
beta[6]  -0.59  6.7e-3  0.25 -1.06  -0.77  -0.6   -0.43 -0.07  1434  1.0
beta[7]  -0.28  6.9e-3  0.25 -0.75  -0.44  -0.28  -0.11  0.21  1264  1.0
beta[8]  -1.29  8.1e-3  0.27 -1.83  -1.46  -1.29  -1.12 -0.77  1084  1.0
beta[9]   0.28  6.6e-3  0.25 -0.21  0.12   0.28   0.44  0.78  1415  1.0

```

```

beta[10] -2.41 6.7e-3 0.25 -2.9 -2.57 -2.4 -2.24 -1.93 1361 1.0
beta[11] -0.2 8.6e-3 0.26 -0.69 -0.37 -0.2 -0.02 0.3 882 1.0
beta[12] -0.12 6.1e-3 0.24 -0.59 -0.28 -0.12 0.05 0.36 1578 1.0
beta[13] -3.2e-3 6.8e-3 0.25 -0.49 -0.17 3.5e-4 0.16 0.48 1319 1.0
beta[14] -0.26 7.5e-3 0.25 -0.78 -0.43 -0.25 -0.08 0.24 1154 1.0
sigma     1.92 8.0e-4 0.03 1.86 1.89 1.91 1.94 1.98 1374 1.0

```

9.3 Bayesian LASSO

The least absolute shrinkage and selection operator (LASSO) is an alternative approach to the use of indicators in a model. The prior shrinks the coefficients β_j towards zero unless there is strong evidence for them to remain in the model.

The original LASSO regression was proposed by [Tibshirani \(1996\)](#) to automatically select a relevant subset of predictors in a regression problem by shrinking some coefficients towards zero (see also [Uemura et al., 2015](#), for a recent application of LASSO for modeling type Ia supernovae light curves). For a typical linear regression problem,

$$y_i = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p + \varepsilon, \quad (9.6)$$

where ε denotes a Gaussian noise. LASSO estimates the linear regression coefficients $\beta = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$ by imposing a L_1 -norm penalty in the form

$$\operatorname{argmin}_{\beta} \left\{ \sum_{i=1}^N \left(y_i - \sum_{j=1}^p \beta_j x_{ij} \right)^2 + \kappa \sum_{j=1}^p |\beta_j| \right\}, \quad (9.7)$$

where $\kappa \geq 0$ is a constant that controls the level of sparseness in the solution. The number of zero coefficients thereby increases as κ increases. [Tibshirani](#) also noted that the LASSO estimate has a Bayesian counterpart when the β coefficients have a double-exponential prior (i.e., a Laplace prior) distribution:

$$f(x, \tau) = \frac{1}{2\tau} \exp\left(-\frac{|x|}{\tau}\right), \quad (9.8)$$

where $\tau = 1/\kappa$ is the scale. The idea was further developed in what is known as Bayesian LASSO (see e.g., [de Souza et al., 2015b](#); [Park et al., 2008](#)). The role of the Laplace prior is to assign more weight to regions either near to zero or in the distribution tails than would a normal prior. The implementation in JAGS is very straightforward; the researcher just needs to replace the normal priors in the β coefficients in the Laplace prior by

```
=====
tauBeta <- pow(sdBeta, -1)
sdBeta ~ dgamma(0.01, 0.01)
for (i in 1:K){
beta[i]~ddexp(0, tauBeta)
}
=====
```

In Stan, the corresponding changes to be applied to Code 9.4 are

```
...
transformed parameters{
  ...
  tauBeta = pow(sdBeta, -1);
  ...
}
model{
  ...
  for (i in 1:K) beta[i] ~ double_exponential(0, tauBeta);
  ...
}
```

We refer the reader to [O'Hara and Sillanpää \(2009\)](#) for a practical review of Bayesian variable selection with code examples.

Further Reading

- Gelman, A., J. Carlin, H. Stern, D. Dunson, A. Vehtari, and D. Rubin (2013). *Bayesian Data Analysis, Third Edition*. Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis.
- Gelman, A., J. Hwang, and A. Vehtari (2014). “Understanding predictive information criteria for Bayesian models.” *Statist. Comput.* 24(6), 997–1016. DOI: [10.1007/s11222-013-9416-2](https://doi.org/10.1007/s11222-013-9416-2).
- O’Hara, R. B. and M. J. Sillanpää (2009). “A review of Bayesian variable selection methods: what, how and which.” *Bayesian Anal.* 4(1), 85–117. DOI: [10.1214/09-ba403](https://doi.org/10.1214/09-ba403).
- Spiegelhalter, D. J., N. G. Best, B. P. Carlin, and A. Van der Linde (2002). “Bayesian measures of model complexity and fit.” *J. Royal Statist. Soc.:Series B (Statistical Methodology)* 64(4), 583–639. DOI: [10.1111/1467-9868.00353](https://doi.org/10.1111/1467-9868.00353).
- Vehtari, A. and J. Ojanen (2012). “A survey of Bayesian predictive methods for model assessment, selection and comparison.” *Statist. Surv.* 6, 142–228. DOI: [10.1214/12-SS102](https://doi.org/10.1214/12-SS102).

10 Astronomical Applications

This chapter presents a series of astronomical applications using some of the models presented earlier in the book. Each section concerns a specific type of astronomical data situation and the associated statistical model. The examples cover a large variety of topics, from solar (sunspots) to extragalactic (type Ia supernova) data, and the accompanying codes were designed to be easily modified in order to include extra complexity or alternative data sets. Our goal is not only to demonstrate how the models presented earlier can impact the classical approach to astronomical problems but also to provide resources which will enable both young and experienced researchers to apply such models in their daily analysis.

Following the same philosophy as in previous chapters, we provide codes in R/JAGS and Python/Stan for almost all the case studies. The exceptions are examples using type Ia supernova data for cosmological parameter estimation and approximate Bayesian computation (ABC). In the former we take advantage of the Stan ordinary differential equation solver, which, at the time of writing, is not fully functional within PyStan.¹ Thus, we take the opportunity to show one example of how Stan can also be easily called from within R. The ABC approach requires completely different ingredients and, consequently, different software. Here we have used the `cosmoabc`² Python package in order to demonstrate how the main algorithm works in a simple toy model. We also point the reader to the main steps towards using ABC for cosmological parameter inference from galaxy cluster number counts. This is considered an advanced topic and is presented as a glimpse of the potential of Bayesian analysis beyond the exercises presented in previous chapters. Many models discussed in this book represent a step forward from the types of models generally used by the astrophysical community. In the future we expect that Bayesian methods will be the predominant statistical approach to the analysis of astrophysical data.

Accessing Data

For all the examples presented in this chapter we will use publicly available astronomical data sets. These have been formatted to allow easy integration with our R and Python codes. All the code snippets in this chapter contain a `path_to_data` variable, which has the format `path_to_data = "~<some path>"`, where the symbol `~` should be substituted for the complete path to our GitHub repository.³ In doing so, given an internet connection the data will be read on the fly from the online source. This format was chosen to avoid long paths within the code snippets.

10.1 Normal Model, Black Hole Mass, and Bulge Velocity Dispersion

We begin by addressing the popular normal model in a scenario which is unavoidable in astronomical research: the presence of errors in measurements (see Section 4.3 for a model description and an example using synthetic data). Our first practical application illustrates the use of a Bayesian Gaussian model for describing the relation between the mass M_\bullet of the supermassive black hole present in the center of a galaxy and the velocity dispersion σ_e of the stars in its bulge.⁴

The study of the global properties of individual galaxies attracted attention in the astronomical community for a while (Djorgovski and Davis, 1987). However, only in the early twenty-first century did the data paradigm allow a quantitative approach to the problem. As soon as this data became available, the so-called M_\bullet - σ_e relation took the center of attention. The first appearance of the relation was in Merritt (2000) and took the form

$$\frac{M_\bullet}{10^8 M_\odot} \approx 3.1 \left(\frac{\sigma_e}{200 \text{ km s}^{-1}} \right)^4, \quad (10.1)$$

originally known as the *Faber–Jackson law* for black holes. Here M_\odot is the mass of the Sun.⁵ Subsequently, Ferrarese and Merritt (2000) and Gebhardt *et al.* (2000) reported a correlation tighter than had previously been expected but with significantly different slopes, which started a vivid debate (see Harris *et al.* 2013, hereafter H2013, and references therein). Merritt and Ferrarese (2001) showed that such discrepancies were due to the different techniques used to derive M_\bullet . Since then an updated relation has allowed the determination of central black hole masses in distant galaxies, where σ_e is easily measured (Peterson, 2008).

In this example we follow H2013, modeling the M_\bullet - σ_e relation as

$$\log \frac{M_\bullet}{M_\odot} = \alpha + \beta \log \frac{\sigma}{\sigma_0}, \quad (10.2)$$

where σ_0 is a reference value, usually chosen to be 200 km s^{-1} (Tremaine *et al.*, 2002); α, β are the linear predictor coefficients to be determined. Notice that in order to make the notation lighter we have suppressed the subscript e from the velocity dispersion in Equation 10.2.

10.1.1 Data

The data set used in this example was presented by H2013 (see also Harris *et al.*, 2014).⁶ This is a compilation of literature data from a variety of sources obtained with the Hubble Space Telescope as well as with a wide range of other ground-based facilities. The original data set was composed of data from 422 galaxies, 46 of which have available measurements of M_\bullet and σ .

10.1.2 The Statistical Model Formulation

Following the model presentation adopted in previous chapters, we show below the mathematical representation of our statistical model, which we encourage the reader to use when publishing similar analyses. This makes it easier for third parties to reproduce and test the model in a variety of different programming languages.

The logarithm of the *observed* central black hole mass, $M \equiv \log(M_\bullet/M_\odot)$, represents our response variable and the logarithm of the *observed* velocity dispersion is our explanatory variable, $\sigma \equiv \log(\sigma/\sigma_0)$. Their *true* values are denoted by $M^{\text{true}} \equiv \log(M_\bullet^{\text{true}}/M_\odot)$ and $\sigma^{\text{true}} \equiv \log(\sigma^{\text{true}}/\sigma_0)$ respectively. In order to take into account the measurement errors in both variables ($\varepsilon_M, \varepsilon_\sigma$) we follow the procedure presented in Chapter 4. In the following statistical model, $N = 46$ is the total number of galaxies, $\eta = \mu = \alpha + \beta\sigma$ is the linear predictor, α is the intercept, β is the slope, and ε is the intrinsic scatter around μ :

$$\begin{aligned}
 M_i &\sim \text{Normal}(M_i^{\text{true}}, \varepsilon_{M,i}^2) \\
 M_i^{\text{true}} &\sim \text{Normal}(\mu_i, \varepsilon^2) \\
 \mu_i &= \alpha + \beta\sigma_i \\
 \sigma_i &\sim \text{Normal}(\sigma_i^{\text{true}}, \varepsilon_{\sigma,i}^2) \\
 \sigma_i^{\text{true}} &\sim \text{Normal}(0, 10^3) \\
 \alpha &\sim \text{Normal}(0, 10^3) \\
 \beta &\sim \text{Normal}(0, 10^3) \\
 \varepsilon^2 &\sim \text{Gamma}(10^{-3}, 10^{-3}) \\
 i &= 1, \dots, N
 \end{aligned} \tag{10.3}$$

where we assign non-informative priors for α, β, σ , and ε .

10.1.3 Running the Model in R using JAGS

Code 10.1 Normal linear model in R using JAGS for accessing the relationship between central black hole mass and bulge velocity dispersion.

```
=====
require(R2jags)

# Data
path_to_data = "~/data/Section_10p1/M_sigma.csv"

# Read data
MS<-read.csv(path_to_data, header = T)

# Identify variables
N      <- nrow(MS)                      # number of data points
```

```

obsx <- MS$obsx          # log black hole mass
errx <- MS$errx          # error on log black hole mass
obsy <- MS$obsy          # log velocity dispersion
erry <- MS$erry          # error on log velocity dispersion

# Prepare data to JAGS
MS_data <- list(
  obsx = obsx,
  obsy = obsy,
  errx = errx,
  erry = erry,
  N = N
)

# Fit
NORM_errors <- "model{

  # Diffuse normal priors for predictors
  alpha ~ dnorm(0,1e-3)
  beta ~ dnorm(0,1e-3)

  # Gamma prior for scatter
  tau ~ dgamma(1e-3,1e-3)           # precision
  epsilon <- 1/sqrt(tau)           # intrinsic scatter

  # Diffuse normal priors for true x
  for (i in 1:N){ x[i]~dnorm(0,1e-3) }

  for (i in 1:N){
    obsx[i] ~ dnorm(x[i], pow(errx[i], -2))
    obsy[i] ~ dnorm(y[i], pow(erry[i], -2))      # likelihood function
    y[i] ~ dnorm(mu[i], tau)
    mu[i] <- alpha + beta*x[i]                   # linear predictor
  }
}

# Define initial values
inits <- function () {
  list(alpha = runif(1,0,10),
       beta = runif(1,0,10))
}

# Identify parameters
params0 <- c("alpha", "beta", "epsilon")

# Fit
NORM_fit <- jags(data = MS_data,
  inits = inits,
  parameters = params0,
  model = textConnection(NORM_errors),
  n.chains = 3,
  n.iter = 50000,
  n.thin = 10,
  n.burnin = 30000
)

# Output
print(NORM_fit,justify = "left", digits=2)
=====
  mu.vect   sd.vect    2.5%   97.5%   Rhat   n.eff
alpha     8.35     0.05    8.24    8.46     1    3000
beta      4.44     0.33    3.80    5.06     1    3000
epsilon   0.27     0.06    0.17    0.39     1    3000
deviance -225.99    13.84   -250.85   -196.55    1    3000

pD = 95.8 and DIC = -130.2

```

The above results are consistent with the values found by H2013. The present authors used the method presented in [Tremaine *et al.* \(2002\)](#), which searched for α and β parameters that minimize

$$\chi^2 = \sum_{i=1}^N \frac{[y_i - \alpha - \beta(x_i - \langle x \rangle)]^2}{(\sigma_{y;i}^2 + \varepsilon_y^2) + \beta^2(\sigma_{x;i}^2 + \varepsilon_x^2)}. \quad (10.4)$$

From Table 2 of H2013 the reported parameter values are $\alpha = 8.412 \pm 0.067$ and $\beta = 4.610 \pm 0.403$. The general trend is recovered, despite the distinct assumptions in modeling and error treatment (see H2013, Section 4.1).

An indication of how the final model corresponds to the original data is shown in Figure 10.1, where the dashed line represents the mean fitted result and the shaded regions denote 50% (darker) and 95% (lighter) prediction intervals. Posterior marginal distributions for the intercept α , slope β , and intrinsic scatter ε are shown in Figure 10.2.

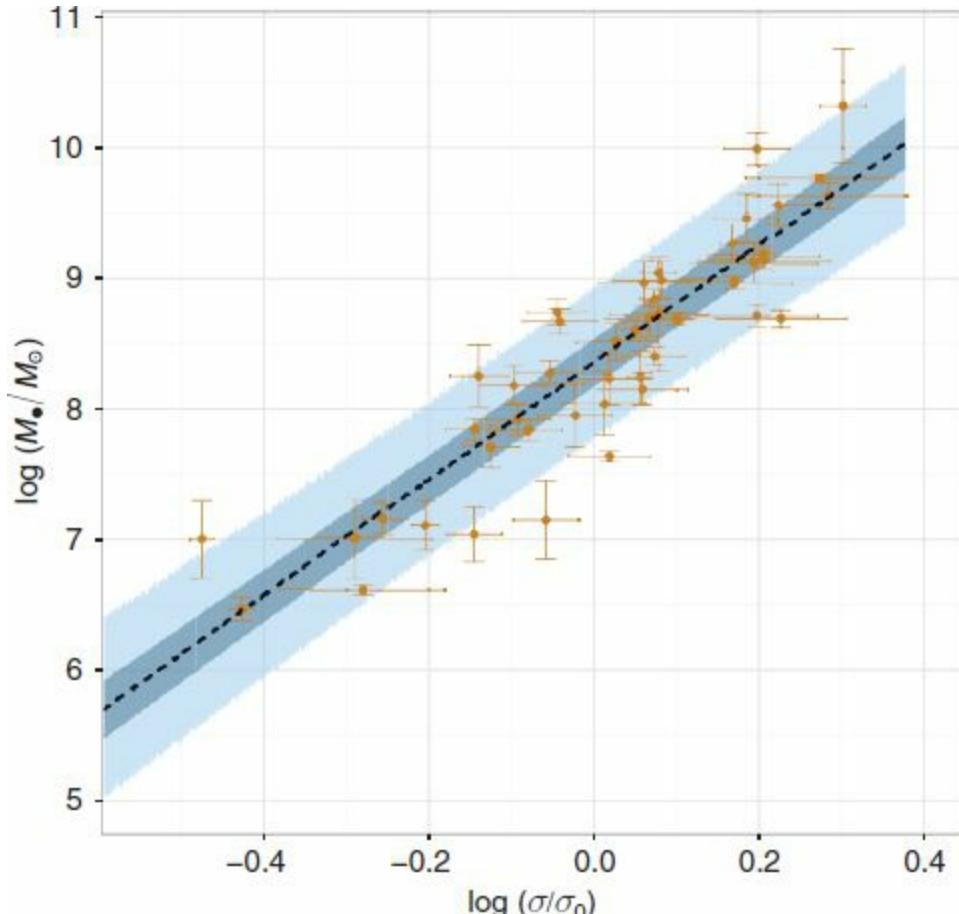


Figure 10.1 Supermassive black hole mass as a function of bulge velocity dispersion described by a Gaussian model with errors in measurements. The dashed line represents the mean and the shaded areas represent the 50% (darker) and 95% (lighter) prediction intervals. The dots and associated error bars denote the observed values and measurement errors respectively.

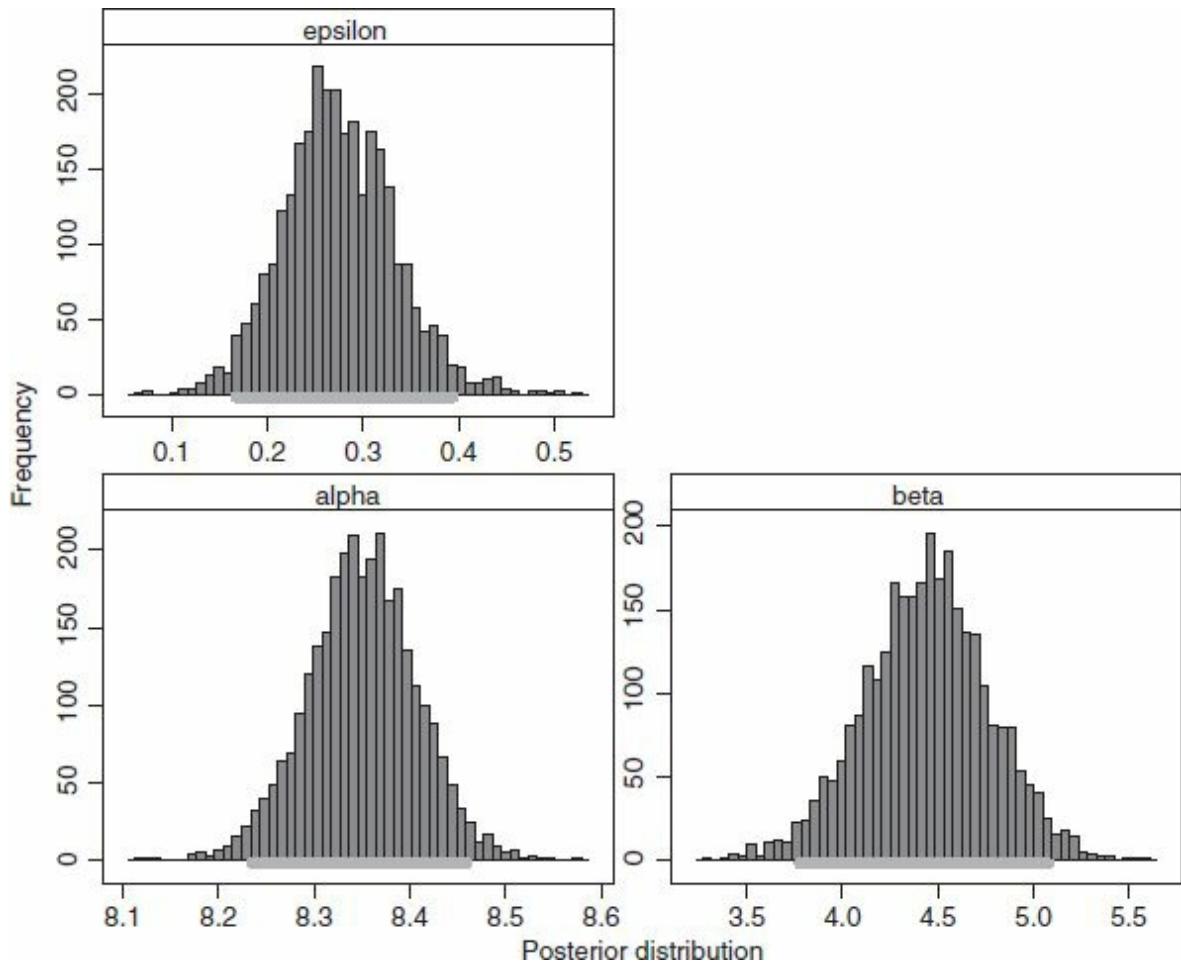


Figure 10.2 Posterior distributions for the intercept (α), slope (β), and intrinsic scatter (ϵ). The horizontal thick lines mark the 95% credible intervals.

10.1.4 Running the Model in Python using Stan

As usual, the corresponding Stan model is very similar to its JAGS counterpart. The main difference, which we must again emphasize, is the necessity to match the variable domain defined in the `parameters` block with the domain of the prior distributions declared in the `model` block (see [Team Stan, 2016](#), Section 3.2).

In what follows we have employed a more compact notation than that presented in [Code 10.1](#). If you want to save information about the linear predictor `mu` you can define it in the `transformed parameters` block ([Section 2.5](#)).

Code 10.2 Normal linear model, in Python using Stan, for assessing the relationship between central black hole mass and bulge velocity dispersion.

```
=====
import numpy as np
import pandas as pd
import pystan

path_to_data = '~/data/Section_10p1/M_sigma.csv
```

```

# Read data
data_frame = dict(pd.read_csv(path_to_data))

# Prepare data for Stan
data = {}
data['obsx'] = np.array(data_frame['obsx'])
data['errx'] = np.array(data_frame['errx'])
data['obsy'] = np.array(data_frame['obsy'])
data['erry'] = np.array(data_frame['erry'])
data['N'] = len(data['obsx'])

# Stan Gaussian model
stan_code=""""
data{
    int<lower=0> N;                      # number of data points
    vector[N] obsx;                       # obs velocity dispersion
    vector<lower=0>[N] errx;              # errors in velocity dispersion measurements
    vector[N] obsy;                       # obs black hole mass
    vector<lower=0>[N] erry;              # errors in black hole mass measurements
}
parameters{
    real alpha;                           # intercept
    real beta;                            # angular coefficient
    real<lower=0> epsilon;               # scatter around true black hole mass
    vector[N] x;                         # true velocity dispersion
    vector[N] y;                         # true black hole mass
}
model{
    # likelihood and priors
    alpha ~ normal(0, 1000);
    beta ~ normal(0, 1000);
    epsilon ~ gamma(0.001, 0.001);

    for (i in 1:N){
        x[i] ~ normal(0, 1000);
        y[i] ~ normal(0, 1000);
    }

    obsx ~ normal(x, errx);
    y ~ normal(alpha + beta * x, epsilon);
    obsy ~ normal(y, erry);
}
"""
# Run mcmc
fit = pystan.stan(model_code=stan_code, data=data, iter=15000, chains=3,
                    warmup=5000, thin=10, n_jobs=3)
=====#
# Output
nlines = 8                                # number of lines in screen output

output = str(fit).split('\n')
for item in output[:nlines]:
    print(item)

      mean   se_mean     sd   2.5%   25%   50%   75% 97.5% n_eff Rhat
alpha     8.35  1.8e-3  0.06  8.24  8.31  8.35  8.39  8.46 995.0  1.0
beta      4.45  0.01    0.34  3.77  4.22  4.45  4.68  5.1  977.0  1.0
epsilon    0.29  1.9e-3  0.06  0.18  0.25  0.28  0.32  0.4  948.0  1.0

```

10.2 Gaussian Mixed Models, Type Ia Supernovae, and Hubble Residuals

We now turn to an astronomical application of a simple mixed model, similar to those described in Chapter 8. Herein we investigate how the relationship between type Ia supernovae (SNe Ia) host galaxy masses and their Hubble residuals (Hrs) changes for spectroscopically and photometrically classified samples.

Type Ia supernovae have been the focus of a great debate within the astronomical community

since they provided the first clues to the current accelerated expansion of the Universe ([Perlmutter et al., 1999](#); [Riess et al., 1998](#)) and, consequently, to the existence of dark energy. This was only possible because SNe Ia remain fairly similar through cosmic history, allowing us to use them as standardizable candles for distance measurements in cosmological scales (see Section 10.11 for a deeper discussion on how to use SNe Ia for cosmological parameter inference).

In order to use SNe Ia data for cosmology they must go through a standardization procedure which corrects for intrinsic variations in color and light-curve shape (stretch).⁷ This data treatment successfully accounts for most SNe Ia light-curve variability. However, a 1σ variability of ≈ 0.1 magnitude still remains, which translates into a 5% uncertainty in distance. In this context, understanding which host galaxy characteristics correlate with Hubble residuals (the difference between the distance modulus after standardization and that predicted from the best-fit cosmological model) can significantly improve the cosmological parameter constraints from SNe Ia. Moreover, different classification methods (spectroscopic or photometric) can introduce biases in the host galaxy–SNe relationship, which should also be taken into account.

[Wolf et al. \(2016\)](#) approached this problem by using a mixed sample of photometrically classified and spectroscopically confirmed SNe Ia to study the relationship between host galaxy and SNe properties, focusing primarily on correlations with Hubble residuals. As a demonstration of a normal model with varying intercepts and slopes, we perform a similar analysis in order to probe the correlation between HRs and host galaxy mass for the spectroscopic (Spec-IA) and photometric (Phot-IA) samples.

10.2.1 Data

The data used in our example was presented by [Wolf et al. \(2016\)](#). It consists of $N = 345$ SNe Ia from the *Sloan Digital Sky Survey – Supernova Survey* (SDSS-SN, [Sako et al. 2014](#)), from which 176 SNe were photometrically classified and 169 were spectroscopically confirmed as Ia. Host galaxy characteristics were derived mainly on the basis of spectra from the SDSS-III *Baryon Oscillation Spectroscopic Survey* (BOSS, [Eisenstein et al. 2011](#)).

In what follows we will use only the subset of this sample comprising the host galaxy mass and Hubble residuals and their respective measurement errors.

10.2.2 Statistical Model Formulation

We will consider the HR measurements as realizations of a random variable following an underlying Gaussian distribution with unknown variance and whose mean is linearly related to the host galaxy mass. Thus, the observed response variable is the Hubble residual, $obsy \equiv HR$, with errors ε_y , and the explanatory variable is the host galaxy mass, $obsx \equiv \log(M/M_\odot)$ with measurement errors ε_x ; a variable j acts as a flag identifying the Phot-IA ($j = 1$) and Spec-IA ($j = 2$) samples. The formal statistical model can be then represented as

$$\begin{aligned}
obsy_i &\sim Normal(y_i, \varepsilon_y^2) \\
y_i &\sim Normal(\mu_i, \varepsilon^2) \\
\mu_i &= \beta_{1j} + \beta_{2j}x_i \\
\varepsilon &\sim Gamma(10^{-3}, 10^{-3}) \\
obsx_i &\sim Normal(x_i, \varepsilon_x^2) \\
x_i &\sim Normal(0, 10^3) \\
\beta_{1j} &\sim Normal(\mu_0, \sigma_0^2) \\
\beta_{2j} &\sim Normal(\mu_0, \sigma_0^2) \\
\mu_0 &\sim Normal(0, 1) \\
\sigma_0 &\sim Uniform(0.1, 5) \\
i &= 1, \dots, N \\
j &\in \{1, 2\}
\end{aligned} \tag{10.5}$$

where we have used a non-informative prior for σ and a common normal hyperprior i.e., a prior associated with the parameters of a prior distribution for the parameters β , connected through μ_0 (the mean) and σ_0 (the standard deviation).

10.2.3 Running the Model in R using JAGS

The implementation of the model listed in Equation 10.5 is straightforward for the reader familiar with the synthetic models presented in Chapter 8. The complete R code is shown below.

Code 10.3 Gaussian linear mixed model, in R using JAGS, for modeling the relationship between type Ia supernovae host galaxy mass and Hubble residuals.

```
=====
library(R2jags)

# Data
path_to_data = "~/data/Section_10p2/HR.csv"
dat <- read.csv(path_to_data, header = T)

# Prepare data to JAGS
nobs  = nrow(dat)
obsx1 <- dat$LogMass
errx1 <- dat$e_LogMass
obsy  <- dat$HR
erry  <- dat$e_HR
type  <- as.numeric(dat$type)      # convert class to numeric flag 1 or 2

jags_data <- list(
  obsx1 = obsx1,
  obsy  = obsy,
  errx1 = errx1,
  erry  = erry,
```

```

K = 2,
N = nobs,
type = type)

# Fit
NORM_errors <-" model{
  tau0~dunif(1e-1,5)
  mu0~dnorm(0,1)

  # Diffuse normal priors for predictors
  for (j in 1:2){
    for (i in 1:K) {
      beta[i,j] ~ dnorm(mu0, tau0)
    }
  }

  # Gamma prior for standard deviation
  tau ~ dgamma(1e-3, 1e-3)          # precision
  sigma <- 1 / sqrt(tau)           # standard deviation

  # Diffuse normal priors for true x
  for (i in 1:N){
    x1[i]~dnorm(0,1e-3)
  }
  # Likelihood function
  for (i in 1:N){
    obsy[i] ~ dnorm(y[i],pow(erry[i],-2))
    y[i] ~ dnorm(mu[i],tau)
    obsx1[i] ~ dnorm(x1[i],pow(errx1[i],-2))
    mu[i] <- beta[1,type[i]] + beta[2,type[i]] * x1[i]
  }
}

inits <- function () {
  list(beta = matrix(rnorm(4, 0, 0.01),ncol = 2))
}

params0 <- c("beta", "sigma")

# Run MCMC
NORM <- jags(
  data = jags_data,
  inits = inits,
  parameters = params0,
  model = textConnection(NORM_errors),
  n.chains = 3,
  n.iter = 40000,
  n.thin = 1,
  n.burnin = 15000)

# Output
print(NORM,justify = "left", digits=3)
=====
```

This will produce the following screen output:

	mu.vect	sd.vect	2.5%	25%	50%	75%	97.5%	Rhat	n.eff
beta[1,1]	0.894	0.224	0.500	0.729	0.877	1.039	1.369	1.035	63
beta[2,1]	-0.087	0.021	-0.132	-0.100	-0.085	-0.071	-0.049	1.038	60
beta[1,2]	0.245	0.179	-0.119	0.128	0.252	0.366	0.587	1.027	97
beta[2,2]	-0.024	0.017	-0.056	-0.035	-0.024	-0.013	0.011	1.027	96
sigma	0.120	0.009	0.103	0.114	0.120	0.126	0.138	1.002	1900

where $\{\beta_{1,1}, \beta_{2,1}\}$ are the intercept and slope for the photometric sample and $\{\beta_{1,2}, \beta_{2,2}\}$ the same two quantities for the spectroscopic sample.

Table 10.1 gives our results along with those reported by [Wolf et al. \(2016\)](#). In order to illustrate the effect of adding the hyperpriors, we also show in this table results obtained without taking them into account⁸ (this means considering $\beta \sim Normal(0, 10^3)$ as a prior for all β s). From this table

it is possible to recognize two main characteristics: the significant difference between slope and intercept values for the two different subsamples and the influence of the common hyperprior on the posterior means (although within the same subsample all results agree within 1σ credible intervals).

Table 10.1 For comparison, our results (in R using JAGS) and those reported by [Wolf et al. \(2016\)](#) for the correlation between the Hubble residuals and the host galaxy mass.

		Spec-Ia	Photo-Ia
Intercept	Wolf et al. (2016)	0.287 ± 0.188	1.042 ± 0.270
	GLMM – no hyperpriors	0.275 ± 0.200	0.939 ± 0.260
	GLMM – with hyperpriors	0.245 ± 0.179	0.894 ± 0.224
Slope	Wolf et al. (2016)	-0.028 ± 0.018	-0.101 ± 0.026
	GLMM – no hyperpriors	-0.027 ± 0.019	-0.091 ± 0.025
	GLMM – with hyperpriors	-0.024 ± 0.017	-0.087 ± 0.021

The differences between the results from the photo and the spec subsamples show that they have distinct behaviors, in other words, that we are actually dealing with separate populations. Nevertheless, using the hyperpriors (and considering that, although recognizable differences exist, these objects still have a lot in common) allows us to constrain further the parameters for both populations (in order to achieve smaller scatter in both the spectroscopic and photometric cases).

The same results are illustrated in Figure 10.3. The upper panel shows the prediction intervals in the parameter space formed by the Hubble residuals as a function of host galaxy mass for the two separate samples, and the lower panel highlights the difference between slope and intercept values. In both panels we can clearly see the different trends followed by each sample.

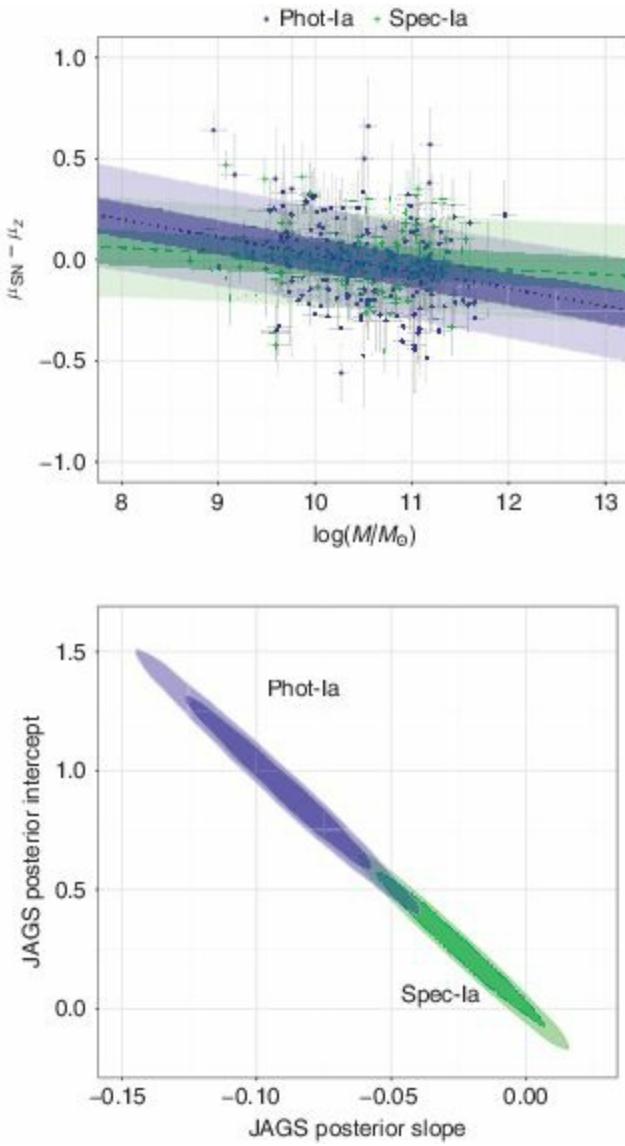


Figure 10.3 Upper panel: Hubble residuals ($HR = \mu_{SN} - \mu_z$) as a function of the host galaxy mass ($\log(M/M_\odot)$) for the PM sample from [Wolf et al. \(2016\)](#). The shaded areas represent 50% (darker) and 95% (lighter) prediction intervals for the spectroscopic (lower) and photometric (upper) samples. Lower panel: Contour intervals showing the 68% (darker) and 95% (lighter) credible intervals of the Spec-Ia and Phot-Ia JAGS posterior distributions for the HR-mass relation.

[Wolf et al. \(2016\)](#), encouraging further analysis into this result, point out, among other reasons, the necessity to have a better understanding of type Ia SNe progenitor systems and a more reliable photometric classification pipeline for type Ia SNe ([Ishida and de Souza, 2013](#); [Kessler et al., 2010](#)).

10.2.4 Running the Model in Python using Stan

The implementation in Python using Stan is also standard. The main difference between Codes 10.3 and 10.4 comes from the fact that, in Stan, the variable definition also defines its domain. Thus, we will follow the guidelines from [Stan \(2016\)](#), adopting a half-normal prior for the shared

hyperparameter `sig0` and using a weak informative prior over `beta`.⁹ These choices were made to illustrate important points the reader should consider when choosing priors; however, as will be made clear shortly, in this example they have no effect on the results. This may not be the case for more complicated scenarios.

In order to facilitate comparison, we designed the `beta` matrix in such a way that the sequence of output parameter values is the same as that appearing in Code 10.3.

Code 10.4 Gaussian linear mixed model, in Python using Stan, for modeling the relationship between type Ia supernovae host galaxy mass and Hubble residuals.

```
=====
import numpy as np
import pandas as pd
import pystan

# Data
path_to_data = '~/data/Section_10p2/HR.csv'
data_frame = dict(pd.read_csv(path_to_data))

# Prepare data for Stan
data = {}
data['obsx'] = np.array(data_frame['LogMass'])
data['errx'] = np.array(data_frame['e_LogMass'])
data['obsy'] = np.array(data_frame['HR'])
data['erry'] = np.array(data_frame['e_HR'])
data['type'] = np.array([1 if item == 'P' else 0
                       for item in data_frame['Type']])
data['N'] = len(data['obsx'])
data['K'] = 2                      # number of distinct populations
data['L'] = 2                      # number of coefficients

# Fit
stan_code=""""
data{
    int<lower=0> N;                  # number of data points
    int<lower=0> K;                  # number of distinct populations
    int<lower=0> L;                  # number of coefficients
    vector[N] obsx;                 # obs host galaxy mass
    vector<lower=0>[N] errx;         # errors in host mass measurements
    vector[N] obsy;                 # obs Hubble Residual
    vector<lower=0>[N] erry;         # errors in Hubble Residual measurements
    vector[N] type;                 # flag for spec/photo sample
}
parameters{
    matrix[K,L] beta;               # linear predictor coefficients
    real<lower=0> sigma;             # scatter around true black hole mass
    vector[N] x;                   # true host galaxy mass
    vector[N] y;                   # true Hubble Residuals
    real<lower=0> sig0;              # scatter for shared hyperprior on beta
    real mu0;                      # mean for shared hyperprior on beta
}
transformed parameters{
    vector[N] mu;                  # linear predictor

    for (i in 1:N) {
        if (type[i] == type[1]) mu[i] <- beta[1,1] + beta[2,1] * x[i];
        else mu[i] = beta[1,2] + beta[2,2] * x[i];
    }
}
model{

    # Shared hyperprior
    mu0 ~ normal(0, 1);
    sig0 ~ normal(0, 5);
    for (i in 1:K){
        for (j in 1:L) beta[i,j] ~ normal(mu0, sig0);
    }

    # Priors and likelihood
    obsx ~ normal(x, errx);
    x ~ normal(0, 100);
}
```

```

y ~ normal(mu, sigma);
sigma ~ gamma(0.5,0.5);
obsy ~ normal(y, erry);
}
"""

# Run mcmc
fit = pystan.stan(model_code=stan_code, data=data, iter=40000, chains=3,
                    warmup=15000, thin=1, n_jobs=3)

# Output
nlines = 10                                # number of lines in screen output

output = str(fit).split('\n')
for item in output[:nlines]:
    print(item)
=====
      mean   se_mean     sd   2.5%   25%   50%   75% 97.5%  n_eff   Rhat
beta[0,0]  0.83   2.1e-3   0.27   0.27   0.65   0.83   1.01   1.34  15974   1.0
beta[1,0] -0.08   2.0e-4   0.03  -0.13  -0.1  -0.08  -0.06  -0.03  15975   1.0
beta[0,1]  0.24   1.2e-3   0.18  -0.11  -0.12  -0.24  -0.36   0.6  22547   1.0
beta[1,1] -0.02   1.2e-4   0.02  -0.06  -0.03  -0.02  -0.01  -0.01  22507   1.0
sigma      0.12   7.6e-5  9.0e-3   0.1   0.11   0.12   0.13   0.14  14034   1.0

```

10.3 Multivariate Normal Mixed Model and Early-Type Contact Binaries

We continue to explore the potential of Gaussian model extensions by presenting an astronomical example which requires the combination of two features described in previous chapters: multivariate normal (Section 4.2) and mixed models (Chapter 8). As a case study we will investigate the period–luminosity–color (PLC) relation in early-type contact and near-contact binary stars.

The star types O, B, and A are hot and massive objects traditionally called “early type”.¹⁰ It is believed that most such stars are born as part of a binary system, being close enough to have a mass exchange interaction with their companion (Figure 10.4 shows the most massive genuinely contact binary system known to date, reported by [Almeida et al. \(2015\)](#)). This interaction will have significant impact in the posterior development of both stars and, given their high frequency of occurrence, play an important role in stellar population evolution.

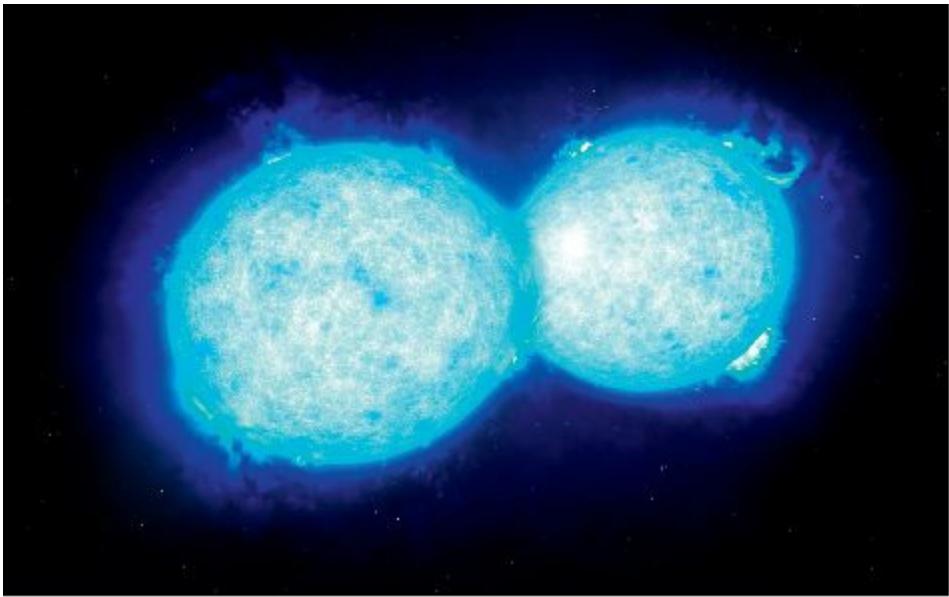


Figure 10.4 Artist’s impression of VFTS 352, the most massive and earliest spectral type genuinely-contact binary system known to date (Almeida *et al.*, 2015). Image credits: ESO /L. Calçada.

A PLC relation is known to exist in close binary stars owing to an observed correlation between the radius of the orbit and the mass of the stars forming the pair (Machida *et al.*, 2008). Our goal is to determine whether subpopulations of genuinely-contact and close-contact systems follow the same underlying relationship (Pawlak, 2016). The mathematical formulation of the PLC relation (e.g. Pawlak, 2016; Rucinski, 2004) is usually stated as

$$M_V = \beta_0 + \beta_1 \log(P) + \beta_2(V - I)_0, \quad (10.6)$$

where M_V is the absolute V -band magnitude, P denotes the period and $(V - I)_0$ is the color between the filters V and I .

10.3.1 Data

The data set we shall use is composed of a sample of $N = 64$ eclipsing binaries classified as near-contact (NC, stars which are close but not in thermal contact), and genuinely-contact (GC, where the two stars are in thermal equilibrium). This sample was built by Pawlak (2016)¹¹ from phase III of the *Optical Gravitational Lensing Experiment* (OGLE-III) catalog of eclipsing binaries in the Large Magellanic Cloud (Graczyk *et al.*, 2011).

10.3.2 The Statistical Model Formulation

The problem stated above and formulated in Equation 10.6 requires a multivariate model with two explanatory variables; the period $P \equiv \log P$ and the color $c \equiv (V - I)_0$, and one response variable, the magnitude $M \equiv M_V$ in the V -band. Moreover, as we wish to take into account the existence of different populations, namely NC ($j = 1$) and GC ($j = 2$), which might follow

distinct relations, we will need an extra categorical variable to identify the object classification, t , and two sets of $K = 3$ coefficients; hence β will be a $K \times J$ matrix. The intrinsic scatter around the response variable M_V in Equation 10.6 is called σ . The complete statistical model can be described as

$$\begin{aligned}
M_{V,i} &\sim \text{Normal}(\mu_i, \sigma_t^2) \\
\mu_i &= \beta_{k=1,t_i} + \beta_{k=2,t_i} P_i + \beta_{k=3,t_i} c_i \\
t_i &= \begin{cases} 1 & \text{if GC} \\ 2 & \text{if NC} \end{cases} \\
\beta_{kj} &\sim \text{Normal}(\mu_0, \sigma_0^2) \\
\mu_0 &\sim \text{Normal}(0, 10^3) \\
\sigma_0 &\sim \text{Gamma}(0.001, 0.001) \\
i &= 1, \dots, N \\
k &= 1, \dots, K \\
j &\in [1, 2]
\end{aligned} \tag{10.7}$$

Notice that, for each data point i , the term t_i is used as an index of linear predictor coefficients, distinguishing between each class of the binary system. This means that the set of β parameters to be used will be dependent on the classification of each data point. It is also important to emphasize that, by employing a hierarchical Bayesian model for the intercepts and slopes, we allow the model to borrow strength across populations. That is, while the model acknowledges that each separate population (NC or GC) has its own parameters, it allows the whole population to contribute to the inference of individual parameters (represented by the slopes and intercepts). This happens via their joint influence on the posterior estimates of the unknown hyperparameters μ_0 and σ_0 .

Although we have applied the concept for a simple model (only two classes of object), the approach can be easily expanded for a more complex situation with dozens or hundreds of classes. It reflects the hypothesis that, although there might be differences among classes, they do share a global similarity (i.e., they are all close binary systems).

10.3.3 Running the Model in R using JAGS

In what follows, we show how to implement the model in JAGS.

Code 10.5 Multivariate normal model in R using JAGS for accessing the relationship between period, luminosity, and color in early-type contact binaries.

```
=====
library(R2jags)

# Data
#Read data
PLC <- read.csv("~/data/Section_10p3/PLC.csv", header = T)

# Prepare data for JAGS
nobs  = nrow(PLC)                                # number of data points
x1    <- PLC$logP                               # log period
x2    <- PLC$V_I                                 # V-I color
y     <- PLC$M_V                                 # V magnitude
```

```

type  <- as.numeric(PLC$type)           # type NC/GC
X     <- model.matrix(~ 1 + x1+x2)       # covariate matrix
K     <- ncol(X)                      # number of covariates per type

jags_data <- list(Y = y,
                  X = X,
                  K = K,
                  type = type,
                  N = nobs)

# Fit
NORM <- "model{

# Shared hyperprior
tau0 ~ dgamma(0.001, 0.001)
mu0 ~ dnorm(0, 1e-3)

# Diffuse normal priors for predictors
for(j in 1:2){
  for (i in 1:K) {
    beta[i,j] ~ dnorm(mu0, tau0)
  }
}
# Uniform prior for standard deviation
for(i in 1:2) {

  tau[i] <- pow(sigma[i], -2)#precision
  sigma[i] ~ dgamma(1e-3, 1e-3)#standard deviation
}
# Likelihood function
for (i in 1:N){
  Y[i]~dnorm(mu[i],tau[type[i]])
  mu[i] <- eta[i]
  eta[i] <- beta[1, type[i]] * X[i, 1] + beta[2, type[i]] * X[i, 2] +
    beta[3, type[i]] * X[i, 3]
}
}

# Determine initial values
inits <- function () {
  list(beta = matrix(rnorm(6,0, 0.01),ncol=2))
}

# Identify parameters
params <- c("beta", "sigma")

# Fit
jagsfit <- jags(data      = jags_data,
                  inits     = inits,
                  parameters = params,
                  model     = textConnection(NORM),
                  n.chains = 3,
                  n.iter   = 5000,
                  n.thin   = 1,
                  n.burnin = 2500)

## Output
print(jagsfit,justify = "left", digits=2)
=====

          mu.vect sd.vect 2.5% 97.5% Rhat n.eff
beta[1,1]  -1.01  0.27 -1.55 -0.49    1  7500
beta[2,1]  -3.29  0.95 -5.15 -1.37    1  7500
beta[3,1]   7.22  1.28  4.61  9.68    1  7500
beta[1,2]  -0.41  0.15 -0.71 -0.11    1  4200
beta[2,2]  -3.19  0.58 -4.33 -2.00    1  7500
beta[3,2]   8.50  0.82  6.87 10.08    1  4100
sigma[1]    0.62  0.09  0.47  0.82    1   920
sigma[2]    0.43  0.05  0.34  0.55    1  5200
deviance   91.82  4.38 85.47 102.28    1  7500

pD = 9.6 and DIC = 101.4

```

According to the notation adopted in Code 10.5, the first column of the `beta` matrix holds coefficients for genuinely-contact (GC) objects while the second column stores coefficients for the near-contact (NC) subsample.

From these we see that, beyond an overall shift in magnitude (due to incompatible posteriors for $\text{beta}[1,1]$ and $\text{beta}[2,1]$), the two populations present a very similar dependence on period (very close posteriors for $\text{beta}[2,1]$ and $\text{beta}[2,2]$) and only marginally agree in their dependence on color (overlapping posteriors for $\text{beta}[3,1]$ and $\text{beta}[3,2]$). Moreover, comparing the numbers in the `sd.vect` column we see that the genuinely-contact systems present a larger scatter than the near-contact systems. Figure 10.5 illustrates the positioning of the two mean models in PLC space.

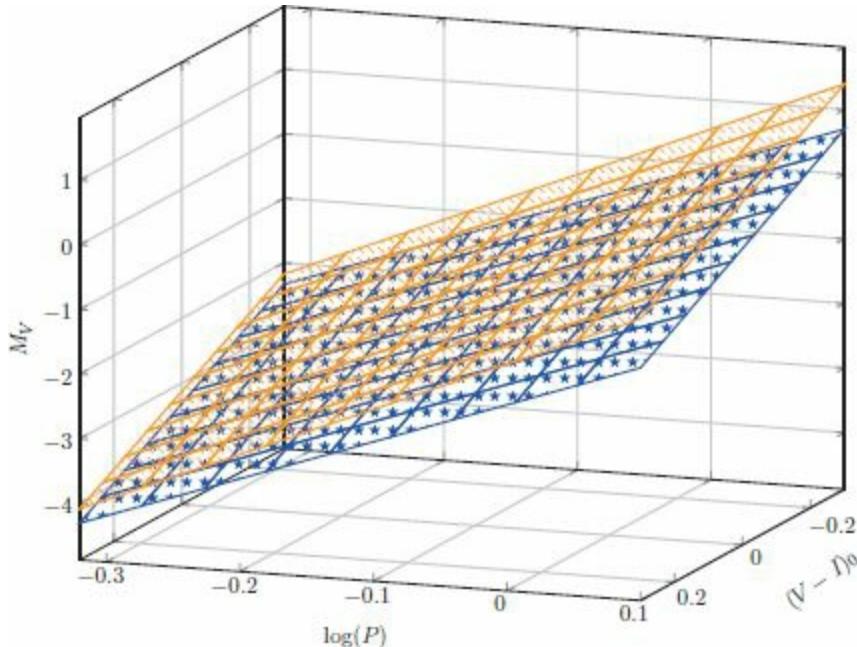


Figure 10.5 Period–luminosity–color relation obtained for the sample of 64 near early-type binary systems from the Large Magellanic Cloud (Pawlak, 2016). The sample is color-coded for genuinely-contact (upper) and near-contact (lower) binaries.

The results presented above are consistent with the least-squares fit employed by Pawlak (2016), and are presented in Table 10.2. This similarity is expected given the quality of the data set (a subset of 64 systems having well-covered light curves and low photometric noise) and the lack of informative priors in our analysis.

Table 10.2 For comparison, the PLC relation coefficients reported by Pawlak (2016) and our results using GLMM.

	Pawlak 2016			
	β_1	β_2	β_2	σ
GC	-0.97 ± 0.26	-3.47 ± 0.87	7.57 ± 1.21	0.55
NC	-0.40 ± 0.15	-0.40 ± 0.15	7.57 ± 1.21	0.40
GLMM				
	β_1	β_2	β_2	σ
GC	-0.40 ± 0.15	-0.97 ± 0.26	7.22 ± 1.28	0.62 ± 0.09

10.3.4 Running the Model in Python using Stan

Code 10.6 Multivariate Gaussian mixed model in Python, using Stan, for accessing the relationship between luminosity, period, and color in early-type contact binaries.

```
=====
import numpy as np
import pandas as pd
import pystan
import statsmodels.api as sm

# Data
path_to_data = '~data/Section_10p3/PLC.csv'

# Read data
data_frame = dict(pd.read_csv(path_to_data))

# Prepare data for Stan
data = {}
data['x1'] = np.array(data_frame['logP'])
data['x2'] = np.array(data_frame['V_I'])
data['y'] = np.array(data_frame['M_V'])
data['nobs'] = len(data['x1'])
data['type'] = np.array([1 if item == data_frame['type'][0] else 0
                       for item in data_frame['type']])
data['M'] = 3
data['K'] = data['M'] - 1

# Fit
# Stan Multivariate Gaussian
stan_code=""""
data{
    int<lower=0> nobs;                      # number of data points
    int<lower=1> M;                          # number of linear predictor coefficients
    int<lower=1> K;                          # number of distinct populations
    vector[nobs] x1;                         # obs log period
    vector[nobs] x2;                         # obs color V-I
    vector[nobs] y;                          # obs luminosity
    int type[nobs];                         # system type (near or genuine contact)
}
parameters{
    matrix[M,K] beta;                      # linear predictor coefficients
    real<lower=0> sigma[K];                # scatter around linear predictor
    real mu0;
    real sigma0;
}
transformed parameters{
    vector[nobs] mu;                      # linear predictor

    for (i in 1:nobs) {
        if (type[i] == type[1])
            mu[i] = beta[1,2] + beta[2,2] * x1[i] + beta[3,2] * x2[i];
        else mu[i] = beta[1,1] + beta[2,1] * x1[i] + beta[3,1] * x2[i];
    }
}
model{
    # priors and likelihood
    mu0 ~ normal(0, 100);
    sigma0 ~ gamma(0.001, 0.001);

    for (i in 1:K) {
        sigma[i] ~ gamma(0.001, 0.001);
        for (j in 1:M) beta[j,i] ~ normal(mu0,sigma0);
    }

    for (i in 1:nobs){
        if (type[i] == type[1]) y[i] ~ normal(mu[i], sigma[2]);
        else y[i] ~ normal(mu[i], sigma[1]);
    }
}"""
# Run mcmc
```

```

fit = pystan.stan(model_code=stan_code, data=data, iter=5000, chains=3,
                   warmup=2500, thin=1, n_jobs=3)

# Output
nlines = 13                                     # number of lines in screen output

output = str(fit).split('\n')
for item in output[:nlines]:
    print(item)
=====
      mean se_mean   sd  2.5%  25%  50%  75% 97.5%  n_eff Rhat
beta[0,0] -1.01  6.5e-3  0.27 -1.55 -1.19 -1.01 -0.83 -0.48 1781.0  1.0
beta[1,0] -3.31  0.02   0.98 -5.25 -3.95 -3.31 -2.67 -1.34 2082.0  1.0
beta[2,0]  7.21  0.03   1.28  4.63  6.39  7.23  8.08  9.64 1975.0  1.0
beta[0,1] -0.42  3.8e-3  0.16 -0.74 -0.52 -0.42 -0.31 -0.1  1836.0  1.0
beta[1,1] -3.2   0.01   0.58 -4.37 -3.58 -3.2  -2.82 -2.08 2156.0  1.0
beta[2,1]  8.46  0.02   0.83  6.83  7.92  8.47  9.03 10.07 1828.0  1.0
sigma[0]   0.62  2.1e-3  0.09  0.47  0.55  0.6  0.67  0.82 1990.0  1.0
sigma[1]   0.42  1.2e-3  0.05  0.34  0.39  0.42  0.46  0.55 2138.0  1.0

```

10.4 Lognormal Distribution and the Initial Mass Function

In this section we show how to use a given probability function to fit a distribution. This means that we are not aiming to build a regression model. Instead, we wish to characterize the underlying probability distribution driving the behavior of a measured quantity. This is exactly the case for the long-standing problem of determining the stellar initial mass function (IMF). We will show a Bayesian approach to the strategy, presented by [Zaninetti \(2013\)](#), by fitting a lognormal distribution (Section 5.2.1) to stellar masses of the star cluster NGC 6611 ([Oliveira et al., 2005](#)).

The IMF determines the probability distribution function for the mass at which a star enters the main sequence. It regulates the relative abundance of massive versus low-mass stars for each stellar generation and influences most observable properties of stellar populations and galaxies ([Bastian et al., 2010](#)). It is also a required input for semi-analytical models of galaxy evolution ([Fontanot, 2014](#)). Given such a crucial role in shaping the stellar population, the ultimate goal of any theory of star formation is to predict the stellar IMF from first principles. Currently, there are a few empirical alternatives for describing the IMF, mostly based on power law functions for different ranges of masses ([Bastian et al., 2010](#)).

The IMF is usually considered to be universal, assuming the shape of a Salpeter power law, $dN \propto M_{\star}^{-2.35} dM$ ([Salpeter, 1955](#)), for stars with masses $M_{\star} > 0.5M_{\odot}$ ([Kroupa, 2001](#)), where dN is the number of stars per bin of stellar mass dM_{\star} . Another common option to fit the IMF is to use a lognormal distribution in order to cover the mass function down to the brown dwarf regime ($M \leq 0.1M_{\odot}$, [Chabrier, 2003](#)). The goal of this section is to demonstrate how to fit a lognormal distribution (Section 5.2.1) to a data set of stellar mass measurements. In this special case there is no explanatory variable.

10.4.1 Data

We will use the photometric observations of stellar masses from NGC 6611, the young and massive cluster that ionizes the Eagle Nebula ([Oliveira et al., 2005](#)). The data are from 208 stars for which

mass measurements are available.¹² We chose this particular data set to allow a simple comparison with the analysis performed by [Zaninetti \(2013\)](#), who made a comprehensive comparison of different probability distributions for fitting the IMF.

10.4.2 Statistical Model Formulation

In the following, we show how to write a model to fit a Bayesian histogram using a lognormal distribution with location and scale parameters μ and σ respectively and response variable given by the stellar mass $M \equiv M_*$. The complete lognormal model can be expressed as

$$\begin{aligned} M_i &\sim \text{LogNormal}(\mu, \sigma^2) \\ \mu &\sim \text{Normal}(0, 10^3) \\ \sigma^2 &\sim \text{Gamma}(10^{-3}, 10^{-3}) \\ i &= 1, \dots, N \end{aligned} \tag{10.8}$$

where we have applied a non-informative Gaussian (gamma) prior for the location (scale) parameters.

10.4.3 Running the Model in R using JAGS

The implementation in JAGS follows:

Code 10.7 Lognormal model in R using JAGS to describe the initial mass function (IMF).

```
=====
library(R2jags)

# Data
path_to_data = "~/data/Section_10p4/NGC6611.csv"

# Read data
IMF <- read.table(path_to_data, header = T)

N <- nrow(IMF)
x <- IMF$Mass

# Prepare data to JAGS
jags_data <- list(x = x,
                   N = N           # sample size
)

# Fit
LNORM <-
model{
  # Uniform prior for standard deviation
  tau   <- pow(sigma, -2)      # precision
  sigma ~ dunif(0, 100)        # standard deviation
  mu    ~ dnorm(0, 1e-3)

  # Likelihood function
  for (i in 1:N){
    x[i] ~ dlnorm(mu, tau)
  }
}"
```

```

# Identify parameters
params <- c("mu", "sigma")

# Run mcmc
LN <- jags(
  data      = jags_data,
  parameters = params,
  model     = textConnection(LNORM),
  n.chains  = 3,
  n.iter    = 5000,
  n.thin    = 1,
  n.burnin  = 2500)

# Output
print(LN, justify = "left", intervals=c(0.025,0.975), digits=2)
=====
      mu.vect sd.vect 2.5% 97.5% Rhat n.eff
mu      -1.26   0.07 -1.39 -1.12    1  6600
sigma    1.03   0.05  0.94  1.14    1  1100
deviance 81.20   1.96 79.27 86.49    1  1100

pD = 1.9 and DIC = 83.1

```

The fitted values can be compared to those from Zaninetti (2013, Table 2), who found $\mu = -1.258$ and $\sigma = 1.029$. Figure 10.6 shows the lognormal distribution fitted to the data set together with the 50% and 95% credible intervals around the mean. The full code to reproduce this plot is given below. Note that the code makes use of the function `jagsresults`, which makes it easier to extract the information from the Markov chain. This function is found in the package `jagstools` and is available on GitHub. It can be installed with the following lines:

```
require(devtools)
install_github("johnbaums/jagstools")
```

Code 10.8 Plotting routine, in R, for Figure 10.6.

```

=====

require(jagstools)
require(ggplot2)

# Create new data for prediction
M = 750
xx = seq(from = 0.75*min(x),
          to = 1.05*max(x),
          length.out = M)

# Extract results
mx <- jagsresults(x=LN, params=c('mu'))
sigmax <- jagsresults(x=LN, params=c('sigma'))

# Estimate values for the Lognormal PDF
ymean <- dlnorm(xx,meanlog=mx[,"50%"],sdlog=sigmax[,"50%"])
ylwr1 <- dlnorm(xx,meanlog=mx[,"25%"],sdlog=sigmax[,"25%"])
ylwr2 <- dlnorm(xx,meanlog=mx[,"2.5%"],sdlog=sigmax[,"2.5%"])
yupr1 <- dlnorm(xx,meanlog=mx[,"75%"],sdlog=sigmax[,"75%"])
yupr2 <- dlnorm(xx,meanlog=mx[,"97.5%"],sdlog=sigmax[,"97.5%"])

# Create a data.frame for ggplot2
gdata <- data.frame(x=xx, mean = ymean,lwr1=ylwr1 ,lwr2=ylwr2,upr1=yupr1,
                     upr2=yupr2)

ggplot(gdata,aes(x=xx))+
  geom_histogram(data=IMF,aes(x=Mass,y = ..density..),
                 colour="red",fill="gray99",size=1,binwidth = 0.075,
                 linetype="dashed")+
  geom_ribbon(aes(x=xx,ymin=lwr1, ymax=upr1,y=NULL),
              alpha=0.45, fill=c("#00526D"),show.legend=FALSE) +
  geom_ribbon(aes(x=xx,ymin=lwr2, ymax=upr2,y=NULL),
              alpha=0.35, fill = c("#00A3DB"),show.legend=FALSE) +
  geom_line(aes(x=xx,y=mean),colour="gray25",
            linetype="dashed",size=0.75,
            show.legend=FALSE) +
  ylab("Density")+
  xlab(expression(M/M[\u0298]))+

```

```

theme_bw() +
theme(legend.background = element_rect(fill = "white"),
      legend.key = element_rect(fill = "white",color = "white"),
      plot.background = element_rect(fill = "white"),
      legend.position = "top",
      axis.title.y = element_text(vjust = 0.1,margin = margin(0,10,0,0)),
      axis.title.x = element_text(vjust = -0.25),
      text = element_text(size = 25))
=====
```

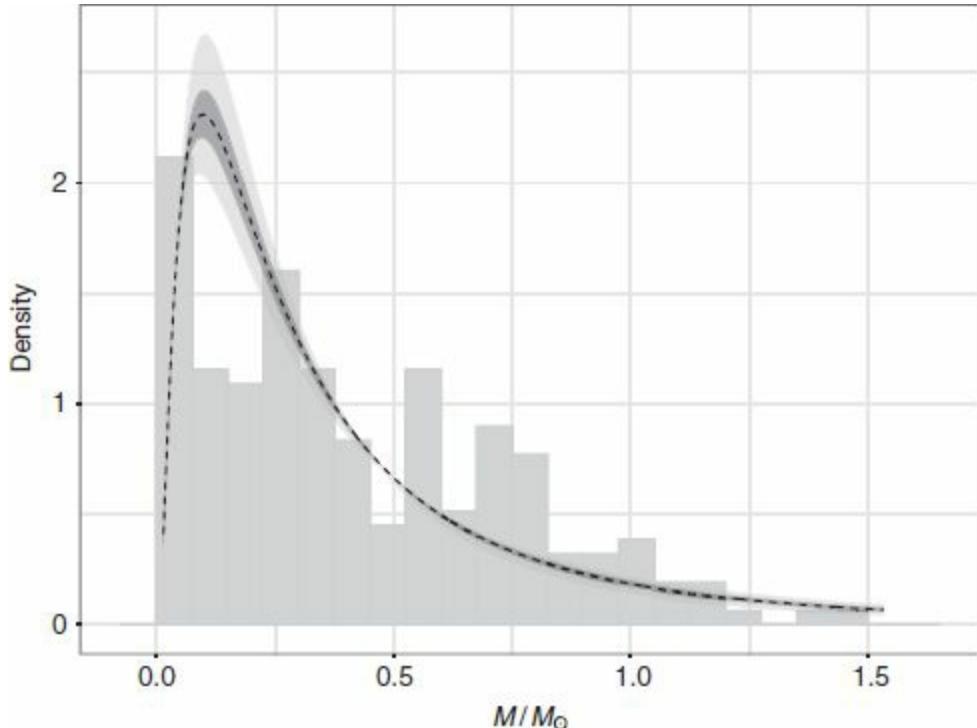


Figure 10.6 Histogram of mass distribution for the NGC 6611 cluster data (208 stars) with a superposed lognormal distribution. The dashed line shows the final mean model and the shaded areas correspond to 50% (darker) and 95% (lighter) credible intervals.

10.4.4 Running the Model in Python using Stan

The implementation in Python using Stan is also straightforward. In the code snippet below we include the commands for generating the posteriors and chain plots, although we do not show them here (the reader using these routines should obtain plots similar to those shown in Figure 3.9.)

Code 10.9 Lognormal model in Python using Stan to describe the initial mass function (IMF).

```

=====
import numpy as np
import pandas as pd
import pylab as plt
import pystan
import statsmodels.api as sm

# Data
path_to_data = '~/data/Section_10p4/NGC6611.csv'

# Read data
data_frame = dict(pd.read_csv(path_to_data))
# Prepare data for Stan
```

```

data = []
data['X'] = data_frame['Mass']
data['nobs'] = data['X'].shape[0]

# Fit
# Stan model
stan_code=""""
data{
    int<lower=0> nobs;           # number of data points
    vector[nobs] X;              # stellar mass
}
parameters{
    real mu;                    # mean
    real<lower=0> sigma;        # scatter
}
model{
    # priors and likelihood
    sigma ~ normal(0, 100);
    mu ~ normal(0, 100);

    X ~ lognormal(mu, sigma);
}
"""

# Run mcmc
fit = pyStan.stan(model_code=stan_code, data=data, iter=5000, chains=3,
                    warmup=2500, thin=1, n_jobs=3)

# Output
print(fit)
=====
# plot chains and posteriors
fit.traceplot()
plt.show()

      mean   se_mean     sd   2.5%   25%   50%   75%   97.5%   n_eff   Rhat
mu     -1.25  1.6e-3  0.07  -1.4   -1.3  -1.25  -1.21  -1.11  2007.0    1.0
sigma   1.04  1.1e-3  0.05  0.94   1.0   1.03   1.07   1.14  2069.0    1.0

```

10.5 Beta Model and the Baryon Content of Low Mass Galaxies

Our next example deals with situations where the response variable can take any real value between 0 and 1 (i.e., a fraction). We discussed the beta model in Section 5.2.4 within a simulated framework. Here we will show how it can be applied directly to model the baryonic gas fraction as a function of galaxy mass.

In trying to compare theoretical predictions of galaxy formation and evolution with observations, one is faced with a difficult situation: how to identify which galaxy properties are driven by internal effects and which are a consequence of interaction with other galaxies in the cluster? Such environmental effects are specially important for low mass galaxies, which are more prone to have their gravitational potential perturbed by very massive neighbors. In this scenario, one possible strategy is to select a sample of isolated galaxies, which are less perturbed by effects such as tidal forces and ram pressure and are better laboratories to study correlations between different galaxy properties without too much environmental interference. This approach was adopted by [Bradford *et al.* \(2015\)](#), who studied the dependence of the baryon fraction in atomic gas, f_{gas} , on other galaxy properties. In this example we will focus in the correlation between f_{gas} and stellar mass ([Bradford *et al.*, 2015](#), Figure 4, left-hand panel).

10.5.1 Data

We will use the data set compiled by Bradford *et al.* (2015), which contains $N = 1715$ galaxies from the NASA-Sloan Atlas¹³ catalog (Blanton *et al.*, 2011). These are considered to be low mass galaxies ($M_{\star} \leq 10^{9.5} M_{\odot}$) in isolation¹⁴ (the projected distance to the nearest neighbor $j = 2$ Mpc). From this we used two columns: the stellar mass, $M = \log(M_{\star}/M_{\odot})$, and the baryon fraction in atomic gas, f_{gas} , defined as

$$f_{\text{gas}} = \frac{M_{\text{gas}}}{M_{\text{gas}} + M_{\star}}, \quad (10.9)$$

where M_{gas} is the atomic-gas mass and M_{\star} is the stellar mass.

10.5.2 The Statistical Model Formulation

In this example, the atomic gas fraction is our response variable f_{gas} and the galaxy stellar mass our explanatory variable M . The data set contains 1715 galaxies and the model holds $K = 2$ linear predictor coefficients. The beta model for this problem can be expressed as follows:

$$\begin{aligned} f_{\text{gas};i} &\sim \text{Beta}(\alpha_i, \beta_i) \\ \alpha_i &= \theta p_i \\ \beta_i &= \theta(1 - p_i) \\ \log\left(\frac{p_i}{1 - p_i}\right) &= \eta_i \\ \eta_i &= \beta_1 + \beta_2 M_i \\ \beta_j &\sim \text{Normal}(0, 10^3) \\ \theta &\sim \text{Gamma}(0.001, 0.001) \\ i &= 1, \dots, N \\ j &= 1, \dots, K \end{aligned} \quad (10.10)$$

where α and β are the shape parameters for the beta distribution.

10.5.3 Running the Model in R using JAGS

Code 10.10 Beta model in R using JAGS, for accessing the relationship between the baryon fraction in atomic gas and galaxy stellar mass.

```
=====
require(R2jags)

# Data
path_to_data = "../data/Section_10p5/f_gas.csv"
```

```

# Read data
Fgas0 <- read.csv(path_to_data, header=T)

# Estimate F_gas
Fgas0$fgas <- Fgas0$M_HI/(Fgas0$M_HI+Fgas0$M_STAR)

# Prepare data to JAGS
N = nrow(Fgas0)

y <- Fgas0$fgas
x <- log(Fgas0$M_STAR, 10)

X <- model.matrix(~ 1 + x)
K <- ncol(X)

beta_data <- list(Y = y,
                     X = X,
                     K = K,
                     N = N)

# Fit
Beta <- "model{
  # Diffuse normal priors for predictors
  for(i in 1:K){
    beta[i] ~ dnorm(0, 1e-4)
  }

  # Diffuse prior for theta
  theta~dgamma(0.01,0.01)

  # Likelihood function
  for (i in 1:N){
    Y[i] ~ dbeta(a[i],b[i])
    a[i] <- theta * pi[i]
    b[i] <- theta * (1-pi[i])
    logit(pi[i]) <- eta[i]
    eta[i] <- inprod(beta[],X[i,])
  }
}

# Define initial values
inits <- function () {
  list(beta = rnorm(2, 0, 0.1),
       theta = runif(1,0,100))
}

# Identify parameters
params <- c("beta","theta")

Beta_fit <- jags(data = beta_data,
                  inits = inits,
                  parameters = params,
                  model = textConnection(Beta),
                  n.thin = 1,
                  n.chains = 3,
                  n.burnin = 5000,
                  n.iter = 7500)

# Output
print(Beta_fit,intervals=c(0.025, 0.975),justify = "left", digits=2)
=====
      mu.vect   sd.vect    2.5%   97.5%   Rhat   n.eff
beta[1]      9.29     0.13     9.00    9.54   1.04    100
beta[2]     -0.98     0.01    -1.00   -0.95   1.03    110
theta        11.71     0.38    10.97   12.45   1.00   2200
deviance   -2331.34    2.18  -2333.77  -2325.69   1.01    340

pD = 2.4 and DIC = -2329.0

```

Figure 10.7 shows the relation between the fraction of atomic gas and the galaxy stellar mass; the dashed line represents the mean relationship and the shaded regions denote the 50% (darker) and 95% (lighter) prediction intervals. From the coefficients, we can estimate that a one-dex¹⁵ variation in M_* decreases the average atomic fraction of the galaxy by a factor $1 - \exp(-0.98) \approx 62.5\%$.

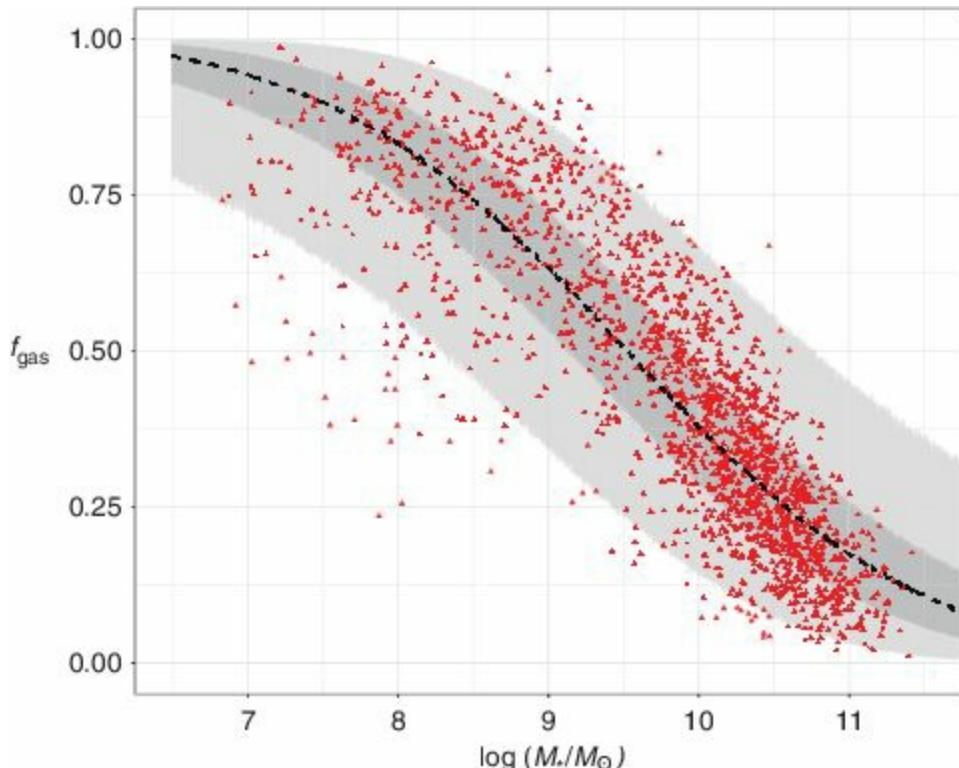


Figure 10.7 Baryon fraction in atomic gas as a function of stellar mass. The dashed line shows the mean posterior fraction and the shaded regions denote the 50% (darker) and 95% (lighter) prediction intervals. The dots are the data points for isolated low mass galaxies shown in Bradford *et al.* (2015, Figure 4, left-hand panel).

10.5.4 Running the Model in Python using Stan

Code 10.11 Beta model in Python using Stan, for accessing the relationship between the fraction of atomic gas and the galaxy stellar mass.

```
=====
import numpy as np
import pandas as pd
import pystan
import statsmodels.api as sm

# Data
path_to_data = '~/data/Section_10p5/f_gas.csv'

# Read data
data_frame = dict(pd.read_csv(path_to_data))

# Built atomic gas fraction
y = np.array([data_frame['M_HI'][i]/
              (data_frame['M_HI'][i] + data_frame['M_STAR'][i])
              for i in range(data_frame['M_STAR'].shape[0])])
```

```

x = np.array([np.log(item) for item in data_frame['M_STAR']])

# Prepare data for Stan
data = {}
data['Y'] = y
data['X'] = sm.add_constant((x.transpose()))
data['nobs'] = data['X'].shape[0]
data['K'] = data['X'].shape[1]

# Fit
# Stan model
stan_code=""""
data{
    int<lower=0> nobs;                                # number of data points
    int<lower=0> K;                                  # number of coefficients
    matrix[nobs, K] X;                             # stellar mass
    real<lower=0, upper=1> Y[nobs];                # atomic gas fraction
}
parameters{
    vector[K] beta;                               # linear predictor coefficients
    real<lower=0> theta;
}
model{
    vector[nobs] pi;
    real a[nobs];
    real b[nobs];

    for (i in 1:nobs){
        pi[i] = inv_logit(X[i] * beta);
        a[i] = theta * pi[i];
        b[i] = theta * (1 - pi[i]);
    }

    # Priors and likelihood
    for (i in 1:K) beta[i] ~ normal(0, 100);
    theta ~ gamma(0.01, 0.01);

    Y ~ beta(a, b);
}
"""
# Run mcmc
fit = pyStan.stan(model_code=stan_code, data=data, iter=7500, chains=3,
                   warmup=5000, thin=1, n_jobs=3)

# Output
print(fit)
=====

      mean   se_mean     sd    2.5%    25%    50%    75% 97.5%   n_eff   Rhat
beta[0]  9.24  5.5e-3  0.18    8.9  9.13  9.24  9.36  9.61  1061.0   1.0
beta[1] -0.42 2.4e-4 7.9e-3 -0.44 -0.43 -0.42 -0.42 -0.41  1068.0   1.0
theta   11.68    0.01    0.4 10.91 11.42 11.68 11.92 12.58  1297.0   1.0

```

10.6 Bernoulli Model and the Fraction of Red Spirals

We now turn to the treatment of binary data. We presented an application of the Bernoulli models using synthetic data in Section 5.3. There we considered a system where the response variable could take two states, success (1) and failure (0), and used a Bernoulli distribution (one particular case of the binomial distribution) to construct our statistical model. In that context, our goal was to determine the parameter of the Bernoulli distribution, p , which represents the chances of getting a success (1) in each realization of the response variable.

Here we will apply this technique to the practical astronomical case of modeling the fraction of red spirals as a function of the bulge size in a given galaxy population. Since the advent of large photometric surveys such as the Sloan Digital Sky Survey (SDSS),¹⁶ astronomers have been using a

well-known correlation between galaxy color and morphology to infer the morphological types of galaxies. According to this relation, spiral galaxies are statistically bluer, and disk-dominated, and hold more star formation than their elliptical, bulge-dominated, counterparts (Mignoli *et al.*, 2009). However significant attention has also been drawn to individuals or groups of galaxies violating this relation (see e.g. Cortese and Hughes, 2009; Mahajan and Raychaudhury, 2009). The Galaxy Zoo project,¹⁷ a combined effort involving professional and citizen scientists, enabled for the first time the direct morphological classification of an unprecedented number of galaxies. This data set revealed the existence of a non-negligible fraction of visually classified spiral galaxies that were redder than usually expected. They are spirals, but with rest-frame colors as red as standard elliptical galaxies.

Masters *et al.* (2010) provided a detailed analysis of the Galaxy Zoo data set with the goal of scrutinizing the physical effects which might lead to the observed fraction of passive red spirals. The exercise presented in this section aims at using the same data set (a selection for blue and red spiral galaxies) to reproduce their Figure 2 from a Bayesian Bernoulli perspective.

Before we dive into this example, it is important to emphasize the crucial difference between the beta model presented in Section 10.5 and the Bernoulli model we are about to discuss. Both involve quantities restricted to the interval $[0, 1]$ but with very different roles. In the beta model, the response variable itself, f_{gas} , is a fraction and thus, by definition, is contained within $[0, 1]$. In Section 10.5, we employed a statistical model as in previous examples, identifying its behavior as a function of the explanatory variable (stellar mass). In the Bernoulli logit case studied here, the response variable is binary (success, red spirals; failure, blue spirals) and our goal is to fit the Bernoulli distribution parameter p . In other words, we aim to model the probability of a success ($p \in [0, 1]$, the probability of being a red spiral) given the explanatory variable (the bulge size). Understanding the differences between these two models (and their potential applications) is essential for the reader to take advantage of the examples shown here in his or her own research.

10.6.1 Data

The Galaxy Zoo clean catalog (Lintott *et al.*, 2008), containing around 900 000 galaxies morphologically classified, was assembled thanks to the work of more than 160 000 volunteers, who visually inspected SDSS images through an online tool. Details on how multiple classifications are converted into the probability that a given galaxy is a spiral, P_{spiral} , are given in Bamford *et al.* (2009); Lintott *et al.* (2008).

The subsample compiled by Masters *et al.* (2010) is presented in two separate tables: one holding 294 red, passive, spirals¹⁸ and the other holding 5139 blue, active, spiral galaxies.¹⁹ We merged these two tables and constructed a single catalog of 5433 lines (galaxies) and two columns (type and fracdev , the fraction of the best-fit light profile from the de Vaucouleurs fit, which is considered a proxy for the bulge size).

10.6.2 The Statistical Model Formulation

Connecting with the Bernoulli model notation, we consider the occurrence of red spirals as our

response variable $T \equiv$ type (thus $T = 1$ if the galaxy is a red spiral, a success, and $T = 0$ otherwise, a failure). The bulge size will act as our explanatory variable $x \equiv \text{fracdev}$ and $\eta = \beta_1 + \beta_2 x$ as the linear predictor, whose intercept and slope are given by β_1 and β_2 , respectively. The Bernoulli parameter p is interpreted as the probability that a galaxy is a red spiral (the probability of success), given its bulge size. The Bernoulli model for this problem can be expressed as follows:

$$\begin{aligned}
 T_i &\sim \text{Bernoulli}(p_i) \\
 \log\left(\frac{p_i}{1 - p_i}\right) &= \eta_i \\
 \eta_i &= \beta_1 + \beta_2 x_i \\
 \beta_j &\sim \text{Normal}(0, 10^3) \\
 i &= 1, \dots, N \\
 j &= 1, \dots, K
 \end{aligned} \tag{10.11}$$

where $N = 5433$ is the number of data points and $K = 2$ the number of linear predictor coefficients.

10.6.3 Running the Model in R using JAGS

Code 10.12 Bernoulli model in R using JAGS, for accessing the relationship between bulge size and the fraction of red spirals.

```

=====
require(R2jags)

# Data
path_to_data = '~/data/Section_10p6/Red_spirals.csv'

# Read data
Red <- read.csv(path_to_data, header=T)

# Prepare data to JAGS
N <- nrow(Red)
x <- Red$fracdev
y <- Red$type

# Construct data dictionary
X <- model.matrix(~ x,
                  data = Red)
K <- ncol(X)
logit_data <- list(Y = y,           # response variable
                     X = X,          # predictors
                     N = N,          # sample size
                     K = K,          # number of columns
)
# Fit
LOGIT <- "model{
  # Diffuse normal priors
  for(i in 1:K){
    beta[i] ~ dnorm(0, 1e-4)
  }
  # Likelihood function
"

```

```

for (i in 1:N){
  Y[i] ~ dbern(p[i])
  logit(p[i]) <- eta[i]
  eta[i]      <- inprod(beta[], X[i,])
}
}

# Define initial values
inits <- function () {
  list(beta = rnorm(ncol(X), 0, 0.1))
}

# Identify parameters
params <- c("beta")

# Fit
LOGIT_fit <- jags(data = logit_data,
                     inits = inits,
                     parameters = params,
                     model = textConnection(LOGIT),
                     n.thin = 1,
                     n.chains = 3,
                     n.burnin = 3000,
                     n.iter = 6000)

# Output
print(LOGIT_fit,intervals=c(0.025, 0.975),justify = "left", digits=2)
=====
          mu.vect    sd.vect    2.5%    97.5%   Rhat   n.eff
beta[1]     -4.89     0.23    -5.23    -4.56     1    2100
beta[2]      8.11     0.59     7.12     9.06     1    2200
deviance   1911.15    44.15  1907.07  1915.78     1    9000

pD = 974.7 and DIC = 2885.8

```

Figure 10.8 illustrates how the probability that a galaxy is a red spiral depends on `fracdev` (or bulge size) according to our model. The dashed line represents the mean posterior, and the shaded regions denote the 50% (darker) and 95% (lighter) credible intervals. In order to facilitate comparison with Figure 2 of [Masters et al. \(2010\)](#), we also show the binned data corresponding to the fraction of red spirals, $N_{\text{red}}/N_{\text{tot}}$, in bin bulge-size each (dots). The model fits the data quite well with a simple linear relation (see Equation 10.12). In order to reproduce this figure with his or her own plot tool, the reader just needs to monitor the parameter p , using

```
params <- c("p")
```

and then extract the values with the function `jagsresults` from the package `jagstools`:

```
px <- jagsresults(x=LOGIT_fit, params=c('p'))
```

The output will look like

```
head(round(px,4))
      mean      sd    2.5%    25%    50%    75%   97.5%   Rhat   n.eff
p[1]  0.0077  0.0058  0.0053  0.0065  0.0073  0.0081  0.0101  1.0046  530
p[2]  0.0077  0.0058  0.0054  0.0066  0.0073  0.0081  0.0102  1.0046  530
p[3]  0.0078  0.0058  0.0054  0.0067  0.0074  0.0082  0.0102  1.0046  530
p[4]  0.0079  0.0058  0.0055  0.0067  0.0075  0.0083  0.0103  1.0046  530
p[5]  0.0079  0.0058  0.0055  0.0068  0.0075  0.0083  0.0104  1.0046  530
p[6]  0.0080  0.0058  0.0056  0.0068  0.0076  0.0084  0.0105  1.0046  530
```

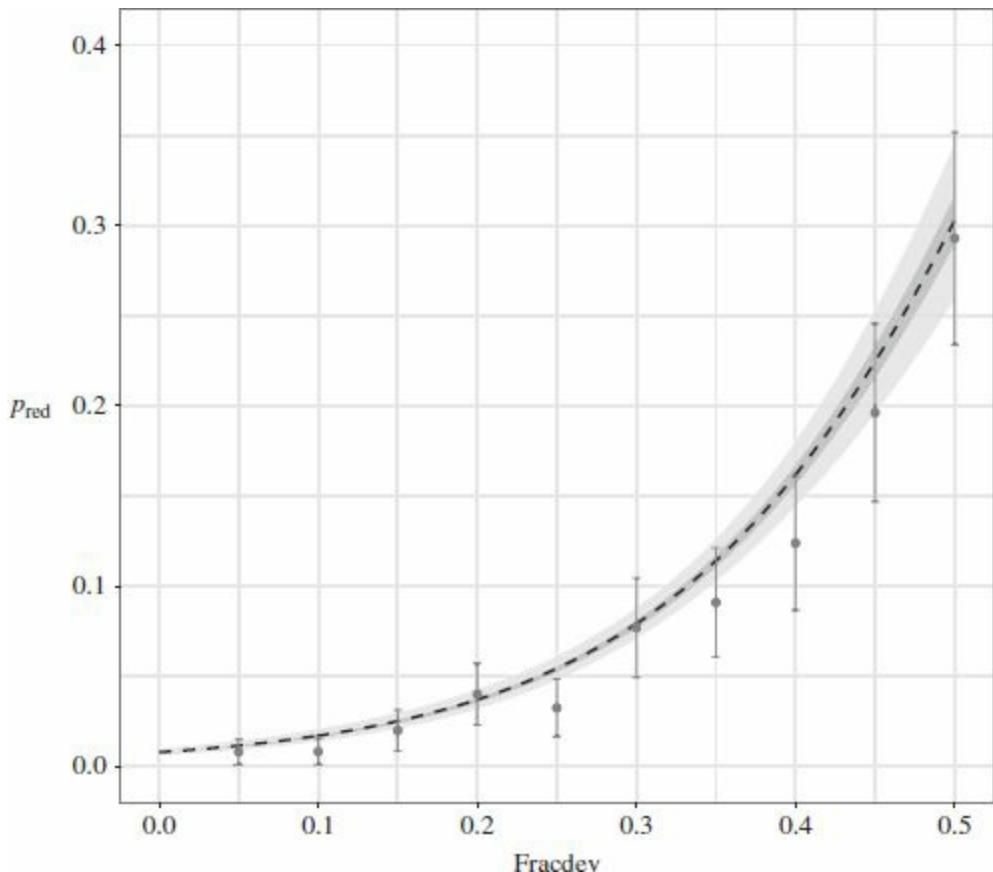


Figure 10.8 Probability that a given galaxy is red spiral, P_{red} , as a function of fracdev (or bulge size). The dashed line represents the posterior mean probability that a galaxy is a red spiral, while the shaded areas illustrate the 50% (darker) and 95% (lighter) credible intervals. The data points, with error bars, represent the fraction of red spirals for each bulge-size bin as presented in Figure 2 of [Masters et al. \(2010\)](#).

10.6.4 Running the Model in Python using Stan

The implementation in Python using Stan is quite straightforward. One important aspect to highlight is that Stan has a built-in Bernoulli distribution with a logit link, `bernoulli_logit`, which handles a lot of simplifications. This allows a more stable computation and a more compact model code.

Code 10.13 Bernoulli model in Python using Stan, for assessing the relationship between bulge size and the fraction of red spirals.

```
=====
import numpy as np
import pandas as pd
import pystan
import statsmodels.api as sm

# Data
path_to_data = '~/data/Section_10p6/Red_spirals.csv'

# Read data
data_frame = dict(pd.read_csv(path_to_data))
x = np.array(data_frame['fracdev'])

# Prepare data for Stan
data = {}
data['X'] = sm.add_constant((x.transpose()))
data['Y'] = np.array(data_frame['type'])
```

```

data['nobs'] = data['X'].shape[0]
data['K'] = data['X'].shape[1]

# Fit
# Stan model
stan_code="""
data{
    int<lower=0> nobs;                      # number of data points
    int<lower=0> K;                         # number of coefficients
    matrix[nobs, K] X;                      # bulge size
    int Y[nobs];                           # galaxy type: 1, red; 0, blue
}

parameters{
    vector[K] beta;                      # linear predictor coefficients
}

model{
    # priors and likelihood
    for (i in 1:K) beta[i] ~ normal(0, 100);

    Y ~ bernoulli_logit(X * beta);
}
"""

# Run mcmc
fit = pystan.stan(model_code=stan_code, data=data, iter=6000, chains=3,
                    warmup=3000, thin=1, n_jobs=3)

# Output
print(fit)
=====
      mean   se_mean     sd   2.5%   25%   50%   75%   97.5%   n_eff   Rhat
beta[0] -4.92   4.6e-3  0.16 -5.25 -5.02 -4.92 -4.81 -4.61  1232.0   1.0
beta[1]   8.18     0.01  0.46  7.29  7.87  8.17  8.49  9.12  1233.0   1.0

```

10.7 Count Models, Globular Cluster Population, and Host Galaxy Brightness

We approach now a recurrent issue in astronomical studies: the necessity to model discrete (count) data. Astronomers often find that they need to establish relationships involving at least one discrete variable, where standard approaches designed to deal with continuous variables cannot operate ([de Souza et al., 2015b](#)). In such situations a common approach is to analyze pairs of measured quantities (x, y) in loglog scale and then apply the standard normal model with a Gaussian error distribution (these are the underlying assumptions behind a χ^2 minimization). Such a transformation is not necessary (see Chapter 5), since the GLM framework allows one to treat the data at its original scale. Moreover, a log transformation makes it impossible to deal with zeros as observations and imposes an arbitrary shift on the entire data set. These transformations are known to have a poor performance and to lead to bias in parameter estimation ([O'Hara and Kotze, 2010](#)).

However, merely stating that one should use a statistical model able to handle count data is far from being a final answer to the above problem. A reader who has gone through Chapter 5 might be aware that there are several distributions which can be used to model count data; choosing between them can pose an additional challenge. Following the arguments presented by [de Souza et al. \(2015b\)](#), we present three distinct models with increasing levels of complexity. This allows us to compare results from Poisson, negative binomial and three-parameter negative binomial models in order to guide the reader through a model selection exercise. We will use, as a case study, measurements of globular cluster (GC) population size N_{GC} and host galaxy visual magnitude M_V .

Globular clusters are spherical groups of stars found mainly in the halos of galaxies (Figure 10.9). They are gravitationally bounded, significantly more dense than the open clusters populating the disk and pervasive in nearby massive galaxies (Kruijssen, 2014). They are also among the oldest known stellar systems, which makes them important pieces in the galaxy evolution puzzle. Being one of the first structures to form in a galaxy, it is reasonable to expect that the GC population is correlated somehow with global host galaxy properties. This hypothesis was confirmed by a number of previous studies, which also found the Milky Way to be an important outlier (Burkert and Tremaine, 2010; Harris and Harris, 2011; Harris *et al.*, 2013, 2014; Rhode, 2012; Snyder *et al.*, 2011). Our galaxy holds a significantly larger number of GCs than expected given the mass of its central black hole. More recently, de Souza *et al.* (2015b) showed that the use of a proper statistical model results in prediction intervals which enclose the Milky Way in a natural way, demonstrating the paramount role played by statistical models in astronomical data modeling.



Figure 10.9 Globular cluster NGC 6388. Image credits: ESO, F. Ferraro (University of Bologna).

10.7.1 Data

We use the same catalog as that described in Section 10.1.1, which was presented in Harris *et al.* (2013, hereafter H2013). This is a compilation of literature data, from a variety of sources, obtained with the Hubble Space Telescope as well as a wide range of other ground-based facilities (see H2013 for further details and references). The original data set holds N_{GC} and visual magnitude measurements for 422 galaxies.

In order to focus on the differences resulting solely from the choice of statistical model, the examples shown below will not take into account errors in measurements. A detailed explanation of how those can be handled, including the corresponding JAGS models, can be found in [de Souza et al. \(2015b\)](#).

10.7.2 The Statistical Poisson Model Formulation

In this example, the population of globular clusters in a given galaxy is our response variable N_{GC} , and the visual absolute magnitude our explanatory variable M_V . The total number of galaxies is denoted by $N = 420$, the linear predictor being $\eta = \mu = \beta_1 + \beta_2 x$, with β_1 and β_2 as the intercept and slope, respectively, and $K = 2$ the number of coefficients in the linear predictor.

Once we have decided to use a discrete distribution, the natural choice is a simple Poisson model. Despite being fairly popular in the astronomical literature, it involves the hypothesis that the mean and scatter are the same throughout the domain of the explanatory variable, an assumption that is rarely fulfilled in real data situations. We show below how it performs in our particular case.

The Poisson model for this problem is expressed as follows:

$$\begin{aligned}
 N_{GC;i} &\sim Poisson(\mu_i) \\
 \mu_i &= \beta_1 + \beta_2 M_{V;i} \\
 \beta_j &\sim Normal(0, 10^3) \\
 i &= 1, \dots, N \\
 j &= 1, \dots, K
 \end{aligned} \tag{10.12}$$

where we have assigned non-informative priors for β_1 and β_2 .

10.7.3 Running the Poisson Model in R using JAGS

The implementation of the model given in Equation 10.13 in R using JAGS is shown below. Notice that, as we intend to compare the results with those from negative binomial and three-parameter negative binomial models, we have also included a calculation of the dispersion parameter.

Code 10.14 Poisson model, in R using JAGS, for modeling the relation between globular clusters population and host galaxy visual magnitude.

```
=====
require(R2jags)
require(jagstools)

# Data
path_to_data = "~/data/Section_10p7/GCs.csv"
```

```

# Read data
GC_dat = read.csv(file=path_to_data, header = T, dec=".")

# Prepare data to JAGS
N <- nrow(GC_dat)
x <- GC_dat$MV_T
y <- GC_dat$N_GC
X <- model.matrix(~ x, data=GC_dat)
K = ncol(X)

JAGS_data <- list(
  Y = y,
  X = X,
  N = N,
  K = K)

# Fit
model.pois <- "model{
  # Diffuse normal priors betas
  for (i in 1:K) { beta[i] ~ dnorm(0, 1e-5)}

  for (i in 1:N){

    # Likelihood
    eta[i] <- inprod(beta[], X[i,])
    mu[i] <- exp(eta[i])
    Y[i] ~ dpois(mu[i])

    # Discrepancy
    expY[i] <- mu[i]           # mean
    varY[i] <- mu[i]           # variance
    PRes[i] <- ((Y[i] - expY[i])/sqrt(varY[i]))^2
  }
  Dispersion <- sum(PRes)/(N-2)
}"

# Define initial values
inits <- function () {
  list(beta = rnorm(K, 0, 0.1))
}

# Identify parameters
params <- c("beta", "Dispersion")

# Start JAGS
pois_fit <- jags(data      = JAGS_data ,
                   inits     = inits,
                   parameters = params,
                   model     = textConnection(model.pois),
                   n.thin   = 1,
                   n.chains = 3,
                   n.burnin = 3500,
                   n.iter   = 7000)

# Output
print(pois_fit , intervals=c(0.025, 0.975), digits=3)
=====

mu.vect    sd.vect    2.5%    97.5%    Rhat    n.eff
Dispersion  1080.231   0.197   1079.888   1080.590  1.010  10000
beta[1]     -11.910   0.035   -11.967   -11.855  1.021  9100
beta[2]     -0.918   0.002   -0.920   -0.915  1.020  9400
deviance    497168.011  898.313  495662.548  498682.310  1.008  10000

pD = 403480.4 and DIC = 900648.4

```

A visual representation of the posterior distribution for the fitted coefficients and the dispersion statistics (Figure 10.10) can be accessed with the following command in R:

```

require(lattice)
source("../CH-Figures.R")
out <- pois_fit$BUGSoutput
MyBUGSHist(out,c("Dispersion",uNames(beta"),K)))

```

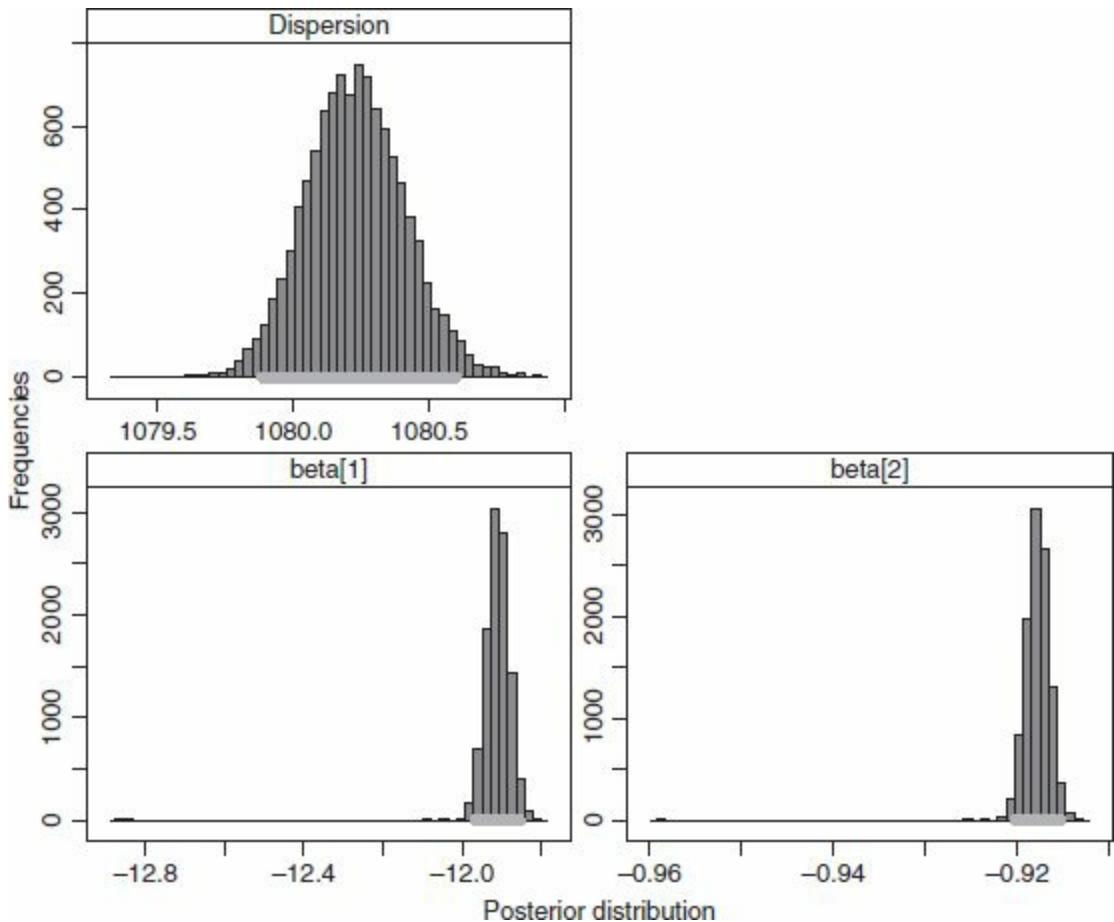


Figure 10.10 Posterior distributions over the intercept (β_1), slope (β_2), and dispersion parameter resulting from the Poisson model shown in Code 10.14. The thick horizontal lines show the 95% credible intervals.

The large dispersion statistic value ($> 10^3$) indicates that this model is not a good description of the underlying behavior of our data. This is confirmed by confronting the original data with the model mean and corresponding prediction intervals. Figure 10.11 clearly shows that the model does not describe a significant fraction of the data points and consequently it has a very limited prediction capability.

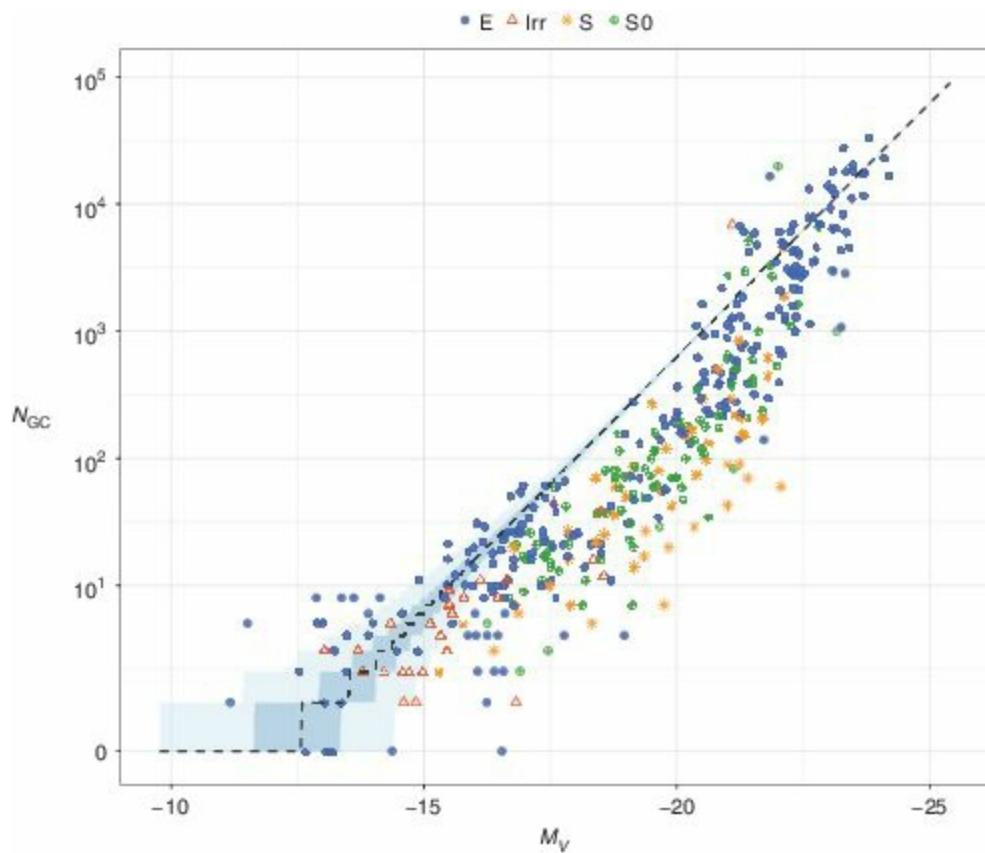


Figure 10.11 Globular cluster population and host galaxy visual magnitude data (points) superimposed on the results from Poisson regression. The dashed line shows the model mean and the shaded regions mark the 50% (darker) and 95% (lighter) prediction intervals.

10.7.4 The Statistical Negative Binomial Model Formulation

As stated in Chapter 5, in cases of overdispersion the Poisson model must be substituted by another model which disentangles the mean and variance parameters. A possible solution is to apply a negative binomial model, which is able to account for the extra scatter. We now investigate how a change in the choice of statistical model affects the posterior mean in this particular case.

The negative binomial model can be expressed as follows:

$$\begin{aligned}
N_{GC;i} &\sim NegBinomial(p_i, \theta) \\
p_i &= \frac{\theta}{\theta + \mu_i} \\
\mu_i &= \exp(\eta_i) \\
\eta_i &= \beta_1 + \beta_2 M_{V;i} \\
\beta_j &\sim Normal(0, 10^3) \\
\theta &\sim Gamma(10^{-3}, 10^{-3}) \\
i &= 1, \dots, N \\
j &= 1, \dots, K
\end{aligned} \tag{10.13}$$

where we denote as θ the extra scatter parameter and where we have assigned non-informative priors for β_j and θ .

10.7.5 Running the Negative Binomial Model in R using JAGS

The corresponding implementation in R using JAGS is shown in Code 10.15. Here we skip the lines dealing with data input and preparation since they are the same as those in Code 10.14. Notice that the extra scatter parameter also requires changes in the definition of dispersion.

Code 10.15 Negative binomial model, in R using JAGS, for modeling the relationship between globular cluster population and host galaxy visual magnitude.

```
=====
# Fit
model.NB <- "model{
  # Diffuse normal priors betas
  for (i in 1:K) { beta[i] ~ dnorm(0, 1e-5)}

  # Prior for theta
  theta ~ dgamma(1e-3,1e-3)
  for (i in 1:N){
    eta[i] <- inprod(beta[], X[i,])
    mu[i] <- exp(eta[i])
    p[i] <- theta/(theta+mu[i])
    Y[i] ~ dnegbin(p[i],theta)

  # Discrepancy
  expY[i] <- mu[i]                                # mean
  varY[i] <- mu[i] + pow(mu[i],2)/theta           # variance
  PRes[i] <- ((Y[i] - expY[i])/sqrt(varY[i]))^2
  }
  Dispersion <- sum(PRes)/(N-3)
}

# Define initial values
inits <- function () {
  list(beta = rnorm(K, 0, 0.1))
}

# Identify parameters
params <- c("beta","theta","Dispersion")
```

```

# Start JAGS
NB_fit <- jags(data = JAGS_data ,
                  inits = inits,
                  parameters = params,
                  model = textConnection(model.NB),
                  n.thin = 1,
                  n.chains = 3,
                  n.burnin = 3500,
                  n.iter = 7000)

# Output

# Plot posteriors
MyBUGSHist(out,c("Dispersion",uNames("beta"),"theta"))

# Dump results on screen
print(NB_fit, intervals=c(0.025, 0.975), digits=3)
=====
      mu.vect    sd.vect      2.5%     97.5%     Rhat   n.eff
Dispersion    1.928     0.207     1.554     2.370    1.005     490
beta[1]     -11.750     0.315    -12.322    -11.109    1.042      62
beta[2]      -0.880     0.016     -0.909     -0.849    1.042      62
theta        1.099     0.071     0.967     1.246    1.002    1500
deviance    5191.414     2.388   5188.709    5197.695    1.004     580

pD = 2.8 and DIC = 5194.3

```

The model is slightly overdispersed ($\text{Dispersion} \approx 2.0$, see the top left panel in Figure 10.12), which could be caused either by a missing covariate or by the lack of complexity in the model. However, Figure 10.13 shows that it provides a quite good fit to the data, with the 95% prediction intervals enclosing most of the data variance. Moreover, comparing its DIC statistic ($\text{DIC} = 5194.3$) with that obtained with the Poisson model ($\text{DIC} = 900648.4$, from Code 10.14) we notice that the negative binomial model is significantly preferable.

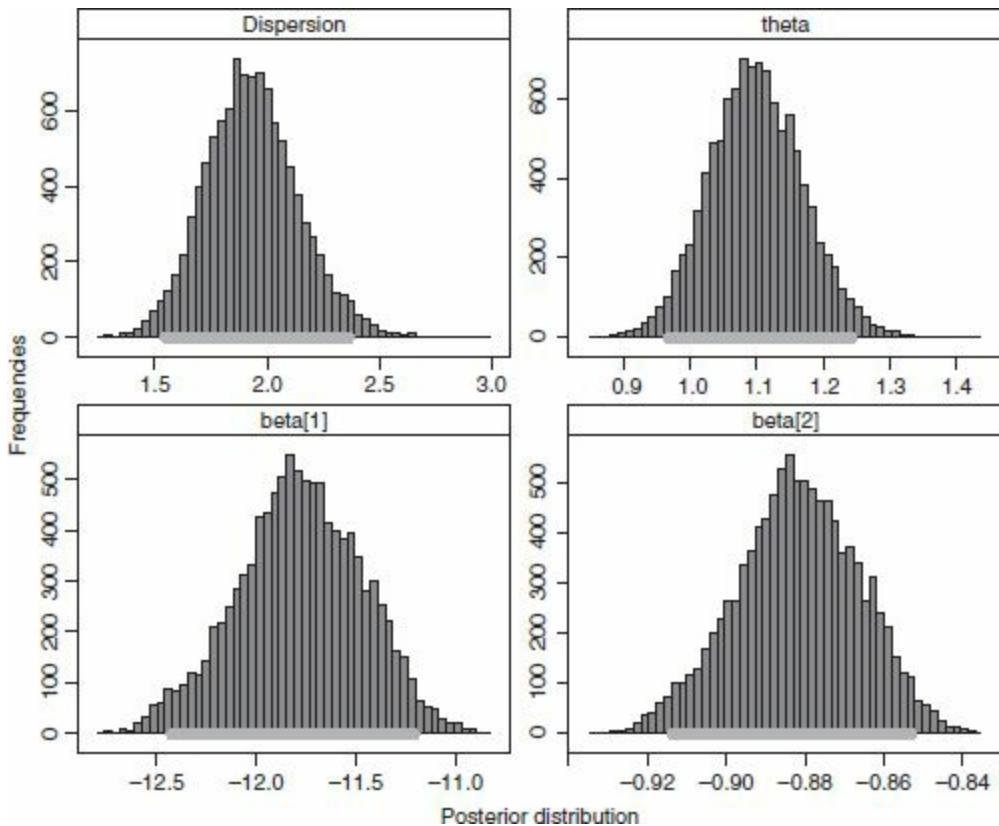


Figure 10.12 Posterior distributions over the intercept (β_1), slope (β_2), scatter (θ), and dispersion parameter resulting from the negative binomial model shown in Code 10.15. The horizontal thick lines show the 95% credible intervals.

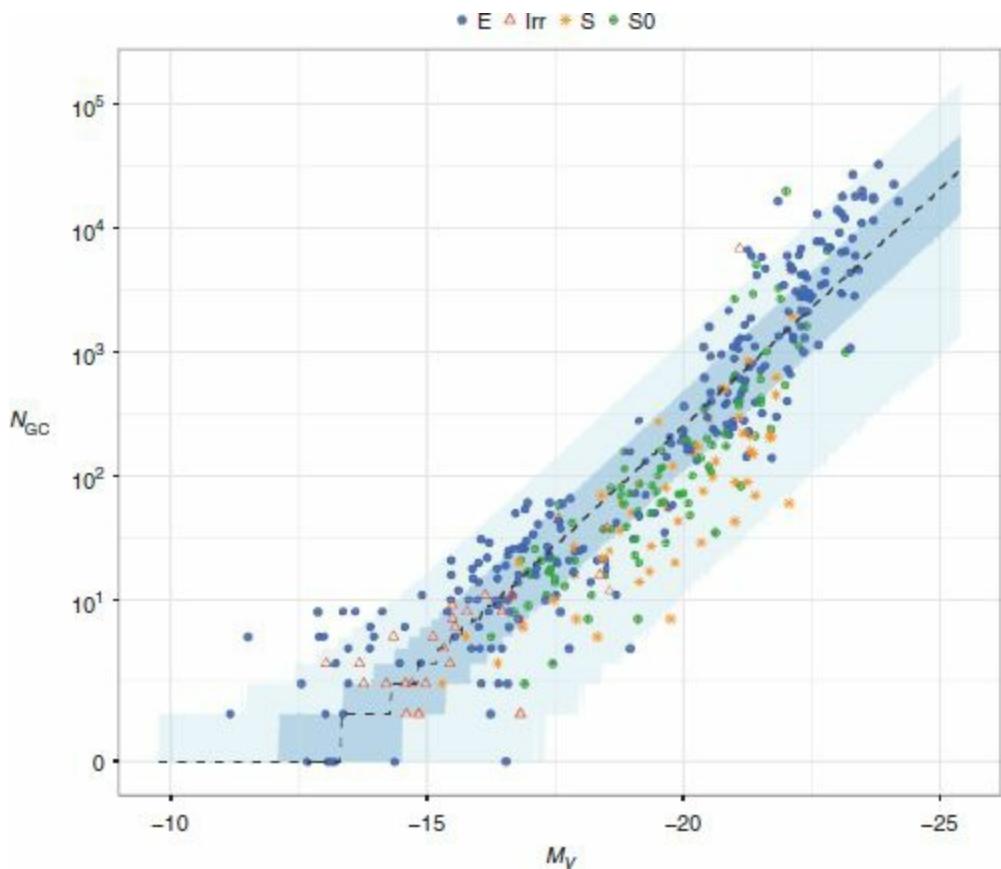


Figure 10.13 Globular cluster population and host galaxy visual magnitude data (points) superimposed on the results from negative binomial regression. The dashed line shows the model mean. The shaded regions mark the 50% (darker) and 95% (lighter) prediction intervals.

10.7.6 The Statistical NB-P Model Formulation

Finally let us investigate whether the use of a more complex statistical model could result in an even better fit to our data. We will test the three-parameter negative binomial model (NB-P), which allows a second parameter to model the data dispersion (see Section 6.5).

The NB-P model for this problem can be expressed as follows:

$$\begin{aligned}
N_{GC;i} &\sim NegBinomial(p_i, \theta \mu_i^Q) \\
p_i &= \frac{\theta \mu_i^Q}{\theta \mu_i^Q + \mu_i} \\
\mu_i &= \exp(\eta_i) \\
\eta_i &= \beta_1 + \beta_2 M_{V;i} \\
\beta_j &\sim Normal(0, 10^3) \\
\theta &\sim Gamma(10^{-3}, 10^{-3}) \\
Q &\sim Uniform(0, 3) \\
i &= 1, \dots, N \\
j &= 1, \dots, K
\end{aligned} \tag{10.14}$$

where we have assigned non-informative priors for β_j and θ and a uniform prior over the additional dispersion parameter Q .

10.7.7 Running the NB-P Model in R using JAGS

The implementation in R using JAGS is shown below. The data input lines to be used in Code 10.16 are the same as those shown in Code 10.15.

Code 10.16 NB-P model in R using JAGS, for modeling the relationship between globular cluster population and host galaxy visual magnitude.

```
=====
# Fit
model.NBP <- "model{

  # Priors for regression coefficients
  # Diffuse normal priors betas
  for (i in 1:K) { beta[i] ~ dnorm(0, 1e-5)}

  # Prior for size
  theta ~ dgamma(1e-3,1e-3)

  # Uniform prior for Q
  Q ~ dunif(0,3)

  # Likelihood
  for (i in 1:N){
    eta[i] <- inprod(beta[], X[i,])
    mu[i] <- exp(eta[i])
    theta_eff[i] <- theta*(mu[i]^Q)
    p[i] <- theta_eff[i]/(theta_eff[i]+mu[i])
    Y[i] ~ dnbinom(p[i],theta_eff[i])

    # Discrepancy
    expY[i] <- mu[i]          # mean
    varY[i] <- mu[i] + pow(mu[i],2-Q)/theta #variance
    PRes[i] <- ((Y[i] - expY[i])/sqrt(varY[i]))^2
  }
}
```

```

        Dispersion <- sum(PRes)/(N-4)#
}"
```

```

# Set initial values
inits <- function () {
  list(
    beta  = rnorm(K, 0, 0.1),
    theta = runif(1,0.1,5),
    Q = runif(1,0,1)
  )
}
```

```

# Identify parameters
params <- c("Q","beta","theta","Dispersion")
```

```

# Start JAGS
NBP_fit <- jags(data      = JAGS_data,
                  inits     = inits,
                  parameters = params,
                  model     = textConnection(model.NBP),
                  n.thin   = 1,
                  n.chains = 3,
                  n.burnin = 5000,
                  n.iter    = 20000)
```

```

# Output
# Plot posteriors
MyBUGSHist(out,c("Dispersion",uNames("beta"),K),"theta"))

# Screen output
print(NBP_fit, intervals=c(0.025, 0.975), digits=3)
=====
```

	mu.vect	sd.vect	2.5%	97.5%	Rhat	n.eff
Dispersion	1.929	0.209	1.545	2.363	1.001	13000
Q	0.018	0.015	0.001	0.057	1.002	2200
beta[1]	-11.822	0.333	-12.447	-11.145	1.003	1200
beta[2]	-0.884	0.017	-0.916	-0.850	1.003	1200
theta	0.996	0.106	0.770	1.187	1.002	1800
deviance	5193.090	3.012	5189.255	5200.795	1.004	1300

pD = 4.5 and DIC = 5197.6

Comparing the mean `Dispersion` parameters for the NB-P (1.929) and negative binomial (1.928) models we notice no significant improvement, which is visually confirmed by comparing Figures 10.12 and 10.14. Notice that the `Q` parameter value is consistent with zero, which indicates that the extra complexity is not being employed in this more detailed description of the data. Finally, the dispersion statistic of the NB-P model (`DIC` = 5197.6) is slightly higher than that for the NB (`DIC` = 5194.3), which also indicates that the NB model is more suited to describe our data.

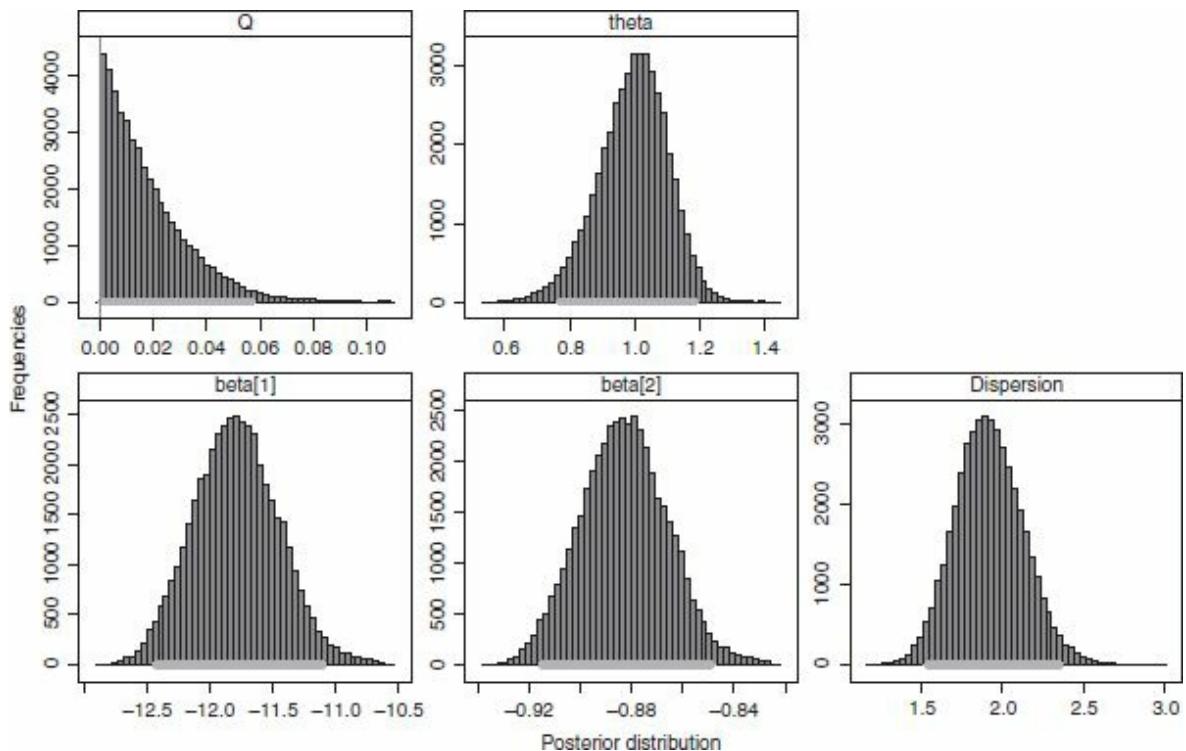


Figure 10.14 Posterior distributions over the intercept (β_1), slope (β_2), and dispersion (θ , Q) parameters resulting from the three-parameter negative binomial model (NB-P) shown in Code 10.16. The horizontal thick lines show the 95% credible intervals.

After comparing three possible statistical models we conclude that, among them, the negative binomial model provides the best fit to the globular-cluster-population versus galaxy-visual-magnitude data. We have also shown that, although in this model there is still some remaining overdispersion, using a more complex statistical model does not improve the final results. This issue might be solved by addressing other important points such as the measurement errors and/or the existence of subpopulations within the data set.

A more detailed analysis of the possible sources and methods to deal with this data set is beyond our scope here, but we invite readers interested in a deeper analysis to check the discussion in [de Souza et al. \(2015b\)](#).

10.7.8 Running the NB-P Model in Python using Stan

We present below the equivalent of Code 10.15 in Python using Stan. The correspondence between the two codes is straightforward. We highlight only the absence of the parameter p . This is due to the different parameterizations for built-in negative binomial distributions in JAGS and Stan. The results for the model and dispersion parameters are almost identical to those obtained with the R counterpart of this code.

Code 10.17 Negative binomial model in Python using Stan, for modeling the relationship between globularcluster population and host galaxy visual magnitude.

```

import numpy as np
import pandas as pd
import pystan
import statsmodels.api as sm

# Data
path_to_data = '~/data/Section_10p7/GCs.csv'

data_frame = dict(pd.read_csv(path_to_data))

# Prepare data for Stan
data = {}
data['X'] = sm.add_constant(np.array(data_frame['MV_T']))
data['Y'] = np.array(data_frame['N_GC'])
data['N'] = len(data['X'])
data['K'] = 2

# Fit
stan_code=""""
data{
    int<lower=0> N;           # number of data points
    int<lower=1> K;           # number of linear predictor coefficients
    matrix[N,K] X;            # galaxy visual magnitude
    int Y[N];                 # size of globular cluster population
}
parameters{
    vector[K] beta;          # linear predictor coefficients
    real<lower=0> theta;
}
model{
    vector[N] mu;             # linear predictor

    mu = exp(X * beta);
    theta ~ gamma(0.001, 0.001);

    # likelihood and priors
    Y ~ neg_binomial_2(mu, theta);
}
generated quantities{
    real dispersion;
    vector[N] expY;           # mean
    vector[N] varY;            # variance
    vector[N] PRes;
    vector[N] mu2;

    mu2 = exp(X * beta);
    expY = mu2;

    for (i in 1:N){
        varY[i] = mu2[i] + pow(mu2[i], 2) / theta;
        PRes[i] = pow((Y[i] - expY[i]) / sqrt(varY[i]),2);
    }
    dispersion = sum(PRes) / (N - (K + 1));
}
"""
# Run mcmc
fit = pystan.stan(model_code=stan_code, data=data, iter=10000, chains=3,
                    warmup=5000, thin=1, n_jobs=3)

# Output
nlines = 9                      # number of lines in screen output

output = str(fit).split('\n')
for item in output[:nlines]:
    print(item)
=====
      mean   se_mean     sd    2.5%    ...   97.5%   n_eff   Rhat
beta[0]    -11.73  6.8e-3   0.33   -12.38   ...  -11.07  2349.0   1.0
beta[1]     -0.88  3.4e-4   0.02   -0.91   ...   -0.85  2339.0   1.0
theta       1.1   1.4e-3   0.07    0.96   ...    1.25  2650.0   1.0
dispersion   1.92  3.4e-3   0.21    1.54   ...    2.35  3684.0   1.0

```

10.8 Bernoulli Mixed Model, AGNs, and Cluster Environment

Our next example describes a mixed model with a binary response variable following a Bernoulli distribution. This is the binomial model equivalent to the normal examples shown in Sections 10.2 and 10.3. As a case study, we reproduce the study from [de Souza et al. \(2016\)](#) exploring the roles of morphology and environment in the occurrence of AGN in elliptical and spiral galaxies.

Active galactic nuclei (AGN) are powered by the accretion of gas into a supermassive black hole located at the center of their host galaxy (e.g. [Lynden-Bell, 1969](#); [Orban de Xivry et al., 2011](#)). The AGN feedback interacts with the gas of the host via radiation pressure, winds, and jets, hence helping to shape the final mass of the stellar components ([Fabian, 2012](#)). Environmental effects can also turn on or off the AGN activity. Instabilities originating from galaxy mergers, and from interactions between the galaxy and the cluster potential, could drive gas towards the galaxy center, powering the AGN. Recently, [Pimbblet et al. \(2013\)](#) found a strong relation between AGN activity and the cluster-centric distance r/r_{200} ²⁰ with significant decrease in AGN fraction towards the cluster center. The interplay between these competing processes results in a very intricate relation between the AGN activity, the galaxy properties, and the local environment, which requires careful statistical analysis.

Using a hierarchical Bayesian model, we show that it is possible to handle such intricate data scenarios on their natural scale and, at the same time, to take into account the different morphological types in a unified framework (without splitting the data into independent samples).

10.8.1 Data

The data set is composed of a subsample of $N = 1744$ galaxies within galaxy clusters from the *Sloan Digital Sky Survey* seventh (SDSS-DR7, [Abazajian et al., 2009](#)) and 12th (SDSS-DR12, [Alam et al., 2015](#)) data release data bases within a redshift range $0.015 < z < 0.1$. The sample is divided into two groups, ellipticals and spirals, relying on the visual classification scheme from the *Galaxy Zoo Project*²¹ ([Lintott et al., 2008](#)).

This sample was classified according to the diagram introduced in the article Baldwin, Phillips, and Terlevich (1981, hereafter BPT), as shown in Figure 10.15. It comprises galaxies with emission lines H β , [OIII], H α , and [NII] and signal to noise ratio $S/N > 1.5$. In order to build a sample with the lowest possible degree of contamination due to wrong or dubious classifications, we selected only the star-forming and Seyfert galaxies (see [de Souza et al., 2016](#), Section 2.1). The galaxies hosting Seyfert AGN were compared with a control sample of inactive galaxies by matching each Seyfert and non-Seyfert galaxy pair against their colors, star formation rates, and stellar masses.

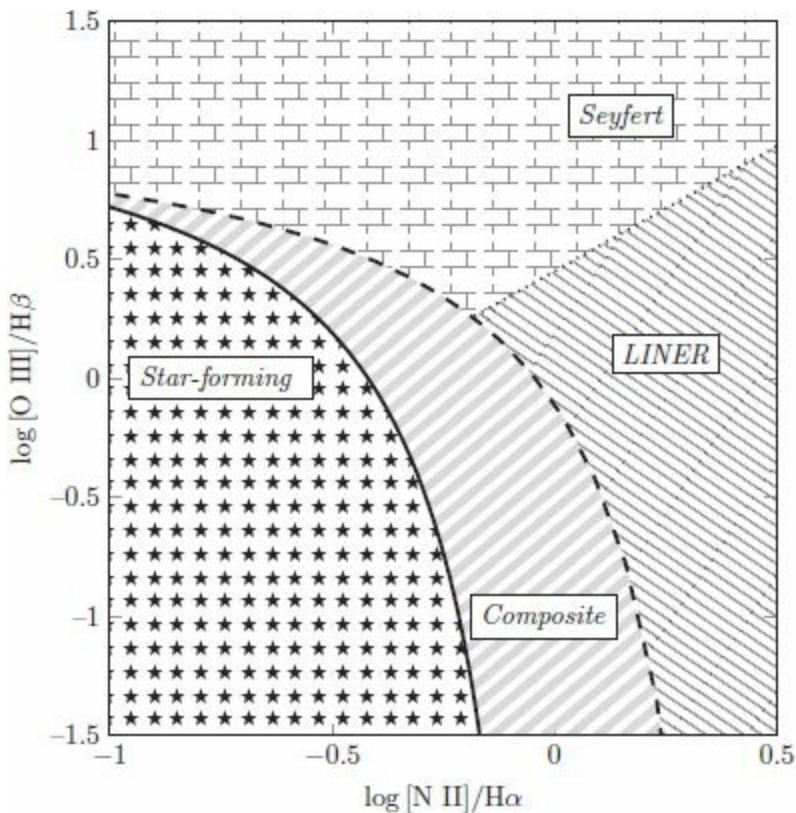


Figure 10.15 Illustrative plot of the BPT plane. The vertical axis represents the ratio $[\text{OIII}]/\text{H}\beta$, while the horizontal axis represents the ratio $[\text{NII}]/\text{H}\alpha$. The solid curve is due to [Kauffmann et al. \(2003\)](#): galaxies above the curve are designated AGN and those below are regular star-forming galaxies. The dashed line represents the [Kewley et al. \(2001\)](#) curve; galaxies between the [Kauffmann et al.](#) and [Kewley et al.](#) curves are defined as composites; weaker AGN whose hosts are also star-forming galaxies. The dotted line is the [Schawinski et al. \(2007\)](#) curve, which separates low-ionization nuclear-emission-line (LINER) and Seyfert objects.

10.8.2 Statistical Model Formulation

The explanatory variables reflecting the environment where the galaxy resides and the host cluster property are the cluster-centric distance r/r_{200} and galaxy cluster mass $M = \log M_{200}$.

The model is portrayed in Equation 10.15:

$$\begin{aligned}
y_i &\sim \text{Bernoulli}(p_i) \\
\text{logit}(p_i) &= \eta_i \\
\eta_i &= x_{i,k}^T \beta_{k,j} \\
x_{i,k}^T &= \begin{pmatrix} 1 & (\log M_{200})_1 & \left(\frac{r}{r_{200}}\right)_1 \\ \vdots & \vdots & \vdots \\ 1 & (\log M_{200})_N & \left(\frac{r}{r_{200}}\right)_N \end{pmatrix} \\
\beta_{k,j} &\sim \text{Normal}(\mu, \sigma^2) \\
\mu &\sim \text{Normal}(0, 10^3) \\
\tau &\sim \text{Gamma}(10^{-3}, 10^{-3}) \\
\sigma^2 &= 1/\tau \\
j &= \text{elliptical, spiral} \\
k &= 1, \dots, 3 \\
i &= 1, \dots, N
\end{aligned} \tag{10.15}$$

It reads as follows: for each galaxy in the data set, composed of N objects, its probability of hosting a Seyfert AGN is described by a Bernoulli distribution whose probability of success, $p \equiv f_{\text{Seyfert}}$, relates to r/r_{200} and $\log M_{200}$ through a logit link function (to ensure that the probabilities will fall between 0 and 1) and the linear predictor

$$\eta = \beta_{1,j} + \beta_{2,j} \log M_{200} + \beta_{3,j} r/r_{200}, \tag{10.16}$$

where j is an index representing whether a galaxy is elliptical or spiral. We assume non-informative priors for the coefficients $\beta_1, \beta_2, \beta_3$, i.e., normal priors with mean μ and standard deviation σ for which we assign shared hyperpriors $\mu \sim \text{Normal}(0, 10^3)$ and $1/\sigma^2 \sim \text{Gamma}(10^{-3}, 10^{-3})$.²² By employing a hierarchical Bayesian model for the varying coefficients β_j , we allow the model to borrow strength across galaxy types. This happens via the joint influence of these coefficients on the posterior estimates of the unknown hyperparameters μ and σ^2 .

10.8.3 Running the Model in R using JAGS

Code 10.18 Bernoulli logit model, in R using JAGS, for accessing the relationship between Seyfert AGN activity and galactocentric distance.

```
library(R2jags)
```

```

# Data
data<-read.csv("~/data/Section_10p8/Seyfert.csv", header=T)

# Identify data elements
X <- model.matrix( ~ logM200 + r_r200, data = data)
K <- ncol(X)                                     # number of predictors
y <- data$bpt                                    # response variable
n <- length(y)                                   # sample size
gal <- as.numeric(data$zoo)                      # galaxy type

# Prepare data for JAGS
jags_data <- list(Y = y,
                   N = n,
                   X = X,
                   gal = gal)

# Fit
jags_model<-"model{
    # Shared hyperpriors for beta
    tau ~ dgamma(1e-3,1e-3)                      # precision
    mu ~ dnorm(0,1e-3)                            # mean

    # Diffuse prior for beta
    for(j in 1:2){
        for(k in 1:3){
            beta[k,j]~dnorm(mu,tau)
        }
    }

    # Likelihood
    for(i in 1:N){
        Y[i] ~ dbern(pi[i])
        logit(pi[i]) <- eta[i]
        eta[i] <- beta[1,gal[i]]*X[i,1]+
                    beta[2,gal[i]]*X[i,2]+
                    beta[3,gal[i]]*X[i,3]
    }
}

# Identify parameters to monitor
params <- c("beta")

# Generate initial values
inits <- function () {
    list(beta = matrix(rnorm(6,0, 0.01),ncol=2))
}

# Run mcmc
jags_fit <- jags(data= jags_data,
                  inits = inits,
                  parameters = params,
                  model.file = textConnection(jags_model),
                  n.chains = 3,
                  n.thin = 10,
                  n.iter = 5*10^4,
                  n.burnin = 2*10^4
)
=====

# Output
print(jags_fit,intervals=c(0.025, 0.975),
      digits=3)
=====
```

An example of the output of the JAGS model is shown below.

```

Inference for Bugs model, fit using jags,
3 chains, each with 50000 iterations
          mu.vect sd.vect 2.5% 97.5%
beta[1,1]   0.049   0.088 -0.112  0.238
beta[2,1]  -0.169   0.094 -0.356  0.003
beta[3,1]   0.197   0.115 -0.010  0.428
beta[1,2]   0.003   0.052 -0.100  0.107
beta[2,2]  -0.024   0.052 -0.130  0.075
beta[3,2]   0.006   0.054 -0.099  0.113
```

To visualize how the model fits the data, we display in Figure 10.16 the predicted probabilities f_{Seyfert} as a function of r/r_{200} for halos with an average mass $\log M_{200} \approx 10^{14} M_\odot$. We present the binned data, for illustrative purposes, and the fitted model and uncertainty. The shaded areas represent the 50% and 95% probability intervals. It should be noted that the fitting was performed without making use of any data binning.

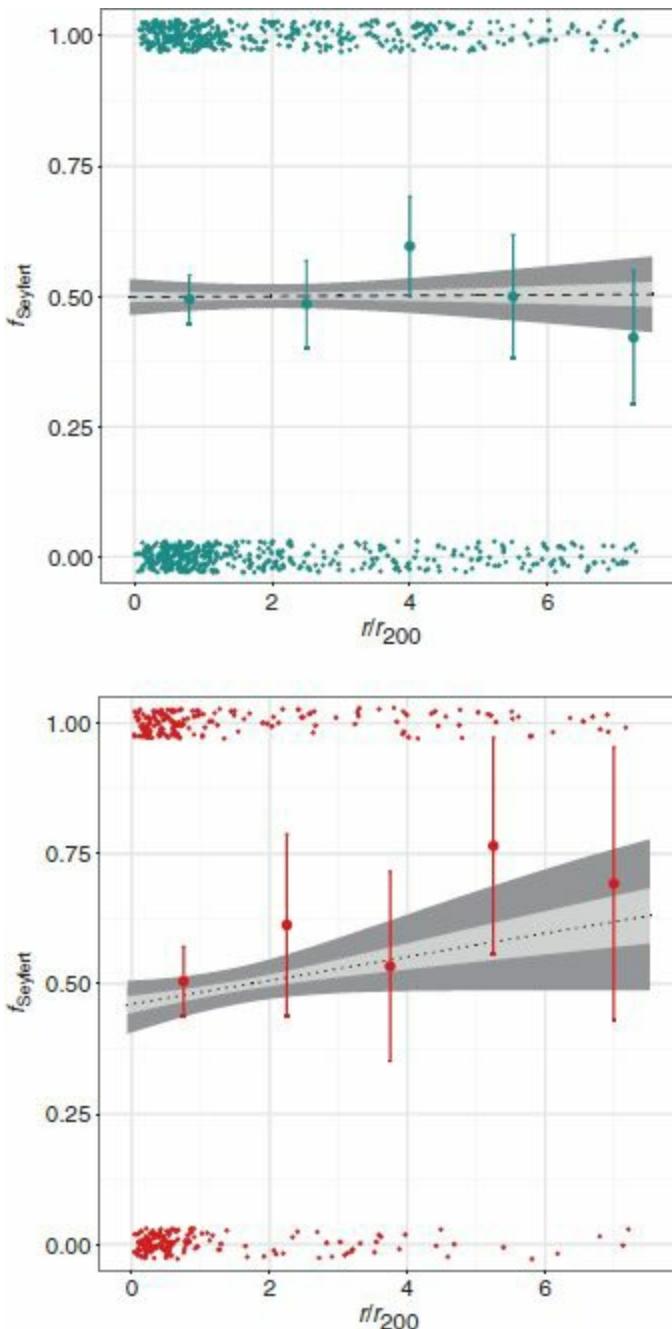


Figure 10.16 Two-dimensional representation of the six-dimensional parameter space describing the dependence of Seyfert AGN activity as a function of r/r_{200} and $\log M_{200}$, for clusters with an average $\log M_{200} \approx 10^{14} M_\odot$: upper panel, spirals; lower panel, elliptical galaxies. In each panel the lines (dashed or dotted) represent the posterior mean

probability of Seyfert AGN activity for each value of r/r_{200} , while the shaded areas depict the 50% and 95% credible intervals. The data points with error bars represent the data when binned, for purely illustrative purposes.

The coefficients for the logit model, whose posterior distribution is displayed in Figure 10.17, represent the log of the odds ratio for Seyfert activity; thus one unit variation in r/r_{200} towards the cluster outskirts for an elliptical galaxy residing in a cluster with an average mass $\log M_{200} = 14$ produces on average a change of 0.197 in the log of the odds ratio of Seyfert activity or, in other words, it is **21.7%** more likely to be a Seyfert galaxy as we move farther from the center. Unlike elliptical galaxies, however, spirals are virtually unaffected by their position inside the cluster or by the mass of their host, all the fitted coefficients being consistent with zero.

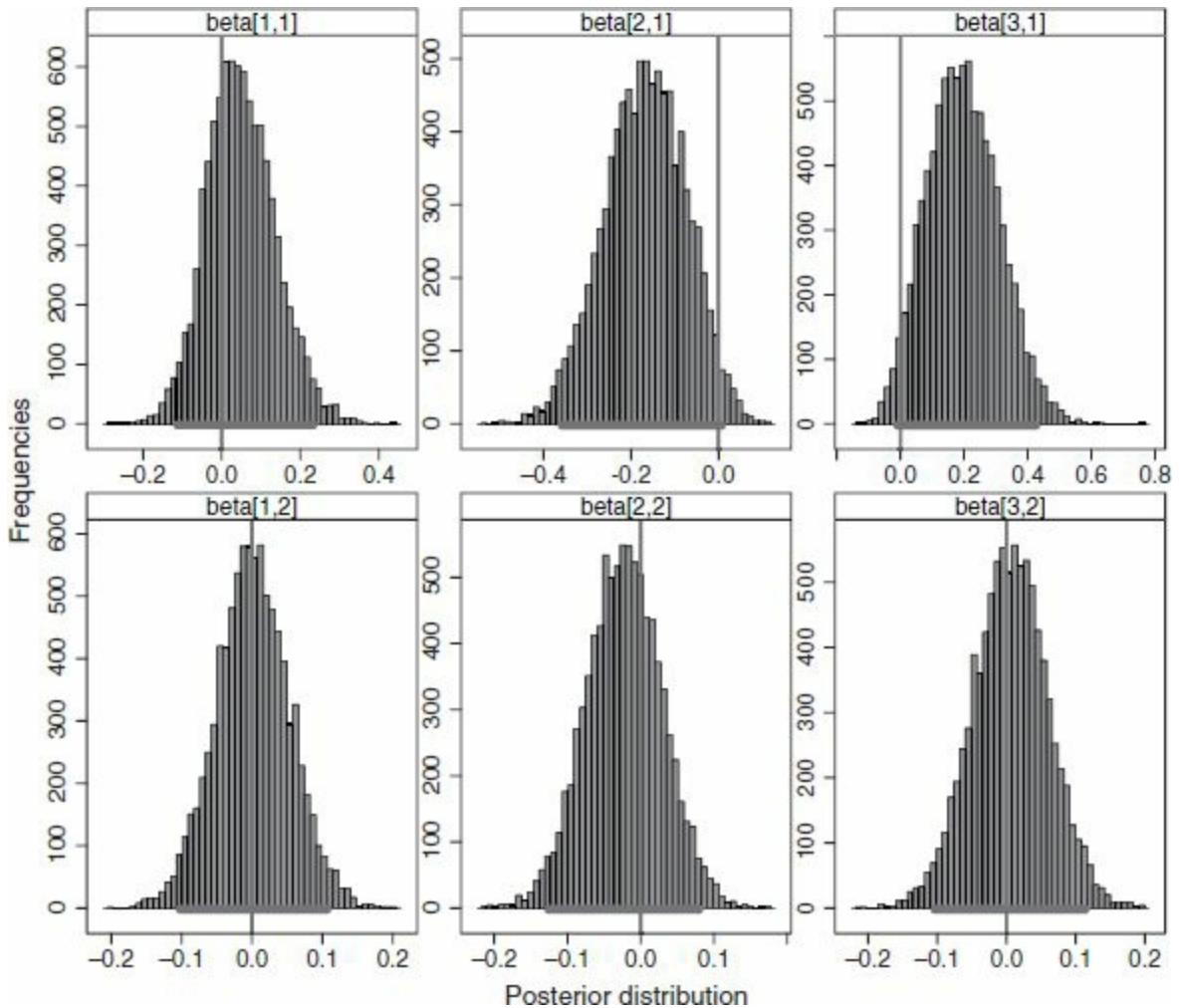


Figure 10.17 Computed posterior for the β coefficients of our model. From left to right: intercept, $\log M_{200}$, and r/r_{200} for elliptical ($j = 1$, upper panels) and spiral ($j = 2$, lower panels) galaxies respectively.

10.8.4 Running the Model in Python using Stan

Notice that here again the logit link is built into the Bernoulli implementation in Stan, so there is no

need to include it manually. Also, in order to facilitate comparison with the results from R/JAGS, we use the `dot_product` function so the configurations of matrices are compatible. This step is not necessary unless you are performing such comparisons.

Code 10.19 Bernoulli logit model, in Python using Stan, for assessing the relationship between Seyfert AGN activity and galactocentric distance.

```
=====
import numpy as np
import pandas as pd
import pystan
import statsmodels.api as sm

# Data
path_to_data = '~/data/Section_10p8/Seyfert.csv'

# Read data
data_frame = dict(pd.read_csv(path_to_data))

x1 = data_frame['logM200']
x2 = data_frame['r_r200']

data = {}
data['Y'] = data_frame['bpt']
data['X'] = sm.add_constant(np.column_stack((x1,x2)))
data['K'] = data['X'].shape[1]
data['N'] = data['X'].shape[0]
data['gal'] = [0 if item == data_frame['zoo'][0] else 1
              for item in data_frame['zoo']]
data['P'] = 2

# Fit
# Stan model
stan_code=""""
data{
    int<lower=0> N;                      # number of data points
    int<lower=0> K;                      # number of coefficients
    int<lower=0> P;                      # number of populations
    matrix[N,K] X;                      # [logM200, galactocentric distance]
    int<lower=0, upper=1> Y[N];          # Seyfert 1/AGN 0
    int<lower=0, upper=1> gal[N];        # elliptical 0/spiral 1
}
parameters{
    matrix[K,P] beta;
    real<lower=0> sigma;
    real mu;
}
model{
    vector[N] pi;

    for (i in 1:N) {
        if (gal[i] == gal[1]) pi[i] = dot_product(col(beta,1),X[i]);
        else pi[i] = dot_product(col(beta,2), X[i]);
    }

    # shared hyperpriors
    sigma ~ gamma(0.001, 0.001);
    mu ~ normal(0, 100);

    # priors and likelihood
    for (i in 1:K) {
        for (j in 1:P) beta[i,j] ~ normal(mu, sigma);
    }
    Y ~ bernoulli_logit(pi);
}
"""
# Run mcmc
fit = pystan.stan(model_code=stan_code, data=data, iter=60000, chains=3,
                   warmup=30000, thin=10, n_jobs=3)

# Output
print(fit)
=====
```

mean	se_mean	sd	2.5%	97.5%	n_eff	Rhat
------	---------	----	------	-----	-----	-----	-------	-------	------

beta[0,0]	0.04	2.2e-3	0.09	-0.11	0.23	1543.0	1.0
beta[1,0]	-0.15	2.9e-3	0.1	-0.36	0.01	1115.0	1.0
beta[2,0]	0.17	5.0e-3	0.12	-0.05	0.42	595.0	1.01
beta[0,1]	9.4e-4	9.4e-4	0.05	-0.1	0.1	2889.0	1.0
beta[1,1]	-0.02	9.3e-4	0.05	-0.12	0.08	2908.0	1.0
beta[2,1]	3.9e-3	1.0e-3	0.05	-0.1	0.11	2745.0	1.0
sigma	0.14	5.7e-3	0.09	0.02	0.37	265.0	1.0
mu	8.0e-3	8.2e-4	0.08	-0.14	0.17	9000.0	1.0

10.9 Lognormal–Logit Hurdle Model and the Halo–Stellar-Mass Relation

We explore now the more subtle problem of considering two separate components in the construction of our statistical model. The so-called hurdle models form a class of zero-altered models where a Bernoulli distribution is used to describe the likelihood of an event which triggers a significant change in the system state (a success). Once this occurs, another distribution is then used to describe the behavior of non-zero elements in the data set (see Section 7.2 for a deeper discussion and examples using synthetic data). This is exactly the data situation an astronomer faces when trying to model the correlation between dark-matter halos and stellar mass in cosmological simulations of the early Universe. A minimum halo mass is necessary to trigger star formation in galaxies, an event that completely changes the configuration of the system and its subsequent development. We will explore this particular example through the application of a lognormal–logit hurdle model to data from a cosmological hydro-simulation.

The standard model of cosmology predicts a structure formation scenario driven by cold dark matter (e.g., [Benson, 2010](#)), where galaxies form from molecular gas cooling within growing dark-matter halos. Hence, understanding the correlation between different properties of the dark-matter halos is of the utmost importance in building up a comprehensive picture of galaxy evolution. Many authors have explored the correlation between dark-halo and baryonic properties, in both the low-redshift regime (e.g., [Bett et al., 2007](#); [Hahn et al., 2007](#); [Macciò et al., 2007](#); [Wang et al., 2011](#)) and high-redshift regimes (e.g., [de Souza et al., 2013](#); [Jang-Condell and Hernquist, 2001](#)). Estimating the strength of these correlations is critical for supporting semi-analytical models, which assume the mass as determinant factor of the halo properties (e.g., [Somerville et al., 2008](#)). In this application we extend the work of [de Souza et al. \(2015a\)](#), who applied a simple Bernoulli model to probe the odds of star-formation activity as a function of the halo properties. Here, we account fully for two independent processes: the minimum halo mass to host star-formation activity and the correlation between halo mass and stellar mass content, once the halo is capable of hosting stars. At the time of writing, this is the first application of hurdle models in astronomy.

10.9.1 Data

The data set used in this work was retrieved from a cosmological hydro-simulation based on [Biffi and Maio \(2013\)](#) (see also [de Souza et al., 2014](#); [Maio et al., 2010, 2011](#)). The simulations have snapshots in the redshift range $9 \lesssim z \lesssim 19$ for a cubic volume of comoving side ~ 0.7 Mpc, sampled with 2×320^3 particles per gas and dark-matter species. The simulation output considered in this work comprises $N = 1680$ halos in the whole redshift range, with about 200

objects at $z = 9$. The masses of the halos are in the range $10^5 M_\odot \lesssim M_{\text{dm}} \lesssim 10^8 M_\odot$, with corresponding stellar masses $0 < M_\star \lesssim 10^4 M_\odot$.

10.9.2 The Statistical Model Formulation

We chose a lognormal–logit hurdle model for this example so that we could take into account the binary and continuous part of the relationship between the dark halo and stellar masses. The model assumes that something triggers the presence of stars, so the halo crosses the *hurdle*. The second mechanism connecting the halo mass to the stellar mass is modeled by another distribution that does not allow zeros, for which we choose the lognormal. In other words: the Bernoulli process is causing the presence or absence of stars in the halo; once stars are present, their abundance is described by a lognormal distribution.

In this context the stellar mass represents our response variable M_\star , and the dark-matter halo is our explanatory variable M_{dm} . Given that there are now two different underlying statistical distributions, we will also have two distinct linear predictors (each one bearing $K = 2$ coefficients). In the model in Equation 10.17 below, $\{\beta_1, \beta_2\}$ and $\{\gamma_1, \gamma_2\}$ are the linear predictor coefficients for the Bernoulli and lognormal distributions, respectively. The complete model can be expressed as follows:

$$\begin{aligned}
 M_{\star;i} &\sim \begin{cases} \text{Bernoulli}(p_i) & \text{if } M_{\star;i} = 0 \\ \text{LogNormal}(\mu_i, \sigma^2) & \text{otherwise} \end{cases} \\
 \text{logit}(p_i) &= \gamma_1 + \gamma_2 M_{\text{dm};i} \\
 \mu_i &= \beta_1 + \beta_2 M_{\text{dm};i} \\
 \beta_j &\sim \text{Normal}(0, 10^3) \\
 \gamma_j &\sim \text{Normal}(0, 10^3) \\
 \sigma &\sim \text{Gamma}(0.001, 0.001) \\
 i &= 1, \dots, N \\
 j &= 1, \dots, K
 \end{aligned} \tag{10.17}$$

where σ denotes the lognormal scatter parameter.

10.9.3 Running the Model in R using JAGS

The implementation of the lognormal–logit hurdle model in JAGS is given below:

```

mass.

=====
require(R2jags)

# Data
dataB <- read.csv("~/data/Section_10p9/MstarZSFR.csv", header = T)
hurdle <- data.frame(x = log(dataB$Mdm,10), y = asinh(1e10*dataB$Mstar))

# prepare data for JAGS
Xc <- model.matrix(~ 1 + x, data = hurdle)
Xb <- model.matrix(~ 1 + x,
                   data = hurdle)

Kc <- ncol(Xc)
Kb <- ncol(Xb)

JAGS.data <- list(
  Y = hurdle$y,           # response
  Xc = Xc,               # covariates
  Xb = Xb,               # covariates
  Kc = Kc,               # number of betas
  Kb = Kb,               # number of gammas
  N = nrow(hurdle),      # sample size
  Zeros = rep(0, nrow(hurdle)))

# Fit
load.module('glm')
sink("ZAPGLM.txt")
cat("
model{
# 1A. Priors beta and gamma
for (i in 1:Kc) {beta[i] ~ dnorm(0, 0.0001)}
for (i in 1:Kb) {gamma[i] ~ dnorm(0, 0.0001)}

# 1C. Prior for r parameter
sigmaLN ~ dgamma(1e-3, 1e-3)
# 2. Likelihood (zero trick)
C <- 1e10
for (i in 1:N) {
  Zeros[i] ~ dpois(-ll[i] + C)

ln1[i] <- -(log(Y[i]) + log(sigmaLN)+log(sqrt(2*sigmaLN)))
ln2[i] <- -0.5*pow((log(Y[i])-mu[i]),2)/(sigmaLN*sigmaLN)
LN[i] <- ln1[i]+ln2[i]

z[i] <- step(Y[i] - 1e-5)
l1[i] <- (1 - z[i]) * log(1 - Pi[i])
l2[i] <- z[i] * ( log(Pi[i]) + LN[i])
ll[i] <- l1[i] + l2[i]

mu[i] <- inprod(beta[], Xc[i,])
logit(Pi[i]) <- inprod(gamma[], Xb[i,])
}
}

", fill = TRUE)
sink()

# Define initial values
inits <- function () {
  list(beta = rnorm(Kc, 0, 0.1),
       gamma = rnorm(Kb, 0, 0.1),
       sigmaLN = runif(1, 0, 10) )}

# Identify parameters
params <- c("beta", "gamma", "sigmaLN")

# Run MCMC
H1 <- jags(data = JAGS.data,
            inits = inits,
            parameters = params,
            model = "ZAPGLM.txt",
            n.thin = 1,
            n.chains = 3,
            n.burnin = 5000,
            n.iter = 15000)

# Output
print(H1,intervals=c(0.025, 0.975), digits=3)

```

	mu.vect	sd.vect	2.5%	97.5%	Rhat	n.eff
beta[1]	-2.1750e+00	0.439	-3.0590e+00	-1.3080e+00	1.047	60
beta[2]	5.8900e-01	0.063	4.6500e-01	7.1600e-01	1.038	66
gamma[1]	-5.3748e+01	4.065	-6.2322e+01	-4.6319e+01	1.001	32000
gamma[2]	7.9080e+00	0.610	6.7900e+00	9.1930e+00	1.001	26000
sigmaLN	2.2100e-01	0.012	2.0000e-01	2.4600e-01	1.001	7900
deviance	3.3600e+13	3.222	3.3600e+13	3.3600e+13	1.000	1

Figure 10.18 shows the posterior distributions for our model. The resulting fit for the relationship between M_* and M_{dm} , and the data, can be seen in Figure 10.19. The shaded areas represent the 50% (darker) and 95% (lighter) credible intervals around the mean, represented by the dashed line. To illustrate the different parts of the process, we also show an illustrative set of layers in Figure 10.20. From bottom to top this figure shows the original dataset, the logistic part of the fit, that provides the probability of stellar presence given a certain value of M_{dm} ; the fitted lognormal model of the positive continuous part, and finally the full model is displayed again in the upper panel.

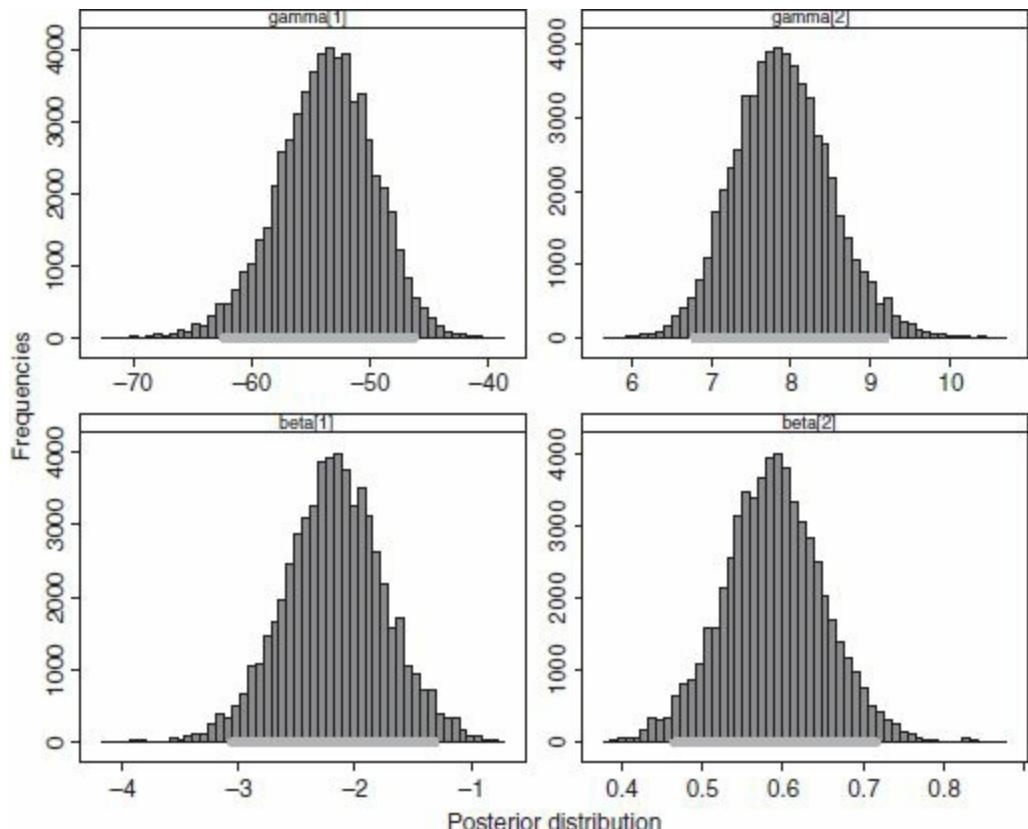


Figure 10.18 Computed posteriors for the β and γ coefficients of our lognormal–logit hurdle model. The horizontal thick line at the bottom of each histogram defines the 95% credible interval.

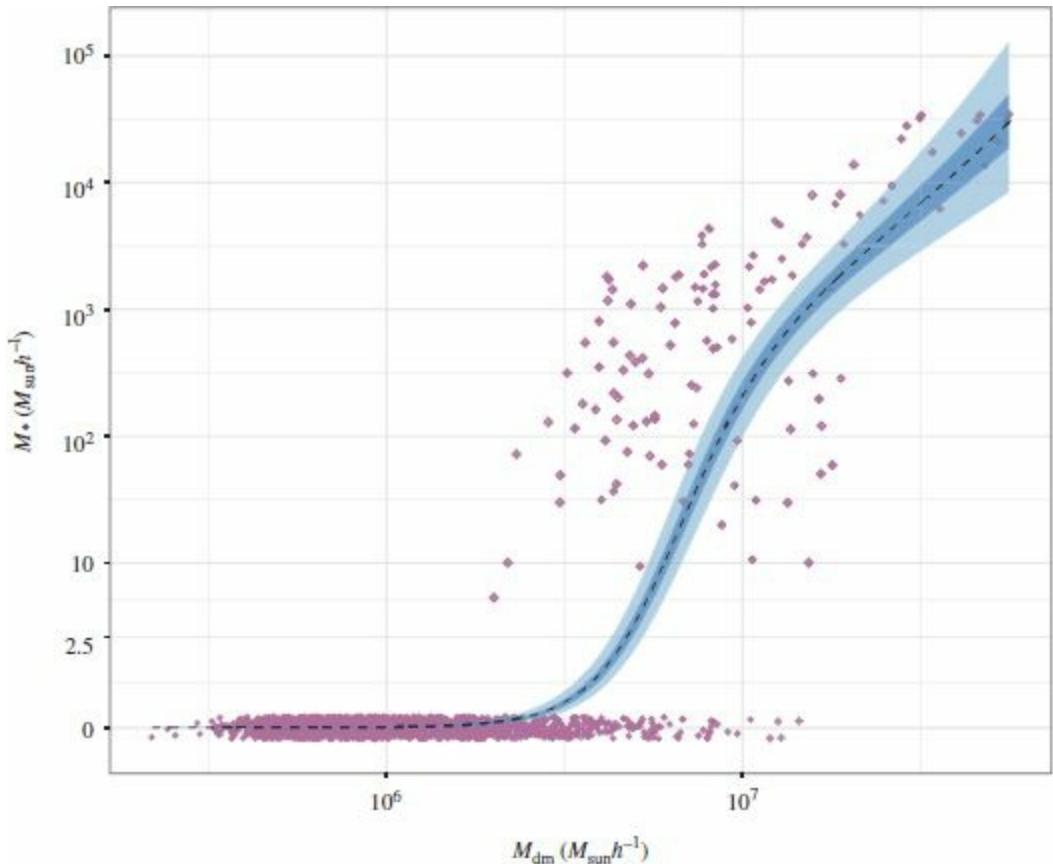


Figure 10.19 Fitted values for the dependence between stellar mass M_{\star} and dark-matter halo mass M_{dm} resulting from the lognormal–logit hurdle model. The dashed line represents the posterior mean stellar mass for each value of M_{dm} , while the shaded areas depict the 50% (darker) and 95% (lighter) credible intervals around the mean; M_{sun} is the mass of the Sun and h is the dimensionless Hubble parameter.

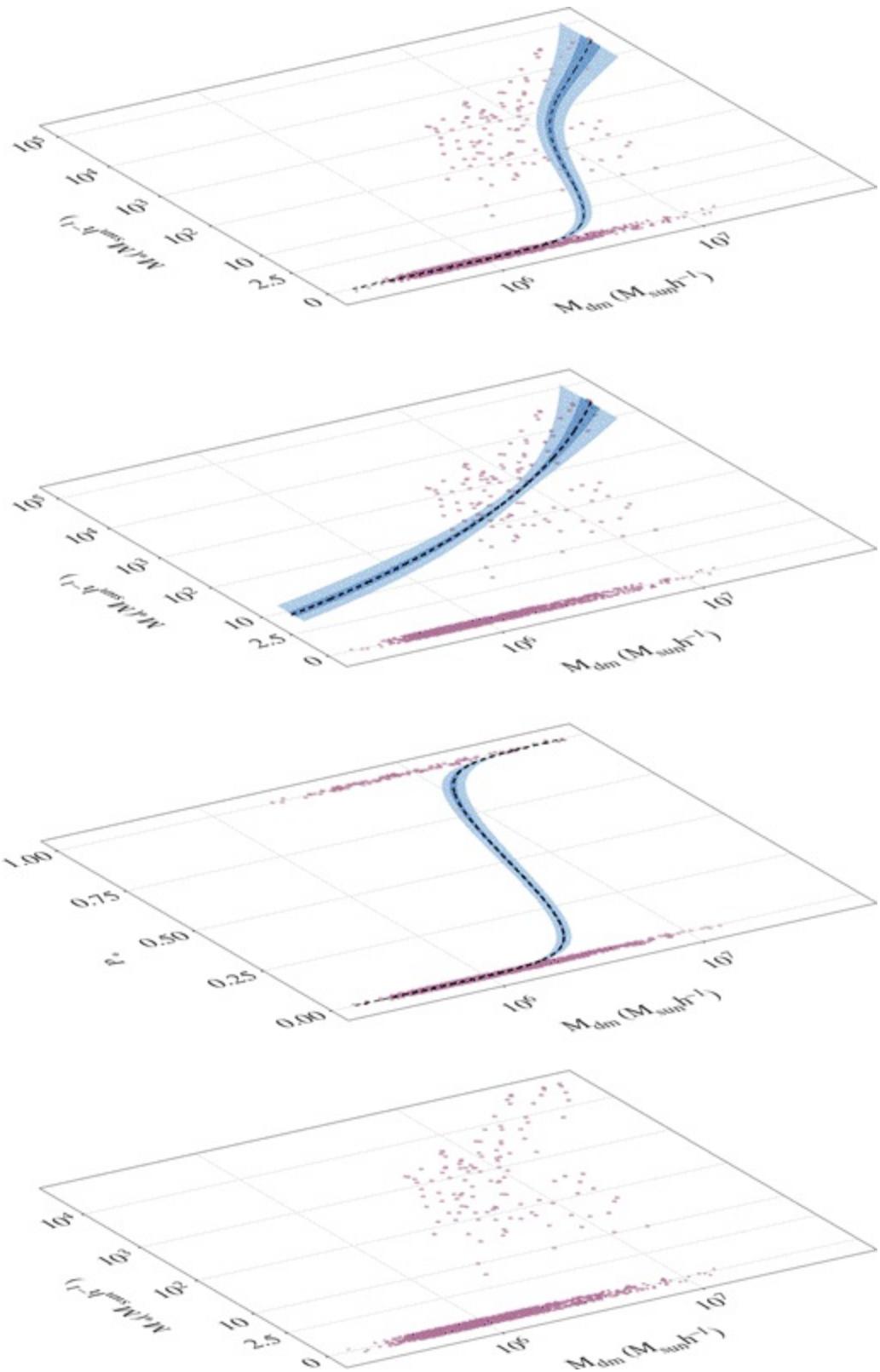


Figure 10.20 Illustration of the different layers in the lognormal–logit hurdle model. From bottom to top: the original data; the Bernoulli fit representing the presence or absence of stars in the halo; the lognormal fit of the positive part (i.e., ignoring the zeros in M_{\star}); and the full model.

10.9.4 Running the Model in Python using Stan

The implementation in Python using Stan follows the guidelines presented in [Team Stan \(2016, Section 10.5\)](#). Stan has built-in tools which allows a concise representation for zero-inflated and hurdle models, as discussed in Section [7.2.4](#).

Code 10.21 Lognormal–logit hurdle model, in Python using Stan, for assessing the relationship between dark halo mass and stellar mass.

```
=====
import numpy as np
import pandas as pd
import pystan
import statsmodels.api as sm

# Data
path_to_data = '~/data/Section_10p9/MstarZSFR.csv'

# Read data
data_frame = dict(pd.read_csv(path_to_data))

# Prepare data for Stan
y = np.array([np.arcsinh(10**10*item) for item in data_frame['Mstar']])
x = np.array([np.log10(item) for item in data_frame['Mdm']])

data = {}
data['Y'] = y
data['Xc'] = sm.add_constant(x.transpose())
data['Xb'] = sm.add_constant(x.transpose())
data['Kc'] = data['Xc'].shape[1]
data['Kb'] = data['Xb'].shape[1]
data['N'] = data['Xc'].shape[0]

# Fit
# Stan model
stan_code=""""
data{
    int<lower=0> N;                      # number of data points
    int<lower=0> Kc;                     # number of coefficients
    int<lower=0> Kb;
    matrix[N,Kb] Xb;                    # dark matter halo mass
    matrix[N,Kc] Xc;
    real<lower=0> Y[N];                  # stellar mass
}
parameters{
    vector[Kc] beta;
    vector[Kb] gamma;
    real<lower=0> sigmaLN;
}
model{
    vector[N] mu;
    vector[N] Pi;

    mu = Xc * beta;
    for (i in 1:N) Pi[i] = inv_logit(Xb[i] * gamma);

    # Priors and likelihood
    for (i in 1:Kc) beta[i] ~ normal(0, 100);
    for (i in 1:Kb) gamma[i] ~ normal(0, 100);
    sigmaLN ~ gamma(0.001, 0.001);

    for (i in 1:N) {
        (Y[i] == 0) ~ bernoulli(Pi[i]);
        if (Y[i] > 0) Y[i] ~ lognormal(mu[i], sigmaLN);
    }
}
"""
# Run mcmc
fit = pystan.stan(model_code=stan_code, data=data, iter=15000, chains=3,
                   warmup=5000, thin=1, n_jobs=3)

# Output
```

```

print(fit)
=====

      mean   se_mean    sd   2.5%   25%   50%   75% 97.5% n_eff Rhat
beta[0] -2.1  7.3e-3  0.53 -3.15 -2.46 -2.11 -1.75 -1.07 5268.0 1.0
beta[1]  0.5  1.1e-3  0.08  0.43  0.53  0.58  0.63  0.73 5270.0 1.0
gamma[0] 54.61  0.06   4.35 46.65 51.63 54.45 57.36 63.66 4972.0 1.0
gamma[1] -8.04 9.2e-3  0.65 -9.39 -8.45 -8.01 -7.59 -6.84 4973.0 1.0
sigmaLN  0.27 2.3e-4  0.02  0.24  0.26  0.27  0.28  0.31 5943.0 1.0

```

10.10 Count Time Series and Sunspot Data

We now take one step further from the classical generalized linear models (GLMs) presented previously. The goal of this section is to show how the framework of GLMs can be used to deal with a special data type, time series. A time series is characterized by a set of measurements of the same event or experiment taken sequentially over time. This is fundamentally different from the situation in previous examples, where the order of the observed points was irrelevant. In a time series, time itself drives causality and builds structure within the data. The state of the system at a given time imposes constraints on its possible states in the near future and has a non-negligible influence on the states further ahead.

In astronomy, this situation is common. The whole transient sky is studied by means of light curves (measurement of brightness over time), which are essentially examples of time series. Despite the fact that frequently, a non-homogeneous time separation between consecutive observations imposes some extra complexity, this does not invalidate the potential application of such models in an astronomical context.²³ Moreover, examples such as the one shown here below fulfill all the basic requirements for time series analysis and illustrate the potential of GLMs to deal with this type of data.

In what follows we will use the time series framework to model the evolution of the number of sunspots over time. The appearance or disappearance of black regions in the Sun's surface (sunspots) has been reported since ancient times ([Vaquero, 2007](#)). Their emergence is connected with solar magnetic activity and is considered to be a precursor of more drastic events such as solar flares and coronal mass ejections. Careful observation of the number of sunspots over time has revealed the existence of a solar cycle of approximately 11 years, which is correlated with extreme emissions of ultraviolet and X-ray radiation. Such bursts can pose significant concerns for astronauts living in space, airline travelers in polar routes, and engineers trying to optimize the lifetime of artificial satellites ([Hathaway, 2015](#)).

Here we show how the classic autoregressive (AR) model (see e.g. the book, [Lunn Jackson, Best *et al.*, 2012](#), for examples of Bayesian time series), which typically assumes the residuals to be normally distributed, can be modified to fit a count time series (the number of sunspots is a discrete response variable). Considering the number of sunspots in the y th year, $N_{\text{spots};y}$, the standard normal AR model states that $N_{\text{spots};y}$ depends linearly on its own value in previous years and on a stochastic term, σ . This can be represented as:

$$N_{\text{spots};y} \sim \text{Normal}(\mu_t, \sigma^2) \quad (10.18)$$

$$\mu_y = \phi_1 + \sum_{y=2}^p \phi_y N_{\text{spots};y-1}$$

where μ_y can be identified as the linear predictor for year y , having $\{\phi_1, \phi_2\}$ as its coefficients, and p represents the order of the AR(p) process. The latter quantifies how the influence of previous measurements on the current state of the system fades with time. A low p means that the system only “remembers” recent events and a high p indicates that even remote times can influence current and future system developments.

Here we will consider a simple case, the AR(1) model,

$$N_{\text{spots};y} \sim \text{Normal}(\mu_t, \sigma^2)$$

$$\mu_y = \phi_1 + \phi_2 N_{\text{spots};y-1}$$

(10.19)

applied to the annual sunspot-number data set, but see [Chattopadhyay and Chattopadhyay \(2012\)](#) for an application of an AR(3) model to the monthly sunspot-number time series.

10.10.1 Data

We will use the mean annual data for the International Sunspot number,²⁴ under the responsibility of the Royal Observatory in Brussels²⁵ since 1980 ([Feehrer, 2000](#)). This is considered the “official” number of sunspots by the International Astronomical Union (IAU). The Sunspot Index Data Center (SIDC), hosted by the Royal Observatory in Brussels, provides daily, monthly, and annual numbers, as well as predictions of sunspot activities. The observations are gathered by a team of amateur and professional astronomers working at 40 individual stations.

10.10.2 Running the Normal AR(1) Model in R using JAGS

As a first exercise we applied the normal AR(1) model to the sunspot data in order to evaluate its efficiency. We do not expect this model to show an ideal performance, since it considers the residuals as normally distributed, which does not agree with the data set at hand (the count data). Our goal is to access the extent of its applicability. Subsequently we will show how to apply a statistical model that is more suitable for this data situation.

Code 10.22 Normal autoregressive model AR(1) for accessing the evolution of the number of sunspots through the years.

```
=====
require(R2jags)
require(jagstools)

# Data

# Read data
sunspot <- read.csv("~/data/Section_10p10/sunspot.csv", header = T, sep=", ")

# Prepare data to JAGS
y <- round(sunspot[,2])
t <- seq(1700, 2015, 1)
N <- length(y)

sun_data <- list(Y = y,      # Response variable
                  N = N)    # Sample size
                  )

# Fit

AR1_NORM <- "model{
  # Priors
  sd2 ~ dgamma(1e-3,1e-3)

  tau <- 1/sd2
  sd <- sqrt(sd2)
  for(i in 1:2){
    phi[i] ~ dnorm(0,1e-2)
```

```

}

mu[1] <- Y[1]

# Likelihood function
for (t in 2:N) {
  Y[t] ~ dnorm(mu[t],tau)
  mu[t] <- phi[1] + phi[2] * Y[t-1]
}

# Prediction
for (t in 1:N){
  Yx[t]~dnorm(mu[t],tau)
}

# Generate initial values for mcmc
inits <- function () {
  list(phi = rnorm(2, 0, 0.1))
}

# Identify parameters
# Include Yx only if you intend to generate plots
params <- c("sd", "phi", "Yx")

# Run mcmc
jagsfit <- jags(data = sun_data,
  inits = inits,
  parameters = params,
  model = textConnection(AR1_NORM),
  n.thin = 1,
  n.chains = 3,
  n.burnin = 3000,
  n.iter = 5000)

# Output
print(jagsfit,intervals = c(0.025, 0.975),justify = "left", digits=2)
=====

      mu.vect    sd.vect    2.5%    97.5%   Rhat   n.eff
phi[1]     13.47     3.12     7.26    19.65     1    6000
phi[2]      0.83     0.03     0.76     0.89     1    6000
sd        35.79     1.43    33.18    38.78     1    3600
deviance  3150.06     2.50   3147.20   3156.59     1     980

pD = 3.1 and DIC = 3153.2

```

In order to visualize how the model fits the data, the reader can ask JAGS to monitor the parameter y_x and then extract the values with the function `jagsresults`. The fitted model is shown in Figure 10.21 and the code to reproduce the figure is given below.

Code 10.23 Model shown in Figure 10.21.

```

=====
require(ggplot2)
require(jagstools)
# Format data for ggplot
sun <- data.frame(x=t,y=y)                      # original data
yx <- jagsresults(x=jagsfit, params=c('Yx'))       # fitted values
gdata <- data.frame(x = t, mean = yx[, "mean"],lwr1=yx[, "25%"],
lwr2=yx[, "2.5%"],upr1=yx[, "75%"],upr2=yx[, "97.5%"])
# Plot
ggplot(sun,aes(x=t,y=y))+
  geom_ribbon(data=gdata,aes(x=t,ymin=lwr1, ymax=upr1,y=NULL),
alpha=0.95, fill=c("#969696"),show.legend=FALSE)+
  geom_ribbon(data=gdata,aes(x=t,ymin=lwr2, ymax=upr2,y=NULL),
alpha=0.75, fill=c("#d9d9d9"),show.legend=FALSE)+
  geom_line(data=gdata,aes(x=t,y=mean),colour="black",
linetype="solid",size=0.5,show.legend=FALSE)+
  geom_point(colour="orange2",size=1.5,alpha=0.75)+
  theme_bw() + xlab("Year") + ylab("Sunspots") +
  theme(legend.background = element_rect(fill="white"),
  legend.key = element_rect(fill = "white",color = "white"),
  plot.background = element_rect(fill = "white"),
  legend.position="top",
  axis.title.y = element_text(vjust = 0.1,margin=margin(0,10,0,0))),
```

```

axis.title.x = element_text(vjust = -0.25),
text = element_text(size = 25,family="serif"))+
geom_hline(aes(yintercept=0),linetype="dashed",colour="gray45",size=1.25)

```

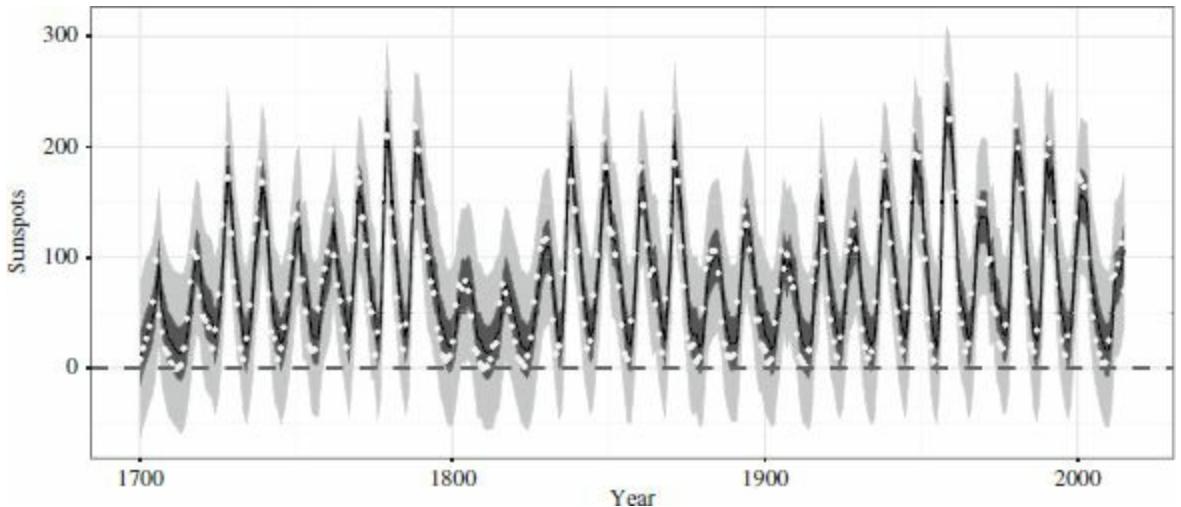


Figure 10.21 Normal AR(1) model fit for sunspot data. The solid line represents the posterior mean and the shaded areas are the 50% (darker) and 95% (lighter) credible intervals; the dots are the observed data. The dashed horizontal line represents a barrier which should not be crossed, since the number of sunspots cannot be negative.

The model fits the data reasonably well (Figure 10.21), given its simplicity, but it allows negative values of sunspot number (the shaded areas do not respect the barrier represented by the dashed line); this is expected given that a model with normal residuals does not impose any restrictions in this regard. Now we show how to change the classical AR model to allow other distributions, in the hope this can serve as a template for the reader to try more complex models on his or her own. Let us try a negative binomial AR(1) model, which can be expressed as follows:

$$\begin{aligned}
N_{\text{spots};y} &\sim NB(p_y, \theta) \\
p_y &= \theta / (\theta + \mu_y) \\
\log(\mu_y) &= \phi_1 + \phi_2 N_{\text{spots};y-1}
\end{aligned} \tag{10.20}$$

10.10.3 Running the Negative Binomial AR Model in R using JAGS

Code 10.24 Negative binomial model (AR1) for assessing the evolution of the number of sunspots through the years.

```

=====
# Fit
AR1_NB<- "model{

  for(i in 1:2){
    phi[i]~dnorm(0,1e-2)
  }
  theta~dgamma(0.001,0.001)
  mu[1] <- Y[1]

  # Likelihood function
  for (t in 2:N) {

```

```

Y[t] ~ dnegbin(p[t],theta)
p[t] <- theta/(theta+mu[t])
log(mu[t]) <- phi[1] + phi[2]*Y[t-1]
}

for (t in 1:N){
  Yx[t] ~ dnegbin(px[t],theta)
  px[t] <- theta/(theta+mu[t])
}
}"
```

Identify parameters
Include Yx only if interested in prediction
params <- c("phi","theta", "Yx")

Generate initial values for mcmc
inits <- function(){
 list(phi=rnorm(2,0,0,1))
}

Run mcmc
jagsfit <- jags(data = sun_data,
 inits = inits,
 parameters = params,
 model = textConnection(AR1_NB),
 n.thin = 1,
 n.chains 3,
 n.burnin 5000,
 n.iter = 10000)

Output
print(jagsfit,intervals=c(0.025, 0.975),justify = "left", digits=2)
=====

	mu.vect	sd.vect	2.5%	97.5%	Rhat	n.eff
phi[1]	3.331	0.063	3.208	3.453	1.001	3200
phi[2]	0.011	0.001	0.009	0.012	1.001	5900
theta	2.550	0.212	2.160	2.977	1.001	4600
deviance	3163.126	2.430	3160.354	3169.398	1.001	7500

pD = 3.0 and DIC = 3166.1

Figure 10.22 illustrates the results for the NB AR(1) model. Note that the model now respects the lower limit of zero counts, but the 95% credible intervals overestimate the counts. It is important to emphasize that there are several other models that should be tested in order to decide which one suits this particular data best. We did not intend to provide a comprehensive overview of time series in the section but instead to provide the reader with a simple example of how to implement different statistical distributions in classical time series models.

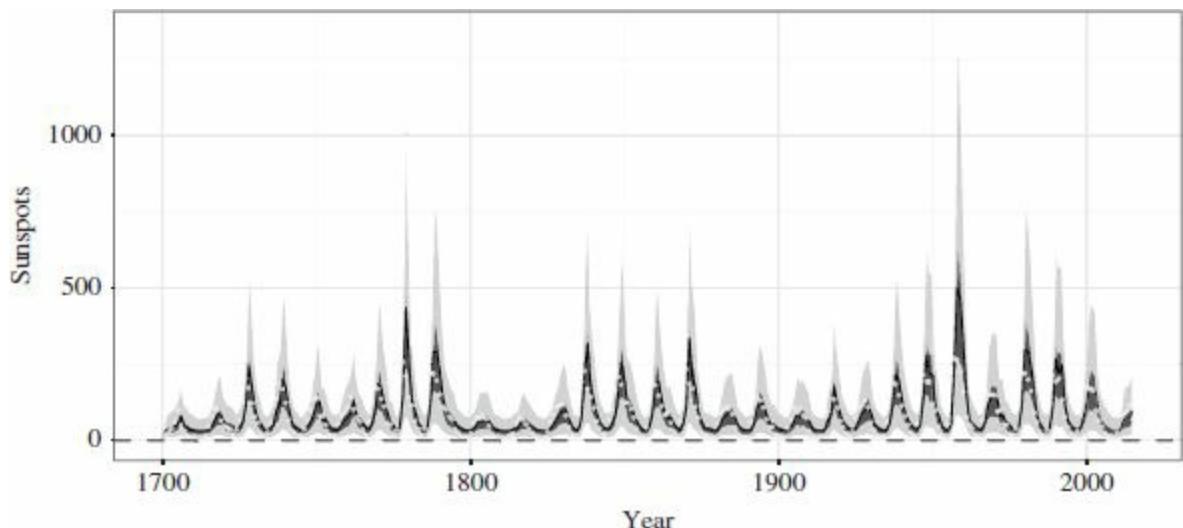


Figure 10.22 Negative binomial AR(1) model fit for sunspot data. The solid line represents the posterior mean and the shaded areas are the 50% and 95% credible intervals; the dots are the observed data.

10.10.4 Running the Negative Binomial AR Model in Python using Stan

The implementation in Python using Stan follows the same lines. We just need to emphasize that Stan has a set of different parameterizations for the negative binomial distribution and, as a consequence, we can use the `neg_binomial_2` function, which does not require the definition of the parameter p .

We show here only the implementation of the negative binomial AR(1) model, since it is not so easy to find in the literature. For a clear description of how to implement the normal AR(1) model in Stan, see [Team Stan \(2016, Part II, Chapter 7\)](#).

Code 10.25 Negative binomial model (AR1) in Python using Stan, for assessing the evolution of the number of sunspots through the years.

```
=====
import numpy as np
import pandas as pd
import pystan
import statsmodels.api as sm

# Data
path_to_data = "~/data/Section_10p10/sunspot.csv"

# Read data
data_frame = dict(pd.read_csv(path_to_data))

# Prepare data for Stan
data = {}
data['Y'] = [int(round(item)) for item in data_frame['nspots']]
data['nobs'] = len(data['Y'])
data['K'] = 2

# Fit
# Stan model
stan_code=""""
data{
    int<lower=0> nobs;                      # number of data points
    int<lower=0> K;                          # number of coefficients
    int Y[nobs];                            # nuber of sunspots
}
parameters{
    vector[K] phi;                         # linear predictor coefficients
    real<lower=0> theta;                   # noise parameter
}
model{
    vector[nobs] mu;
    mu[1] = Y[1];                           # set initial value
    for (t in 2:nobs) mu[t] = exp(phi[1] + phi[2] * Y[t - 1]);
    # Priors and likelihood
    theta ~ gamma(0.001, 0.001);
    for (i in 1:K) phi[i] ~ normal(0, 100);
    Y ~ neg_binomial_2(mu, theta);
}
"""

# Run mcmc
fit = pystan.stan(model_code=stan_code, data=data, iter=7500, chains=3,
                   warmup=5000, thin=1, n_jobs=3)

# Output
```

```

print(fit)
=====
      mean se_mean    sd  2.5%  25%  50%  75% 97.5% n_eff Rhat
phi[0]   3.33  1.6e-3  0.06  3.22  3.29  3.33  3.38  3.45 1510.0  1.0
phi[1]   0.01  1.5e-5 6.2e-4 9.4e-3 0.01  0.01  0.01  0.01 1699.0  nan
theta    2.57  5.8e-3  0.21  2.16  2.42  2.56  2.71  3.01 1386.0  1.0

```

The output results are almost identical to those found using R and JAGS. The reader might notice, however, that one `Rhat` value is reported to be `nan`. This is an arithmetic instability which appears when calculating `Rhat` for a parameter value very close to zero, as is the case for `phi[1]`, and is not connected with convergence failure.²⁶

Further Reading

Hathaway, D. H. (2015). “The solar cycle.” *Living Rev. Solar Phys.* 12. DOI: [10.1007/lrsp-2015-4](https://doi.org/10.1007/lrsp-2015-4). arXiv:1502.07020 [astro-ph.SR].

Harrison, J., A. Pole, M. West (1994). *Applied Bayesian Forecasting and Time Series Analysis*. Springer.

10.11 Gaussian Model, ODEs, and Type Ia Supernova Cosmology

Type Ia supernovae (SNe Ia) are extremely bright transient events which can be used as standardizable candles for distance measurements in cosmological scales. In the late 1990s they provided the first evidence for the current accelerated expansion of the Universe (Perlmutter *et al.*, 1999; Riess *et al.*, 1998) and consequently the existence of dark energy. Since then they have been central to every large-scale astronomical survey aiming to shed light on the dark-energy mystery.

In the last few years, a considerable effort has been applied by the astronomical community in an attempt to popularize Bayesian methods for cosmological parameter inference, especially when dealing with type Ia supernovae data (e.g. Andreon, 2011; Ma *et al.*, 2016; Mandel *et al.*, 2011; Rubin *et al.*, 2015; Shariff *et al.*, 2015). Thus, we will not refrain from tackling this problem and showing how Stan can be a powerful tool to deal with such complex model.

At maximum brightness the observed magnitude of an SN Ia can be connected to its *distance modulus* μ through the expression

$$m_{\text{obs}} = \mu + M - \alpha x_1 + \beta c, \quad (10.21)$$

where m_{obs} is the observed magnitude, M is the magnitude, and x_1 and c are the stretch and color corrections derived from the SALT2 standardization (Guy *et al.*, 2007), respectively. To take into account the effect of the host stellar mass M_\star on M and β , we use the correction proposed by Conley *et al.* (2011):

$$M = \begin{cases} M & \text{if } M_\star < 10^{10} M_\odot \\ M + \Delta M & \text{otherwise} \end{cases} \quad (10.22)$$

Considering a flat Universe, $\Omega_k = 0$, containing dark energy and dark matter, the cosmological connection can be expressed as

$$\mu = 5 \log_{10} \left(\frac{d_L}{10pc} \right), \quad (10.23)$$

$$d_L(z) = (1+z) \frac{c}{H_0} \int_0^z \frac{dz}{E(z)}, \quad (10.24)$$

$$E(z) = \sqrt{\Omega_m(1+z)^3 + (1-\Omega_m)(1+z)^{3(1+w)}}, \quad (10.25)$$

where d_L is the luminosity distance, c the speed of light, H_0 the Hubble constant, Ω_m the dark-energy density, and w the dark-energy equation of state parameter.

In what follows we will begin with a simplified version of this problem and, subsequently, guide the reader through implementations of further complexity.

10.11.1 Data

We used data provided by [Betoule et al. \(2014\)](#), known as the joint light-curve analysis (JLA) sample.²⁷ This is a compilation of data from different surveys which contains 740 high quality spectroscopically confirmed SNe Ia up to redshift $z \sim 1.0$.

10.11.2 The Statistical Model Formulation

Our statistical model will thus have one response variable (the observer magnitude, m_{obs}) and four explanatory variables (the redshift z , the stretch x_1 , the color c , and the host galaxy mass M_{host}).

The complete statistical model can be expressed as

$$\begin{aligned}
m_{\text{obs};i} &\sim \text{Normal}(\eta_i, \varepsilon) \\
\eta_i &= 25 + 5 \log_{10}(dl_i(H_0, w, \Omega_m)) + M(\Delta M) - \alpha x_{1;i} + \beta c_i \\
\varepsilon &\sim \text{Gamma}(10^{-3}, 10^{-3}) \\
M &\sim \text{Normal}(-20, 5) \\
\alpha &\sim \text{Normal}(0, 1) \\
\beta &\sim \text{Normal}(0, 10) \\
\Delta M &\sim \text{Normal}(0, 1) \\
\Omega_m &\sim \text{Uniform}(0, 1) \\
H_0 &\sim \text{Normal}(70, 5) \\
i &= 1, \dots, N
\end{aligned}$$

(10.26)

where dl is given by Equation 10.24 and M by Equation 10.22. We use conservative priors over the model parameters. These are not completely non-informative but they do allow a large range of values to be searched without putting a significant probability on non-physical values.

10.11.3 Running the Model in R using Stan

After carefully designing the model we must face the extra challenge posed by the integral in the luminosity-distance definition. Fortunately, Stan has a built-in ordinary differential equation (ODE) solver which provides a user-friendly solution to this problem (the book [Andreon and Weaver, 2015](#), Section 8.12, shows how this can be accomplished using JAGS). However, it is currently not possible to access this feature using `pystan`,²⁸ so we take this opportunity to show how Stan can also be handled from R using the package `rstan`.

Stan's ODE solver is explained in detail in the manual [Team Stan \(2016, Section 44\)](#). We advise the reader to go through that section before using it in your research. Here we would merely like to call attention for a couple of important points.

- The ODE takes as input a function which returns the derivatives at a given time, or, as in our case, a given redshift. This function must have, as input, time (or redshift), system state (the solution at $t = 0$), parameters (input a list if you have more than one parameter), real data, and integer data, in that order. In cases where your problem does not require integer or real data, you must still input an empty list (see Code 10.26 below).
- You must define a zero point for time (redshift, $z_0 = 0$) and the state of the system at that point (in our case, $\epsilon_0 = 0$). These must be given as input data in the dictionary to be passed to Stan. We strongly advise the reader to play a little with a set of simulated toy data before jumping into a real scientific scenario, so that you can develop an intuition about the behavior of the function and about the initial parameters of your model.

Our goal with this example is to provide a clean environment where the ODE solver role is highlighted. A more complex model is presented subsequently.

Code 10.26 Bayesian normal model for cosmological parameter inference from type Ia supernova data in R using Stan.

```
=====
library(rstan)

# Preparation
# Set initial conditions
z0 = 0                                # initial redshift
E0 = 0                                # integral(1/E) at z0

# physical constants
c = 3e5                                 # speed of light
H0 = 70                                 # Hubble constant

# Data
# Read data
data <- read.csv("~/data/Section_10p11/jla_lcparams.txt", header=T)

# Remove repeated redshift
data2 <- data[!duplicated(data$zcmb),]

# Prepare data for Stan
nobs <- nrow(data2)                      # number of SNe
index <- order(data2$zcmb)                # sort according to redshift
ObsMag <- data2$mb[index]                 # apparent magnitude
redshift <- data2$zcmb[index]              # redshift
color <- data2$color[index]               # color
x1 <- data2$x1[index]                    # stretch
hmass <- data2$m3rdvar[index]            # host mass

stan_data <- list(nobs = nobs,
                  E0 = array(E0, dim=1),
                  z0 = z0,
                  c = c,
                  H0 = H0,
                  obs_mag = ObsMag,
                  redshift = redshift,
                  x1 = x1,
                  color = color,
                  hmass = hmass)

# Fit
stan_model="
functions {
  /**
   * ODE for the inverse Hubble parameter.
   * System State E is 1 dimensional.
   * The system has 2 parameters theta = (om, w)
   *
   * where
   *
   *   om:      dark matter energy density
   *   w:       dark energy equation of state parameter
   *
   * The system redshift derivative is
   *
   * d.E[1] / d.z =
   * 1.0/sqrt(om * pow(1+z,3) + (1-om) * (1+z)^(3 * (1+w)))
   *
   * @param z redshift at which derivatives are evaluated.
   * @param E system state at which derivatives are evaluated.
   * @param params parameters for system.
   * @param x_r real constants for system (empty).
   * @param x_i integer constants for system (empty).
  */
  real[] Ez(real z,
            real[] H,
            real[] params,
            real[] x_r,
            int[] x_i) {
    real dEdz[1];
    ...
}
```

```

dEdz[1] = 1.0/sqrt(params[1]^(1+z)^3
+(1-params[1])*(1+z)^(3*(1+params[2])));

    return dEdz;
}
} data {
    int<lower=1> nobs;           // number of data points
    real E0[1];                  // integral(1/H) at z=0
    real z0;                     // initial redshift, 0
    real c;                      // speed of light
    real H0;                     // Hubble parameter
    vector[nobs] obs_mag;        // observed magnitude at B max
    real x1[nobs];               // stretch
    real color[nobs];            // color
    real redshift[nobs];          // redshift
    real hmass[nobs];             // host mass
}
transformed data {
    real x_r[0];                 // required by ODE (empty)
    int x_i[0];
}
parameters{
    real<lower=0, upper=1> om;      // dark matter energy density
    real alpha;                    // stretch coefficient
    real beta;                     // color coefficient
    real Mint;                     // intrinsic magnitude
    real deltaM;                   // shift due to host galaxy mass
    real<lower=0> sigint;          // magnitude dispersion
    real<lower=-2, upper=0> w;       // dark matter equation of state
                                    // parameter
}
transformed parameters{
    real DC[nobs,1];              // co-moving distance
    real pars[2];                 // ODE input = (om, w)
    vector[nobs] mag;             // apparent magnitude
    real dl[nobs];                // luminosity distance
    real DH;                      // Hubble distance = c/H0

    pars[1] = om;
    pars[2] = w;
    DH = (c/H0);

    # Integral of 1/E(z)
    DC = integrate_ode_rk45(Ez, E0, z0, redshift, pars, x_r, x_i);

    for (i in 1:nobs) {
        dl[i] = DH * (1 + redshift[i]) * DC[i, 1];
        if (hmass[i] < 10) mag[i] = 25 + 5 * log10(dl[i])
                                + Mint - alpha * x1[i] + beta
                                * color[i];
        else
            mag[i] = 25 + 5 * log10(dl[i])
                        + Mint + deltaM - alpha * x1[i] + beta * color[i];
    }
}
model {
    # Priors and likelihood
    sigint ~ gamma(0.001, 0.001);
    Mint ~ normal(-20, 5.);
    beta ~ normal(0, 10);
    alpha ~ normal(0, 1);
    deltaM ~ normal(0, 1);

    obs_mag ~ normal(mag, sigint);
}

# Run MCMC
fit <- stan(model_code = stan_model,
             data = stan_data,
             seed = 42,
             chains = 3,
             iter = 15000,
             cores = 3,
             warmup = 7500
)

```

```
# Output
# Summary on screen
print(fit,pars=c("om", "Mint","alpha","beta","deltaM", "sigint"),
      intervals=c(0.025, 0.975), digits=3)
=====
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
om	0.232	0.001	0.091	0.036	0.172	0.243	0.300	0.380	7423	1
Mint	-19.059	0.000	0.017	-19.094	-19.071	-19.059	-19.048	-19.027	8483	1
w	-0.845	0.002	0.180	-1.237	-0.960	-0.829	-0.708	-0.556	7457	1
alpha	0.119	0.000	0.006	0.106	0.114	0.119	0.123	0.131	16443	1
beta	2.432	0.001	0.071	2.292	2.384	2.432	2.480	2.572	16062	1
deltaM	-0.031	0.000	0.013	-0.055	-0.039	-0.031	-0.022	-0.006	11938	1
sigint	0.159	0.000	0.004	0.151	0.156	0.159	0.162	0.168	16456	1

The results are consistent with those reported by [Ma et al. \(2016, Section 4\)](#), who applied Bayesian graphs to the same data. A visual representation of posteriors over Ω_m and w is shown in Figure 10.23, left-hand panel.

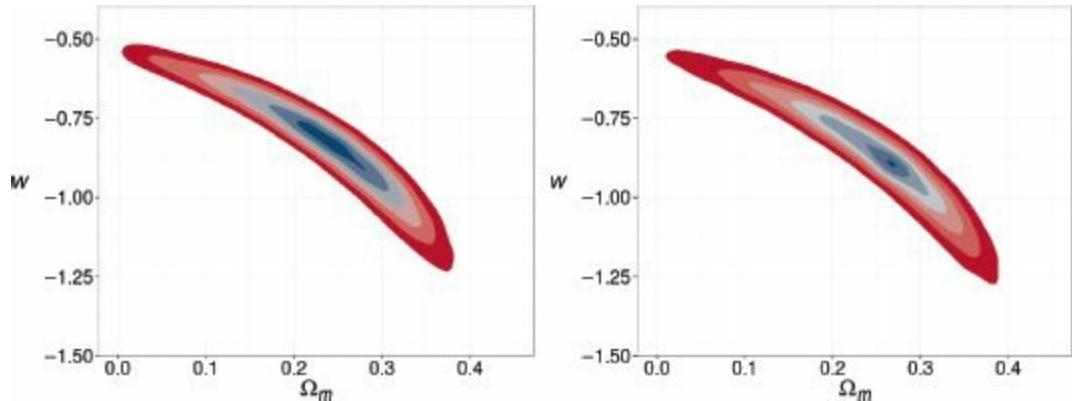


Figure 10.23 Joint posterior distributions over the dark-matter energy density Ω_m and equation of state parameter w obtained from a Bayesian Gaussian model applied to the JLA sample. Left: the results without taking into account errors in measurements. Right: the results taking into account measurement errors in color, stretch, and observed magnitude.

10.11.4 Errors in Measurements

The next natural complexity to be added to this example is the existence of errors in the measurements. Following the recipe described in Section 4.3, we present below a model that includes errors in color, stretch, and observed magnitude.

$$\begin{aligned}
 m_{\text{obs};i} &\sim \text{Normal}(m_{\text{true};i}, m_{\text{err};i}) \\
 m_{\text{true};i} &\sim \text{Normal}(\eta_i, \varepsilon) \\
 \eta_i &= 25 + 5 \log_{10}(dl_i(H_0, w, \Omega_m)) + M(\Delta M) - \alpha x_{1\text{true};i} + \beta c_{\text{true};i} \\
 \varepsilon &\sim \text{Gamma}(10^{-3}, 10^{-3}) \\
 x_{1\text{true};i} &\sim \text{Normal}(0, 5) \\
 x_{1;i} &\sim \text{Normal}(x_{1\text{true};i}, x_{\text{err};i}) \\
 c_{\text{true};i} &\sim \text{Normal}(0, 2) \\
 c_i &\sim \text{Normal}(c_{\text{true};i}, c_{\text{err};i}) \\
 M &\sim \text{Normal}(-20, 5) \\
 \alpha &\sim \text{Normal}(0, 1) \\
 \beta &\sim \text{Normal}(0, 10) \\
 \Delta M &\sim \text{Normal}(0, 1) \\
 \Omega_m &\sim \text{Uniform}(0, 1) \\
 i &= 1, \dots, N
 \end{aligned}$$

(10.27)

Small modifications are necessary to implement these effects. In Code 10.26 we must read the measured errors and include them in the list of inputs:

```
# Prepare data for Stan
ObsMagErr <- data2$dbm[index]      # error in apparent magnitude
colorerr <- data2$dcolor[index] # error on color
x1err    <- data2$dx1[index]     # error on stretch

stan_data <- list( ...
  obs_magerr = ObsMagErr,
  colorerr = colorerr,
  x1err = x1err,
  ...)
```

We then add this to the Stan model:

```
data{
```

```

vector[nobs] obs_magerr;
real colorerr[nobs];
real xierr[nobs];
}

parameters{
  real truemag[nobs];
  real truec[nobs];
  real truex1[nobs];
}

transformed parameters{
  for (i in 1:nobs) {
    dl[i] = DH * (1 + redshift[i]) * DC[i, 1];
    if (hmass[i] < 10) mag[i] = 25 + 5 * log10(dl[i])
      + Mint - alpha * truex1[i] + beta *
      truec[i];
    else mag[i] = 25 + 5 * log10(dl[i])
      + Mint + deltaM - alpha * truex1[i] + beta *
      truec[i];
  }
}
model{
  truec ~ normal(0,2);
  color ~ normal(truec, colorerr);
  truex1 ~ normal(0,5);
  x1 ~ normal(truex1, xierr);
  truemag ~ normal(mag, sigint);
  obs_mag ~ normal(truemag, obs_magerr);
}

```

Such modifications will result in

	mean	se_mean	sd	2.5%	97.5%	n_eff	Rhat
om	0.242	0.001	0.090	0.043	0.390	5661	1.000
Mint	-19.051	0.000	0.016	-19.084	-19.020	6694	1.000
w	-0.877	0.003	0.185	-1.283	-0.572	5340	1.000
alpha	0.125	0.000	0.006	0.113	0.137	6426	1.002
beta	2.569	0.001	0.067	2.440	2.704	2672	1.002
deltaM	-0.043	0.000	0.012	-0.067	-0.019	8113	1.001
sigint	0.016	0.001	0.008	0.005	0.034	39	1.069

Figure 10.23 shows joint posterior distributions over M_* and w . Comparing the two panels of this figure, we see the effect on the posterior shape of adding the errors, especially the tighter constraints for the 1σ levels (dark blue). The numerical results also corroborate with our expectations: including the errors in measurement increases Ω_m , and decreases w and also has non-negligible effects on α and β .

Given the code snippets provided above, small modifications can be added to include further effects such as the presence of other uncertainties and the systematic effects described in [Betoule et al. \(2014, Section 5.5\)](#). However, it is important to be aware of your computational capabilities, given that the memory usage can escalate very fast.²⁹ Contemporary examples of similar exercises can be found in the literature pointed below.

Further Reading

- Andreon, S. and B. Weaver (2015). *Bayesian Methods for the Physical Sciences: Learning from Examples in Astronomy and Physics*. Springer Series in Astrostatistics. Springer.
- Ma, C., P.-S. Corasaniti, and B. A. Bassett (2016). “Application of Bayesian graphs to SN Ia data analysis and compression.”

- Mandel, K. S., G. Narayan, and R. P. Kirshner (2011). “Type Ia supernova light curve inference: hierarchical models in the optical and near-infrared.” *Astrophys. J.* 731, 120. DOI: 10.1088/0004-637X/731/2/120. arXiv:1011.5910.
- Rubin, D. *et al.* (2015). “UNITY: Confronting supernova cosmology’s statistical and systematic uncertainties in a unified Bayesian framework.” *Astrophys. J.* 813, 137. DOI: 10.1088/0004-637X/813/2/137. arXiv: 1507.01602.
- Shariff, H. *et al.* (2015). “BAHAMAS: new SNIa analysis reveals inconsistencies with standard cosmology.” ArXiv e-prints. arXiv: 1510.05954.

10.12 Approximate Bayesian Computation

All the examples tackled up to this point require the definition and evaluation of a likelihood function. However, it is not unlikely that in real-world situations one is faced with problems whose likelihood is not well known or is too complex for the computational resources at hand. In astronomy, such situations may appear in the form of a selection bias in stellar studies (Janson *et al.*, 2014; Sana *et al.*, 2012), data quality in time series from AGN X-ray emissions (Shimizu and Mushotzky, 2013; Uttley *et al.*, 2002), or stellar coronae emission in the UV (Kashyap *et al.*, 2002), to cite just a few. In this last astronomical example we present an alternative algorithm which enables parameter inference without the need to explicitly define a likelihood function.³⁰

Approximate Bayesian computation (ABC) uses our ability to perform quick and realistic simulations as a tool for parameter inference (a technique known as *forward simulation inference*). The main idea consists in comparing the “distance” between a large number of simulated data sets and the observed data set and keeping a record only of the parameter values whose simulation satisfies a certain distance threshold.

Despite being proposed more than 30 years ago (Rubin, 1984) ABC techniques have only recently appeared in astronomical analysis (e.g. Akeret *et al.*, 2015; Cameron and Pettitt, 2012; Ishida *et al.*, 2015; Killedar *et al.*, 2015; Lin and Kilbinger, 2015; Robin *et al.*, 2014; Schafer and Freeman, 2012; Weyant *et al.*, 2013). Here we present an evolution of the original ABC algorithm called *Population Monte Carlo ABC* (PMC-ABC, Beaumont *et al.*, 2009). It uses an initial set of random parameter values $\{\theta\}$ drawn from the priors (called a *particle system*), which evolves through incremental approximations and ultimately converges to the true posterior distribution. We give below details about the necessary ingredients, the complete ABC algorithm (Algorithm 1) and instructions for implementation using the Python package CosmoABC.³¹

The main ingredients necessary to implement ABC are: (i) a simulator, or forward model, (ii) prior probability distributions $p(\theta)$ over the input parameters θ , and (iii) a distance function, $\rho(\mathcal{D}_O, \mathcal{D}_S)$, where \mathcal{D}_O and \mathcal{D}_S denote the observed and simulated catalogs (data sets for θ), respectively. We begin by describing a simple toy model so that the reader can gain a better intuition about how the algorithm and code can be used. Subsequently we briefly describe how the same code can be applied to the determination of cosmological parameters from measurements of galaxy cluster number counts.

Suppose we have a catalog (data set) of P observations $\mathcal{D}_O = \{x_1, \dots, x_P\}$, which are

considered to be realizations of a random variable \mathcal{X} following a Gaussian distribution, $\mathcal{X} \sim Normal(\mu, \sigma^2)$. Our goal is to determine the credible intervals over the model parameters $\boldsymbol{\theta} = \{\mu, \sigma\}$ on the basis of \mathcal{D}_O and $p(\boldsymbol{\theta})$. The ABC algorithm can be summarized in three main steps.

1. Draw a large number of parameters $\boldsymbol{\theta}$, from the prior distribution, p .
2. Generate a synthetic catalog \mathcal{D}_S for each $\boldsymbol{\theta}$ and calculate its distance from the observed catalog, $\rho = \rho(\mathcal{D}_O, \mathcal{D}_S)$.
3. Use the subset of parameters $\boldsymbol{\theta}$ with the smallest distances ρ .

10.12.1 Distance

The definition of an appropriate distance, or *summary statistic*, is paramount when designing an ABC algorithm. For illustrative purposes it can be pictured as a type of dimensionality-reduction function whose purpose is to summarize the difference between two catalogs in one number. The summary statistic needs to be null for identical catalogs, $\rho(\mathcal{D}_S, \mathcal{D}_S) = 0$, and ideally it should increase as steeply as possible as the parameter values get further from the fiducial values. A summary statistic is called *sufficient* over a given parameter when it is able to encompass all possible information about the parameter enclosed in the data set (Aeschbacher *et al.*, 2012). As might be expected, frequently it is not possible to obtain an adequate summary statistic for real world problems. In practice, however, we need only to find one that is good enough for our immediate goals. We urge the reader to test the properties of his or her proposed summary statistic before adding it to an ABC environment.

10.12.2 Population Monte Carlo ABC

The algorithm proposed by Beaumont *et al.* (2009) begins by drawing a large number M of parameters $\boldsymbol{\theta}$ from the prior, each called a *particle*. For each particle we generate a synthetic data set and calculate its distance $\rho = \rho(\mathcal{D}_O, \mathcal{D}_S)$ to the real catalog. Once all distances are computed we select only the N particles with smallest ρ . The subset of parameter values which survive a distance threshold is called a *particle system*; the initial one is $\mathcal{S}_{t=0}$. At this stage all particles $\mathcal{L}_{t=0}$ are given the same weight, $W_{t=0}^j = 1.0/N$, for $j \in [1, N]$. Then we define a new distance threshold $\varepsilon_{t=1} = 75\%$ quantile of all $\rho \in \mathcal{S}_{t=0}$.

Once the first particle system, $t = 0$, is populated with N sets of parameter values, i.e., particles, we can assign a weight W_t^j to the j th particle $\boldsymbol{\theta}_t^j$ in particle system t :

$$W_t^j = \frac{p(\boldsymbol{\theta}_t^j)}{\sum_{i=1}^N W_{t-1}^i Normal(\boldsymbol{\theta}_t^j; \boldsymbol{\theta}_{t-1}^i, C_{t-1})}, \quad (10.28)$$

where $p(\boldsymbol{\theta}_t^j)$ is the prior probability distribution for $\boldsymbol{\theta}_t^j$, W_{t-1}^i is the weight of the i th particle in

particle system $t - 1$, and $\text{Normal}(\boldsymbol{\theta}^j; \boldsymbol{\theta}_{t-1}^j, C_{t-1})$ denotes a Gaussian PDF centered on $\boldsymbol{\theta}_{t-1}^j$ with covariance matrix built from \mathcal{S}_{t-1} and calculated at $\boldsymbol{\theta}^j$. It is important to note that the distribution, in this case the Gaussian PDF, must be chosen according to the data set characteristics. For our synthetic model the Gaussian works well, but it should be replaced if the parameter space has special restrictions (e.g. discrete values).

In the construction of the subsequent particle systems $\mathcal{S}_{t>0}$ we use the weights in Equation 10.28 to perform an importance sampling, as shown in Algorithm 1. The process is repeated until convergence, which, in our case, happens when the number of particle draws needed to construct a particle system is larger than a given threshold.

Algorithm 1: PMC-ABC algorithm implemented in COSMOABC. A glossary for the terms used here is presented in Table 10.3.

Data: \mathcal{D}_O → observed catalog.

Result: ABC posterior distributions over the model parameters.

```

 $t \leftarrow 0$ 
 $K \leftarrow M$ 
for  $J = 1, \dots, M$  do
  | Draw  $\theta$  from the prior  $p(\theta)$ .
  | Use  $\theta$  to generate  $\mathcal{D}_S$ .
  | Calculate distance,  $\rho = \rho(\mathcal{D}_O, \mathcal{D}_S)$ .
  | Store parameter and distance values,  $\mathcal{S}_{t=0} \leftarrow \{\theta, \rho\}$ 
end

Sort elements in  $\mathcal{S}_{t=0}$  by  $|\rho|$ .
Keep only the  $N$  parameter values with lowest distance in  $\mathcal{S}_{t=0}$ .
 $C_{t=0} \leftarrow$  covariance matrix from  $\mathcal{S}_{t=0}$ 
for  $L = 1, \dots, N$  do
  |  $W_1^L \leftarrow 1/N$ 
end

while  $N/K > \Delta$  do
  |  $K \leftarrow 0$ .
  |  $t \leftarrow t + 1$ .
  |  $\mathcal{S}_t \leftarrow []$ 
  |  $\varepsilon_t \leftarrow$  75th-quantile of distances in  $\mathcal{S}_{t-1}$ .
  | while  $\text{len}(\mathcal{S}_t) < N$  do
    |   |  $K \leftarrow K + 1$ 
    |   | Draw  $\theta_0$  from  $\mathcal{S}_{t-1}$  with weights  $\bar{W}_{t-1}$ .
    |   | Draw  $\theta$  from  $\mathcal{N}(\theta_0, C_{t-1})$ .
    |   | Use  $\theta$  to generate  $\mathcal{D}_S$ .
    |   | Calculate distance  $\rho = \rho(\mathcal{D}_O, \mathcal{D}_S)$ 
    |   | if  $\rho \leq \varepsilon_t$  then
    |   |   |  $\mathcal{S}_t \leftarrow \{\theta, \rho, K\}$ 
    |   |   |  $K \leftarrow 0$ 
    |   | end
  | end
  | for  $J = 1, \dots, N$  do
    |   |  $\bar{W}_t^J \leftarrow$  Equation (10.28).
  | end
  |  $W_t \leftarrow$  normalized weights.
  |  $C_t \leftarrow$  weighted covariance matrix from  $\{\mathcal{S}_t, W_t\}$ .
end
```

Table 10.3 Glossary for Algorithm 1.

Parameter	Description
\mathcal{D}_O	Observed data set
\mathcal{D}_S	Simulated catalog
M	Number of draws for the first iteration
\mathcal{S}	Particle system
N	Number of particles in \mathcal{S}

t	Time-step (iteration) index
K	Number of draws index
W	Importance weights
ϵ	Distance threshold
Δ	Convergence criterion
θ	Vector describing a particle
$p(\cdot)$	Prior distribution
$\rho(\cdot, \cdot)$	Distance function
$\mathcal{N}(\bullet; \theta, C)$	Gaussian PDF at \bullet with $\mu = \theta$, $\text{cov} = C$

10.12.3 Toy Model

Consider a catalog \mathcal{D}_0 containing P measurements of the same observable, $\{x_1, \dots, x_P\}$. Our model states that each measurement is a realization of the random variable \mathcal{X} , driven by a Gaussian probability distribution with mean μ_0 and scale parameter σ_0 (these are the parameters of our model). We also know, from previous experience, that the parameters are constrained in certain intervals. In other words, our priors say that $\mu_0 \in [\mu_-, \mu_+]$ and $\sigma_0 \in [\sigma_-, \sigma_+]$.

Assuming that our model is true, and the behavior of \mathcal{X} can be entirely described by a Gaussian distribution, all the information contained in a catalog can be reduced to its mean and standard deviation. Consequently, we choose a combination of these quantities as our summary statistics:

$$\rho = \text{abs}\left(\frac{\bar{\mathcal{D}}_0 - \bar{\mathcal{D}}_s}{\bar{\mathcal{D}}_0}\right) + \text{abs}\left(\frac{\sigma_{\mathcal{D}0} - \sigma_{\mathcal{D}s}}{\sigma_{\mathcal{D}0}}\right), \quad (10.29)$$

where $\bar{\mathcal{D}}_0$ is the mean of all measurements in catalog Ω_m and $\sigma_{\mathcal{D}0}$ is its standard deviation.

10.12.4 CosmoABC

The algorithm, toy model, and distance function described above are implemented in the Python package³² CosmoABC.³³

As we highlighted before, the first step in any ABC analysis is to make sure your distance definition behaves satisfactorily in ideal situations. Thus, we advise you to start with a synthetic “observed” data set so you can assess the efficiency of your summary statistic.

From the folder `~cosmoabc/examples/` copy the files `toy_model.input` and `toy_model_functions.py` to a new and empty directory.³⁴ You might be interested in taking a look inside the `toy_model_functions.py` file to understand how the priors, distance, and simulator are inserted into

In order to start using the package in a test exercise, make sure that the keyword `path_to_obs` is set to `None`. This guarantees that you will be using a synthetic “observed” catalog. In order to visualize the behavior of your distance function, run in the command line

```
$ test_ABC_distance.py -i toy_model.input -f toy_model_functions.py
```

You will be asked to enter the name of the output file and the number of particles to be drawn. The plot shown in Figure 10.24 will be generated and the following information will be shown on the screen

```
Distance between identical catalogs = [ 0. ]
New parameter value = [-0.55545252 2.59537953]
Distance between observed and simulated data = [ 2.89005959]
```

The first line of this output indicates that the distance definition behaves as expected for identical catalogs, the second shows a random set of parameter values drawn from the prior (so you can check whether they fall within the expected boundaries) and the third line shows the calculated distance from the original to the new catalog. The code will then ask you to input the number of particles to be drawn for the visual output and Figure 10.24 will be generated. We emphasize that this is only one possible strategy to evaluate the distance function, and it will not work in more complex scenarios. There is an extensive literature which might help building intuition on the subject for more complex cases (e.g. [Aeschbacher et al., 2012](#); [Burr and Skurikhin, 2013](#)).

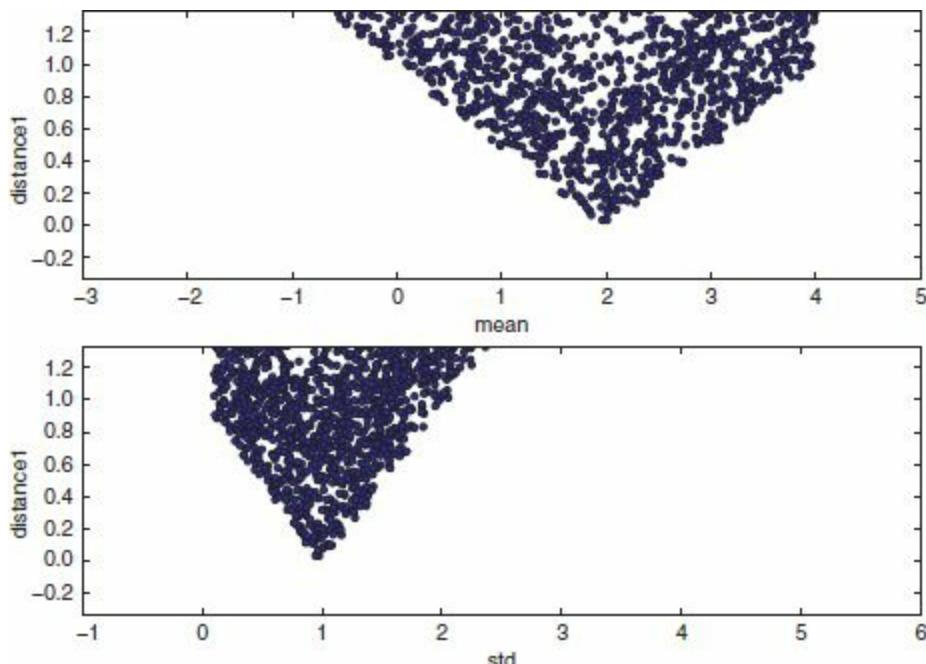


Figure 10.24 Distance function ρ from Equation 10.29 as a function of the parameters `mean` (upper panel) and `std` (lower panel).

The complete ABC algorithm can be run by typing

```
$ run_ABC.py -i toy_model.input -f toy_model_functions.py
```

The code will output one file for each particle system and a `results.pdf` file with their graphical representation. Figure 10.25 shows a few steps. At $t = 0$ (top left) we have an almost homogeneous density across the parameter space. As the system evolves ($t = 0$, top right, and $t = 12$, bottom left) the profiles converge to the fiducial parameter values (mean = 2 and std = 1, bottom right).

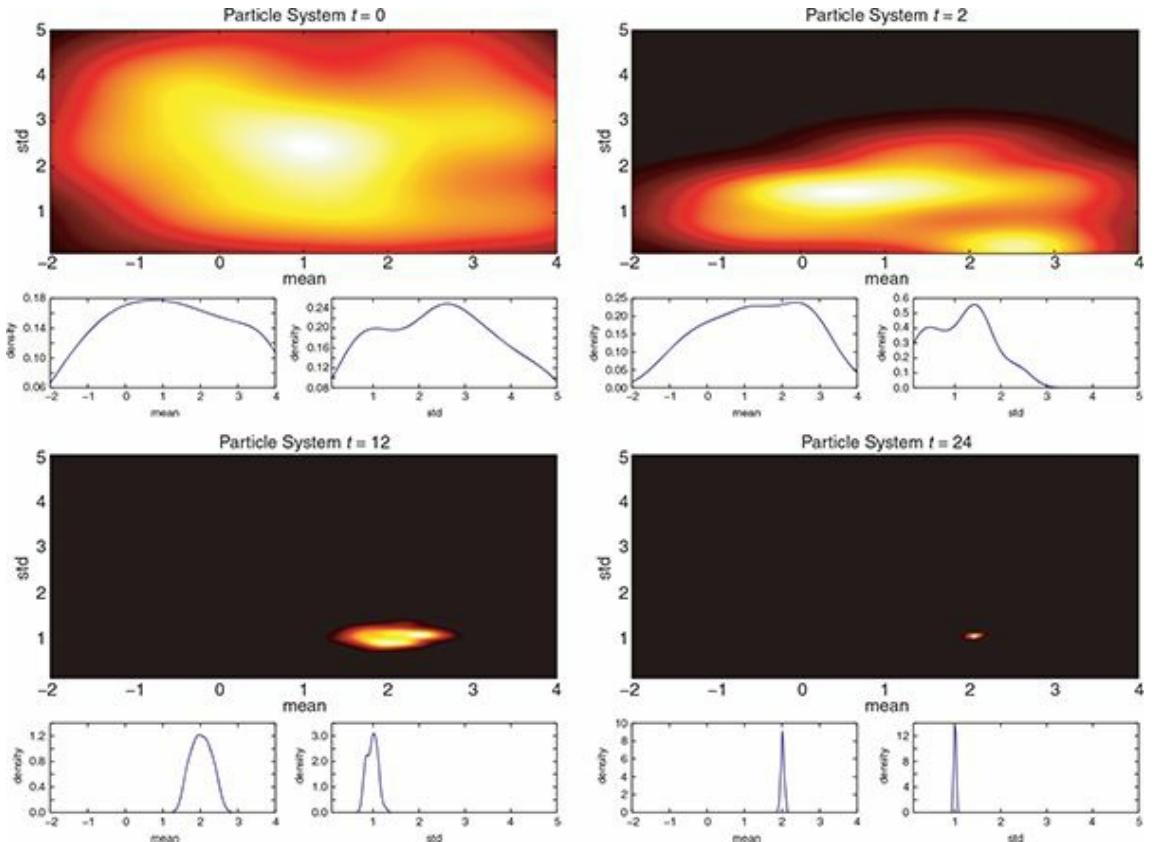


Figure 10.25 Evolution of the particle systems at different stages t of the ABC iterations for the toy model and distance function described in Section 10.12.1. At each stage t the upper panel shows the density of particles in the two dimensional parameter space of our toy model (mean, std) and the lower left- and right-hand panels show the density profiles over the same parameters.

In order to include your own simulator, distance function, and/or priors, customize the functions in the file `toy_model_functions.py` and change the corresponding keywords in the `toy_model.input` file. Remember that you can also use the latter to change the prior parameters, the size of the particle system, and the convergence thresholds.

The Sunyaev–Zeldovich effect (SZ) is the spectral distortion caused by the interaction between photons from the cosmic microwave background (CMB) and the hot plasma filling the intracluster medium (ICM). The integrated SZ flux, Y_S , is proportional to the thermal energy stored in the ICM and consequently it is possible to use measurements of CMB distortions to estimate the mass of a galaxy cluster (Sunyaev and Zeldovich, 1972). Carrying out a systematic survey of Y_{SZ} , and the respective cluster redshifts, has enabled the construction of three-dimensional maps of the matter distribution throughout the history of the Universe (Bleem *et al.*, 2015).

Furthermore, the concordance cosmological model (see the book Dodelson, 2003) and the theory of cluster formation (Kravtsov and Borgani, 2012) provide a framework where the number and spatial distribution of clusters can be used to constrain the cosmological parameters (see Ishida *et al.*, 2015, Appendix A, and references therein).

CosmoABC includes a Python warp of the C NUMCOSMO library³⁵ which allows the quick simulation of SZ survey data and enables the user to vary a large set of input parameters (see Penna-Lima *et al.*, 2014; Vitenti and Penna-Lima, 2014, for detailed descriptions). Using the file `~/cosmoabc/NumCosmo.input` (which is extensively commented), the reader can easily run the exercise performed above for the case of SZ-like data. (Ishida *et al.*, 2015) reported a detailed study of this capability using a synthetic observed catalog and demonstrated the potential of the combination of the two packages.

Finally, it is important to emphasize that ABC should not be considered as a substitute for MCMC. In cases where the likelihood is well known and tractable there is no reason to settle for an approximate solution. As a consequence, a more traditional MCMC, taking advantage of the full likelihood, should be preferred. However, in the absence of a well-behaved computationally feasible likelihood (e.g. Weyant *et al.*, 2013), ABC is a powerful alternative.

Further Reading

- Cameron, E. and A. N. Pettitt (2012). “Approximate Bayesian computation for astronomical model analysis: a case study in galaxy demographics and morphological transformation at high redshift”. *MNRAS* 425, 44–65.
 DOI: [10.1111/j.1365-2966.2012.21371.x](https://doi.org/10.1111/j.1365-2966.2012.21371.x). arXiv: 1202.1426 [astro-ph.IM].
- Ishida, E. E. O., S. D. P. Vitenti, M. Penna-Lima, and V. Busti (2015). “COSMOABC: likelihood-free inference via population monte carlo approximate Bayesian computation”. *Astron. Comput.* 13, 1–11. doi: [10.1016/j.ascom.2015.09.001](https://doi.org/10.1016/j.ascom.2015.09.001). arXiv: 1504.06129.

10.13 Remarks on Applications

The examples discussed in this chapter represent only a glimpse of the potential to be unraveled in the intersection between astronomy and statistics. We hope the resources provided in this volume encourage researchers to join the challenge of breaking down the cultural barriers which so often have prevented such interdisciplinary endeavors.

It is important to highlight that, beyond the many advantages astronomers can gain by

collaborating with statisticians, the challenge of dealing with astronomical data is also a fertile ground for statistical research. As an observational science, astronomy has to deal with data situations that are usually absent in other areas of research: the presence of systematic errors in measurements, missing data, outliers, selection bias, censoring, the existence of foreground–background effects and so forth.

Such challenges are already becoming overwhelmingly heavy, a weight astronomers alone cannot carry and statisticians rarely have the opportunity to observe. As a final remark, we would like to emphasize that the astronomical examples in this volume were made as accessible as possible, so that statisticians who have never worked with astronomy before might also contemplate its wonders and complexity.

We very much hope to see, in the coming decades, the recognition of astronomical data as a major driver for the development of new statistical research.

¹ This should not be an issue if you are using a PyStan version higher than 2.9.0.

² Developed by the COsmostatistics INitiative (COIN).

³ <https://raw.githubusercontent.com/astrobayes/BMAD/master>

⁴ Note that this example is also addressed in the book by [Andreon and Weaver \(2015\)](#), who considered a different data set.

⁵ In astronomy the solar mass is a unit of measurement, which corresponds to approximately 2×10^{30} kg.

⁶ The complete catalog can be obtained at www.physics.mcmaster.ca/~harris/GCS_table.txt.

⁷ This correction is based on empirical knowledge, since there is no consensus model determining which physical elements (environmental effects, different progenitor systems, etc; see e.g. [Hillebrandt and Niemeyer, 2000](#) and [Maoz et al., 2014](#)) are responsible for such variations.

⁸ The corresponding code is not displayed, to avoid repetition, but it is available in the online material (see the introductory statements at the start of this chapter).

⁹ An order of magnitude higher than our expectations for the parameter values ([Stan, 2016](#)).

¹⁰ This nomenclature refers to an old model of stellar evolution prior to the discovery of nuclear fusion as a source of stellar energy. Although the model is now discredited, the nomenclature is still used; see the book [Jain \(2016\)](#).

¹¹ The data table is given in the paper.

¹² Complete data available at <http://vizier.u-strasbg.fr/viz-bin/VizieR-3?-source=J/MNRAS/392/1034/table1>

¹³ <http://wwwnsatlas.org>

¹⁴ Complete data available at http://www.astro.yale.edu/jdbradford/data/hilmd/table_1_bradford_2015.fits.

¹⁵ An x -dex variation represents a change by a factor 10^x .

¹⁶ www.sdss.org/

¹⁷ www.galaxyzoo.org/

¹⁸ <http://data.galaxyzoo.org/data/redspirals/RedSpiralsA1.txt>

¹⁹ <http://data.galaxyzoo.org/data/redspirals/BlueSpiralsA2.txt>

²⁰ The quantity r_{200} is the radius inside which the mean density is 200 times the critical density of the Universe at the cluster redshift.

²¹ Data from this project was also used in Section 10.6.

²² The inverse gamma prior accounts for the fact that the variance is always positive.

²³ In standard time series analysis, an equally spaced time interval is required between two consecutive observations. This restriction can be circumvented by taking into account the existence of missing data (see the book Pole, West, and Harrison, Chapter 1).

²⁴ Complete data available at http://www.sidc.be/silso/DATA/EISN/EISN_current.csv

²⁵ <http://www.sidc.be/silso/home>

²⁶ https://groups.google.com/forum/#!topic/stan-users/hn4W_p8j3fs

²⁷ The complete data set is available at http://supernovae.in2p3.fr/sdss_snlsz_jla/ReadMe.html

²⁸ This might not be a problem if you are using a pystan version higher than 2.9.0.

²⁹ Team Stan (2016, Section 4.3) provides a few tips which might help in optimizing more complex models.

³⁰ See Ishida *et al.* (2015).

³¹ <https://pypi.python.org/pypi/CosmoABC>

³² We describe briefly here how to run the examples; a more detailed description on how to add customized distances, simulators, and priors is given in Ishida *et al.* (2015, hereafter I2015) and in the code documentation – <http://CosmoABC.readthedocs.io/en/latest/>.

³³ <https://pypi.python.org/pypi/CosmoABC>

³⁴ CosmoABC generates a lot of output files.

³⁵ www.nongnu.org/numcosmo/

11 The Future of Astrostatistics

Astrostatistics has only recently become a fully fledged scientific discipline. With the creation of the International Astrostatistics Association, the Astroinformatics & Astrostatistics Portal (ASAIP), and the IAU Commission on Astroinformatics and Astrostatistics, the discipline has mushroomed in interest and visibility in less than a decade.

With respect to the future, though, we believe that the above three organizations will collaborate on how best to provide astronomers with tutorials and other support for learning about the most up-to-date statistical methods appropriate for analyzing astrophysical data. But it is also vital to incorporate trained statisticians into astronomical studies. Even though some astrophysicists will become experts in statistical modeling, we cannot expect most astronomers to gain this expertise. Access to statisticians who are competent to engage in serious astrophysical research will be needed.

The future of astrostatistics will be greatly enhanced by the promotion of degree programs in astrostatistics at major universities throughout the world. At this writing there are no MS or PhD programs in astrostatistics at any university. Degree programs in astrostatistics can be developed with the dual efforts of departments of statistics and astronomy–astrophysics. There are several universities that are close to developing such a degree, and we fully expect that PhD programs in astrostatistics will be common in 20 years from now. They would provide all the training in astrophysics now given in graduate programs but would also add courses and training at the MS level or above in statistical analysis and in modeling in particular.

We expect that Bayesian methods will be the predominant statistical approach to the analysis of astrophysical data in the future. As computing speed and memory become greater, it is likely that new statistical methods will be developed to take advantage of the new technology. We believe that these enhancements will remain in the Bayesian tradition, but modeling will become much more efficient and reliable. We expect to see more non-parametric modeling taking place, as well as advances in spatial statistics – both two- and three-dimensional methods.

Finally, astronomy is a data-driven science, currently being flooded by an unprecedented amount of data, a trend expected to increase considerably in the next decade. Hence, it is imperative to develop new paradigms of data exploration and statistical analysis. This cross-disciplinary approach

is the key to guiding astronomy on its continuous mission to seek the next great discovery, observing unexplored regions of the cosmos and both witnessing and understanding things no human being has dreamed before.

Further Reading

- Hilbe, J. M. (2012). "Astrostatistics in the international arena." *Statistical Challenges in Modern Astronomy V*, eds. D. E. Feigelson and J. G. Babu. Springer, pp. 427-433. DOI: [10.1007/978-1-4614-3520-4_40](https://doi.org/10.1007/978-1-4614-3520-4_40).
- Hilbe, J. M. (2013). "Astrostatistics: a brief history and view to the future." *Astrostatistical Challenges for the New Astronomy*. ed. J . M. Hilbe, Springer, pp. 1–13. DOI: [10.1007/978-1-4614-3508-2_1](https://doi.org/10.1007/978-1-4614-3508-2_1).

Appendix A Bayesian Modeling using INLA

The methodology of integrated nested Laplace approximation (INLA) is based on sampling, but not on MCMC. The package can be downloaded directly from the developer's web site, www.math.ntnu.no/inla/givemeINLA.R.

The INLA package was initially developed by Rue *et al.* (2009). The authors have continued to develop and maintain the software, which supports Bayesian generalized linear models (GLMs), generalized additive models (GAMs), smoothing spline models, state space models, semiparametric regression, spatial and spatio-temporal models, log-Gaussian Cox processes, and geostatistical and geoadditive models.

The key to INLA is that the method is based on directly computing accurate approximations of the posterior marginals. Instead of using MCMC sampling, which can sometimes take hours, and even days, to estimate the posterior means, INLA can estimate posteriors in just a few seconds. The stated goal of the developers has been to provide researchers with criteria for model comparison as well as a variety of predictive measures. Being able to execute Bayesian modeling much faster than when using MCMC has also been a goal.

The INLA package has been used foremost by statisticians in implementing Bayesian models for a wide range of data situations and to provide the analyst with a simplified means to develop generalized additive models using a number of different types of smoothers. Ecologists have in particular been using INLA for spatial and spatial-temporal analysis. It is in this respect that INLA can be beneficial to future astrostatistical research.

We shall provide a simple example of a Bayesian negative binomial, using the same synthetic data as in Chapter 6. Readers may compare the results given there with the model output displayed below. The INLA negative binomial, using the `nbinomial` distribution, employs an indirect relationship between the mean and dispersion parameters. Simply invert the dispersion to obtain the direct value.

This is an R package which can be directly installed and loaded using the following directive. The `names` function provides a list of likelihood names that can be used to develop models. Once the package has been installed, use the standard `library(*)` function to load INLA:

```

> source("http://www.math.ntnu.no/inla/givemeINLA.R")
> # Or, if installed:
library(INLA)
> names(inla.models()$likelihood)

```

Now create the negative binomial synthetic data used in Chapter 6. We can request that the DIC statistic be displayed using the `control.compute` option:

```

library(MASS)
set.seed(141)
nobs <- 2500
x1 <- rbinom(nobs, size = 1, prob = 0.6)
x2 <- runif(nobs)
xb <- 1 + 2.0*x1 - 1.5*x2
a <- 3.3
theta <- 0.303      # 1/a
exb <- exp(xb)
nby <- rnbinom(n = nobs, mu = exb, theta = theta)
negbml <- data.frame(nby, x1, x2)

# Load and run INLA negative binomial

require(INLA)          # if installed INLA in earlier session
f1 <- nby ~ 1 + x1 + x2
NB <- inla(f1, family = "nbinomial", data = negbml,
           control.compute = list(dic = TRUE))

summary(NB)

Time used:
Pre-processing    Running inla    Post-processing      Total
      0.4185        1.1192        0.1232       1.6609

Fixed effects:
      mean      sd  0.025quant  0.5quant  0.975quant    mode   kld
(Intercept) 0.9865  0.0902     0.8111   0.9859    1.1650  0.9847   0
x1          2.0404  0.0808     1.8814   2.0405    2.1986  2.0408   0
x2         -1.6137  0.1374    -1.8837  -1.6137   -1.3442 -1.6136   0

The model has no random effects
Model hyperparameters:
size for the nbinomial observations (overdispersion)      mean      sd  0.025quant
                                                               0.2956  0.0099   0.2767
size for the nbinomial observations (overdispersion)  0.5quant  0.975quant   mode
                                                               0.2954   0.3152   0.295

Expected number of effective parameters(std dev): 3.011(4e-04)
Number of equivalent replicates : 830.30

Deviance information criterion (DIC): 11552.55
Effective number of parameters: 3.982

Marginal log-Likelihood: -5789.96

# invert dispersion to obtain alpha.
> 1/0.2956
[1] 3.38295

```

The parameter values are close to the values we set in the synthetic data, including the dispersion statistic of 3.3. In addition, the time taken for producing the Bayesian posteriors and related statistics is displayed as a total of 1.66 seconds. The `inla` function itself took only 1.12 seconds. This compares with some 5 minutes for executing the JAGS function on the same data. Generalized additive models and spatial analysis models may be called by extending the above code. The INLA reference manual provides directions.

Unfortunately, giving informative priors to parameters is not as simple as in JAGS or Stan. Also, some `inla` functions do not have the same capabilities as the models we give in Chapters 6 and 7.

For instance, the generalized Poisson dispersion, δ , is parameterized in such a way that negative values cannot be estimated or displayed for it. Therefore, the dispersion parameter displayed for underdispersed data is incorrect. We explain why this is the case in Section 6.3. In addition, the `inla` functions for the Bayesian zero-inflated Poisson and negative binomial models display only the binary component intercept value. The count component is displayed properly but the binary component is deficient. It is promised in future versions that the zero-inflated model limitation will be fixed, but no mention has been made of correcting the Bayesian generalized Poisson.

Finally, researchers using the `inla` function are limited to models which are pre-programmed into the software. There are quite a few such models, but statistically advanced astronomers will generally want more control over the models they develop. Nevertheless, INLA provides researchers with the framework for extending Bayesian modeling to spatial, spatial-temporal, GAM, and other types of analysis. Built-in `inla` functions such as the generalized extreme value (gev) distribution can be of particular importance to astronomers. Bayesian survival or failure analysis functions are also available. However, astrostatisticians should be aware of the limitations of the `inla` function prior to committing their research data to it.

Further Reading

Rue, H., S. Martino, and N. Chopin (2009). “Approximate Bayesian inference for latent Gaussian models by using integrated nested Laplace approximations.” *J. Roy. Statist. Soc.: Series B (Statistical Methodology)* 71(2), 319–392. DOI: [10.1111/j.1467-9868.2008.00700.x](https://doi.org/10.1111/j.1467-9868.2008.00700.x).

Appendix B Count Models with Offsets

Offsets for count models are used to adjust the counts for their being collected or generated over different periods of time or from different areas. More counts of some astrophysical event may occur in larger areas than in smaller areas, or over longer periods than over shorter periods.

An offset is added to the linear predictor of a model and is constrained to have a coefficient equal to unity. It is not a parameter to be estimated but is given in the data as an adjustment factor. In the synthetic data below, which generates a Poisson model with an offset, we can see how the offset enters the model. Remember that, since the Poisson model has a log link, the offset must be logged when added to the linear predictor.

When count models are run with offsets, it is typical to refer to them as rate models, meaning the rate at which counts are occurring in different areas or over different periods of time. For the models in this text, offsets can be employed for the Poisson, negative binomial, generalized Poisson, and NB-P models.

Code B.1 Data for Poisson with offset.

```
=====
library(MASS)
x1 <- runif(5000)
x2 <- runif(5000)
m <- rep(1:5, each=1000, times=1)*100      # creates offset as defined
logm <- log(m)                            # log the offset
xb <- 2 + .75*x1 - 1.25*x2 + logm        # linear predictor w offset
exb <- exp(xb)
py <- rpois(5000, exb)
pdata <- data.frame(py, x1, x2, m)
=====
```

The offset, `m`, is the same for each group of counts in the data – five groups of 1000 each.

```
> table(m)
m
100 200 300 400 500
1000 1000 1000 1000 1000
```

Code B.2 Bayesian Poisson with offset.

```
=====
require(R2jags)
X <- model.matrix(~ x1 + x2, data = pdata)
K <- ncol(X)
```

```

model.data <- list(Y = pdata$py,
                    N = nrow(pdata),
                    X = X,
                    K = K,
                    m = pdata$m)      #  list offset

sink("PRATE.txt")
cat("
model{
# Diffuse normal priors betas
for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001)}

# Likelihood
for (i in 1:N){
  Y[i] ~ dpois(mu[i])
  log(mu[i]) <- inprod(beta[], X[i,]) + log(m[i])      # offset added
}
}

", fill = TRUE)
sink()

# Initial parameter values
inits <- function () {
  list(
    beta = rnorm(K, 0, 0.1)
  )
}

params <- c("beta")      # posterior parameters to display

poisR <- jags(data = model.data,
               inits = inits,
               parameters = params,
               model.file = "PRATE.txt",
               n.thin = 3,
               n.chains = 3,
               n.burnin = 3000,
               n.iter = 5000)
print(poisR, intervals=c(0.025, 0.975), digits=3)
=====

          mu.vect   sd.vect   2.5%   97.5%   Rhat   n.eff
beta[1]     2.000    0.001   1.998   2.002   1.001   6000
beta[2]     0.750    0.001   0.748   0.752   1.001   6000
beta[3]    -1.249    0.001  -1.251  -1.247   1.001   6000
deviance 50694.553   26.536 50691.145 50700.292   1.001   6000

```

pD = 352.2 and DIC = 51046.8

We ran a Poisson maximum likelihood estimation (MLE) on the `pdata`, showing that the results are the same, that is, the Bayesian Poisson with offsets model having diffuse priors and the maximum likelihood model results are nearly the same:

```

poio <- glm(py ~ x1 + x2 + offset(log(m)), family=poisson, data=pdata)
summary(poio)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) 2.0001451 0.0008601 2325.6 <2e-16
x1          0.7498218 0.0011367  659.7 <2e-16
x2         -1.2491636 0.0011594 -1077.4 <2e-16
Null deviance: 1653888.7 on 4999 degrees of freedom
Residual deviance: 4941.7 on 4997 degrees of freedom
AIC: 50697

```

Frequently, count data is structured in grouped format. In the `ratep` data below, the counts `y` are adjusted for being generated or counted from areas `m` of different sizes, or being recorded over different periods of time, also `m`. This is generic code that can be used for spatial or temporal adjustment.

We next show specific data that could have come from a table of information. Consider the table below with variables x_1 , x_2 , and x_3 . We have y counts from m observations, either area sizes or time periods; y is the count variable and m the offset.

	1	x_3	0
x_2	1	2	3
	1	2	3
	1	6/45 9/39 17/29 11/54 13/47 21/44	
x_1	0	8/36 15/62 7/66 10/57 19/55 12/48	

To put the information from the table into a form suitable for modeling, we may create the following variables and values:

```

Grouped data
=====
y <- c(6,11,9,13,17,21,8,10,15,19,7,12)
m <- c(45,54,39,47,29,44,36,57,62,55,66,48)
x1 <- c(1,1,1,1,1,0,0,0,0,0,0,0)
x2 <- c(1,1,0,0,1,1,0,0,1,1,0,0)
x3 <- c(1,0,1,0,1,0,1,0,1,0,1,0)
ratep <- data.frame(y,m,x1,x2,x3)
=====
```

With a minor amendment to the above JAGS code we can model the tabular data as follows.

Code B.3 Bayesian Poisson with offset grouped data.

```

=====
require(R2jags)
X <- model.matrix(~ x1 + x2 + x3, data = ratep)
K <- ncol(X)

model.data <- list(Y = ratep$y,
                     N = nrow(ratep),
                     X = X,
                     K = K,
                     m = ratep$m)

sink("PRATE.txt")
cat("
model{
# Diffuse normal priors betas
for (i in 1:K) { beta[i] ~ dnorm(0, 0.0001)}

# Likelihood
for (i in 1:N){
  Y[i] ~ dpois(mu[i])
  log(mu[i]) <- inprod(beta[], X[i,]) + log(m[i])
}
",
",fill = TRUE)
sink()

# Initial parameter values
inits <- function () {
  list(
    beta = rnorm(K, 0, 0.1)
  )
}

params <- c("beta")

poisR <- jags(data = model.data,
               inits = inits,
               parameters = params,
               model.file = "PRATE.txt",
               n.thin = 3,
```

```

n.chains = 3,
n.burnin = 8000,
n.iter = 12000)
print(poisR, intervals=c(0.025, 0.975), digits=3)
=====
      mu.vect  sd.vect    2.5%   97.5%   Rhat  n.eff
beta[1]   -1.448   0.149  -1.751  -1.165  1.003  1000
beta[2]    -0.048   0.026  -0.103  -0.001  1.002  1900
beta[3]     0.524   0.176   0.174   0.879  1.002  1200
beta[4]    -0.331   0.178  -0.686   0.020  1.001  4000
deviance    74.265  3.391  70.609  81.717  1.002  4000
pD = 5.7 and DIC = 80.0

```

The maximum likelihood Poisson results on the same data are displayed as

```

poir <- glm(y ~ x1 + x2 + x3 + offset(log(m)), family=poisson, data=ratep)
summary(poir)

```

Coefficients:

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	-1.44589	0.14853	-9.735	< 2e-16
x1	-0.04570	0.02549	-1.793	0.07301
x2	0.52872	0.17495	3.022	0.00251
x3	-0.32524	0.17249	-1.886	0.05935

```

Null deviance: 30.563 on 11 degrees of freedom
Residual deviance: 18.569 on 8 degrees of freedom
AIC: 78.105

```

The results are close to the Bayesian model with diffuse priors. Remember that repeated sampling will produce slightly different results, but they will generally be quite close to what is displayed above.

Bayesian Negative Binomial Model with Offset

The negative binomial model can be used with an offset in the same manner as the Poisson model. As for the Poisson, declaring the offset in the JAGS list of model specifics and adding the log of the offset to the linear predictor is all that is needed. The synthetic code below generates data for a negative binomial model with offset m, where the values of the intercept and predictors x1 and x2 are respectively 2, 0.75, and -1.25 . The indirect parameterization of the dispersion parameter, theta, is 20, and the direct parameterization (α) is $1/\theta = 0.049$.

Code B.4 Bayesian negative binomial with offset.

```

=====
library(MASS)
require(VGAM)
set.seed(42)
nobs <- 500
x1 <- rbinom(nobs, size=1, 0.5)
x2 <- rbinom(nobs, size=1, 0.5)
m <- rep(1:5, each=100, times=1)*100      # creates offset as defined
logm <- log(m)
theta <- 20
xb <- 2+0.75*x1 -1.25*x2+logm
exb <- exp(xb)
nby <- rnegbin(n=nobs, mu=exb, theta = theta)
nbdata <- data.frame(nby,m,x1,x2)
summary(mynb0 <- glm.nb(nby ~ x1 + x2 + offset(log(m)), data=nbdata))
=====
Coefficients:
      Estimate  Std. Error   z value   Pr(>|z|)  
(Intercept) 2.00345   0.01673  119.72 <2e-16 ***
x1          0.74246   0.01987   37.37 <2e-16 ***

```

```

x2      -1.24313     0.01987    -62.55    <2e-16 ***
(Dispersion parameter for Negative Binomial(20.7699) family taken to be 1)

Null deviance: 5710.73  on 499  degrees of freedom
Residual deviance: 503.91  on 497  degrees of freedom
AIC: 7227.4

```

The indirect dispersion parameter is 20.77. To calculate the direct dispersion we invert this value.
The directly parameterized Bayesian negative binomial model dispersion is therefore

```

# NB direct parameterization dispersion parameter
> 1/mynb0$theta
[1] 0.04814651

```

Code B.5 Bayesian negative binomial with offset grouped data.

```

=====
require(R2jags)
X <- model.matrix(~ x1 + x2, data = nbdata)
K <- ncol(X)

model.data <- list(
  Y = nbdata$nby,
  X = X,
  K = K,
  N = nrow(nbdata),
  m = nbdata$m)

sink("NBOFF.txt")
cat("
  model{
    # Priors for betas
    for (i in 1:K) { beta[i] ~ dnorm(0, 0.01)}
    # Prior for alpha
    alpha ~ dgamma(0.01, 0.01)

    # Likelihood function
    for (i in 1:N){
      Y[i] ~ dnbin(p[i], 1/alpha)
      p[i] <- 1.0/(1.0 + alpha*mu[i])
      log(mu[i]) <- inprod(beta[], X[i,])+log(m[i])
    }
  }
  ",fill = TRUE)
sink()

inits <- function () {
  list(
    beta = rnorm(K, 0, 0.1),
    alpha = runif(0.01, 1)
  )
}
params <- c("beta", "alpha")

NBofd <- jags(data = model.data,
                inits = inits,
                parameters = params,
                model = "NBOFF.txt",
                n.thin = 3,
                n.chains = 3,
                n.burnin = 10000,
                n.iter = 15000)

print(NBofd, intervals=c(0.025, 0.975), digits=3)
=====
```

	mu.vect	sd.vect	2.5%	97.5%	Rhat	n.eff
alpha	0.049	0.003	0.043	0.055	1.001	4000
beta[1]	2.003	0.017	1.970	2.036	1.001	15000
beta[2]	0.743	0.020	0.705	0.782	1.001	15000
beta[3]	-1.243	0.020	-1.282	-1.204	1.001	15000
deviance	7223.361	2.754	7219.854	7230.171	1.001	15000

pD = 3.8 and DIC = 7227.2

The posterior means are all as expected given the synthetic data and the fact that we have used diffuse priors. Note that the dispersion parameter is estimated as 0.049. This is the value we expected as well. We may convert back to the indirect dispersion value by changing the term θ to α and amending the lines

```
Y[i] ~ dnegbin(p[i], 1/alpha)
p[i] <- 1/(1 + alpha*mu[i])
```

to

```
Y[i] ~ dnegbin(p[i], theta)
p[i] <- theta / (theta + mu[i])
```

This produces the code needed for the indirect parameterization. The code in full is provided below:

Code B.6 Indirect parameterization.

```
=====
require(R2jags)
X <- model.matrix(~ x1 + x2 )
K <- ncol(X)

model.data <- list(
  Y = nbdata$nby,
  X = X,
  K = K,
  N = nrow(nbdata),
  m = nbdata$m)

sink("NBOFF.txt")
cat("
  model{
    # Priors for betas
    for (i in 1:K) { beta[i] ~ dnorm(0, 0.01)}
    # Prior for theta
    theta ~ dgamma(0.01, 0.01)

    # Likelihood function
    for (i in 1:N){
      Y[i] ~ dnegbin(p[i], theta)
      p[i] <- theta / (theta + mu[i])
      log(mu[i]) <- inprod(beta[], X[i,])+log(m[i])
    }
  }
  ",fill = TRUE)
sink()

inits <- function () {
  list(
    beta = rnorm(K, 0, 0.1),
    theta = runif(0.01, 1)
  )
}
params <- c("beta", "theta")

NBofi <- jags(data = model.data,
               inits = inits,
               parameters = params,
               model = "NBOFF.txt",
               n.thin = 3,
               n.chains = 3,
               n.burnin = 15000,
               n.iter = 25000)

print(NBofi, intervals=c(0.025, 0.975), digits=3)
=====
```

	mu.vect	sd.vect	2.5%	97.5%	Rhat	n.eff
beta[1]	2.004	0.017	1.972	2.036	1.001	15000
beta[2]	0.742	0.020	0.703	0.780	1.001	5900

```

beta[3]      -1.243    0.020    -1.282    -1.205    1.001    6900
theta        20.647   1.332    18.104    23.371    1.001   15000
deviance    7223.359  2.778   7219.842   7230.322    1.001   15000

```

pD = 3.9 and DIC = 7227.2

The models fit the synthetic data properly. The same logic as employed for the JAGS Bayesian Poisson and negative binomial models can be used for the generalized Poisson, NB-P, and similar count models.

Finally, the above negative binomial models work with the grouped data, `ratep`. Amending the indirect negative binomial code for use with the `ratep` data, the results are given below. They are close to the values produced using R's `glm.nb` function.

```

> print(NBofi, intervals=c(0.025, 0.975), digits=3)
Inference for Bugs model at "NBOFF.txt", fit using jags,
 3 chains, each with 25000 iterations (first 15000 discarded), n.thin = 3
n.sims = 10002 iterations saved
      mu.vect    sd.vect    2.5%    97.5%     Rhat    n.eff
beta[1]   -1.567    0.275   -2.099   -1.001    1.001    9600
beta[2]    0.218    0.302   -0.376    0.827    1.001    6900
beta[3]    0.394    0.301   -0.189    1.004    1.001   10000
beta[4]   -0.190    0.288   -0.762    0.391    1.002    1900
theta     14.570   22.236    1.899   69.229    1.006     680
deviance  76.084    3.721   71.261   85.449    1.001   10000

```

pD = 6.9 and DIC = 83.0

Appendix C Predicted Values, Residuals, and Diagnostics

We did not address the matter of predicted or fitted values at length in the text. We assume that the reader is familiar with these basic statistical procedures. There are several hints that may be given for calculating these statistics using JAGS. Fitted values and diagnostics may be calculated within the main JAGS model code, or they may be calculated following posterior parameter estimation. We prefer to follow the recommendation of Zuur *et al.* (2013) and calculate model diagnostics within the model code. However, the actual diagnostics are displayed after the model estimation. The code below comes from Code 6.4 in the text. The data are assumed to be generated from Code 6.2. The `pois` data from Code 6.2 and predictors `x1` and `x2` are supplied to the `x` matrix at the beginning of the JAGS code below. New diagnostic code is provided in the modules labeled “Model mean, variance, Pearson residuals”, “Simulated Poisson statistics”, and “Poisson log-likelihood”. The lines with objects `L`, `AIC`, `PS`, and `PSSIM` are also added as diagnostic code. In addition, since we wish to use the calculated statistics – which are in fact the means of statistics obtained from simulated posterior values – to evaluate the model, they must be saved in the `params` object. Therefore, instead of only saving “`beta`”, which gives the posterior means of the predictors (these are analogous to frequentist coefficients or slopes), we add fitted, residual, and other statistics. The names and explanations of the new diagnostic values are provided in the code.

Code C.1 JAGS Bayesian Poisson model with diagnostic code.

```
=====
require(R2jags)
x <- model.matrix(~ x1 + x2, data = pois)
K <- ncol(x)
model.data <- list(Y = pois$py,
                     X = X,
                     K = K,
                     N = nrow(pois))
sink("Poi.txt")
cat("
  model{
    for (i in 1:K) {beta[i] ~ dnorm(0, 0.0001)}
    for (i in 1:N) {
      Y[i] ~ dpois(mu[i])
      log(mu[i]) <- inprod(beta[], X[i,])
    }
  }

  # Model mean, variance, Pearson residuals
  for (i in 1:N) {
    ExpY[i] <- mu[i]                                # Poisson mean
    VarY[i]  <- mu[i]                                # Poisson variance
    E[i]     <- (Y[i] - ExpY[i]) / sqrt(VarY[i])   # Pearson residual
  }

  # Simulated Poisson statistics
  for (i in 1:N) {
```

```

YNew[i] ~ dpois(mu[i])                      # simulated Poisson
means
ENew[i] <- (YNew[i] - ExpY[i]) / sqrt(VarY[i]) # simulated Pearson
residual
D[i] <- E[i]^2                                # squared Pearson
residuals
DNew[i] <- ENew[i]^2                          # squared simulated P
residuals
}

# Poisson log-likelihood
for (i in 1:N) {
  ll[i] <- Y[i] * log(mu[i]) - mu[i] - loggam(Y[i] +1)
}
L <- sum(ll[1:N])                            # log-likelihood
AIC <- -2 * sum(ll[1:N]) + 2 * K            # AIC statistic
PS <- sum(D[1:N])                           # Model Pearson statistic
PSSim <- sum(DNew[1:N])                     # Simulated Pearson statistic

}", fill = TRUE)
sink()

inits <- function () {
  list(beta = rnorm(K, 0, 0.1))}

params <- c("beta", "ExpY", "E", "PS", "PSSim", "YNew", "L", "AIC")

POI <- jags(data = model.data,
             inits = inits,
             parameters = params,
             model = "Poi.txt",
             n.thin = 1,
             n.chains = 3,
             n.burnin = 3000,
             n.iter = 5000)
# Print(POI, intervals=c(0.025, 0.975), digits=3)

```

We next load the source file we provide on the book's web site. It is based in part on code from Zuur *et al.* (2013). The file contains several functions that allow us to develop nicer-looking output as well as trace and distribution plots for each parameter specified:

```
> source(~ /CH-figures.R)
```

If we ran the `print()` code at the end of the above model code, the residuals and fitted value for each observation in the model would be displayed. In order to avoid this, and to obtain only the basic information, we can use the code below.

```

> POIoutput <- POI$BUGSoutput          # posterior results
> OUT1 <- MyBUGSOutput(POIoutput, c(uNames("beta", K), "L", "AIC"))
> print(OUT1, digits = 4)
      mean      se    2.5%   97.5%
beta[1]  1.006 1.614e-02  0.9902   1.022
beta[2] -1.497 4.552e-03 -1.5034  -1.491
beta[3] -3.498 6.091e-03 -3.5034  -3.492
L     -1657.978 6.491e+01 -1659.6016 -1654.769
AIC    3321.957 1.298e+02  3315.5388 3325.203

```

The following code provides a test of the specification of the model as Poisson. The test can be used for any model. If the model fits well we expect that the summary parameters of the model will be similar to the summary data from the simulated Poisson model. Ideally, the mean number of times that the summary values of the model are greater or less than the summary values of the simulated data should be about equal. Therefore, we look for a value of about 0.5, which is at times referred to as a Bayesian model *p*-value. Values close to 0 or 1 indicated a poorly specified model; i.e., the model is not truly Poisson.

```
> out <- POI$BUGSoutput  
> mean(out$sims.list$PS > out$sims.list$PSSim)  
[1] 0.9561667
```

The following code provides a Bayesian dispersion statistic for the Poisson model. Values over 1.0 indicate likely overdispersion. Values under 1.0 indicate likely underdispersion.

```
> E <- out$mean$E      # average iterations per observation  
> N <- nrow(pois)    # observations in model  
> p <- K              # model betas  
> sum(E^2)/(N-p)     # Bayesian dispersion statistic  
[1] 1.444726
```

The data appears to be mis-specified as a Poisson model. Our Bayesian p -value is far too high and the dispersion statistic is considerably greater than 1.0. The two statistics are consistent. We can obtain values for the log-likelihood and AIC statistic, assigning them to another value using the code below. Note that the values are identical to the displayed table output above.

```
> mean(out$sims.list$L)  
[1] -1657.978  
  
> mean(out$sims.list$AIC)  
[1] 3321.957
```

Finally, trace plots and distribution curves (histograms) of each posterior parameter can be obtained using the code below. The graphics are not displayed here.

```
> vars <- c("beta[1]", "beta[2]", "beta[3]")  
> MyBUGSChains(POI$BUGSoutput, vars)  
> MyBUGSHist(POI$BUGSoutput, vars)
```

Other statistics and plots can be developed from the diagnostic code calculated in the Poisson code above. Plots of Pearson residuals versus the fitted values are particularly useful when checking for a fit. Several more advanced figures have been displayed in the text for specific models. The code for these figures can be obtained from the book's web site.

References

Books

- Andreon, S. and B. Weaver (2015). *Bayesian Methods for the Physical Sciences: Learning from Examples in Astronomy and Physics*. Springer Series in Astrostatistics. Springer.
- Chattopadhyay, A. K. and T. Chatropadhyay (2014). *Statistical Methods for Astronomical Data Analysis*. Springer Series in Astrostatistics. Springer.
- Cowles, M. K. (2013). *Applied Bayesian Statistics: With R and OpenBUGS Examples*. Springer Texts in Statistics. Springer.
- Dodelson, S. (2003). *Modern Cosmology*. Academic Press.
- Feigelson, E. D. and G. J. Babu (2012a). *Modern Statistical Methods for Astronomy: With R Applications*. Cambridge University Press.
- Feigelson, E. D. and G. J. Babu (2012b). *Statistical Challenges in Modern Astronomy V*. Lecture Notes in Statistics. Springer.
- Finch, W. H., J. E. Bolin, and K. Kelley (2014). *Multilevel Modeling Using R*. Chapman & Hall/CRC Statistics in the Social and Behavioral Sciences. Taylor & Francis.
- Gamerman, D. and H. F. Lopes (2006). *Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference, Second Edition*. Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis.
- Gelman, A., J. Carlin, H. Stern, D. Dunson, A. Vehtari, and D. Rubin (2013). *Bayesian Data Analysis, Third Edition*. Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis.
- Hardin, J. W. and J. M. Hilbe (2012). *Generalized Linear Models and Extensions, Third Edition*. Taylor & Francis.
- Hilbe, J. M. (2011). *Negative Binomial Regression, Second Edition*. Cambridge University Press.
- Hilbe, J. M. (2014). *Modeling Count Data*. Cambridge University Press.
- Hilbe, J. M. (2015). *Practical Guide to Logistic Regression*. Taylor & Francis.
- Hilbe, J. M. and A. P. Robinson (2013). *Methods of Statistical Model Estimation*. EBL-Schweitzer. CRC Press.
- Ivezic, Z., A. J. Connolly, J. T. Vanderplas, and A. Gray (2014). *Statistics, Data Mining, and Machine Learning in Astronomy: A Practical Python Guide for the Analysis of Survey Data*. EBSCO ebook academic collection. Princeton University Press.
- Jain, P. (2016). *An Introduction to Astronomy and Astrophysics*. CRC Press.
- Korner-Nievergelt, F. et al. (2015). *Bayesian Data Analysis in Ecology Using Linear Models with R, BUGS, and Stan*. Elsevier Science.
- Kruschke, J. (2010). *Doing Bayesian Data Analysis: A Tutorial Introduction with R*. Elsevier Science.
- Lunn, D., C. Jackson, N. Best, A. Thomas, and D. Spiegelhalter (2012). *The BUGS Book: A Practical Introduction to Bayesian Analysis*. Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis.
- McElreath, R. (2016). *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*. Chapman & Hall/CRC Texts in Statistical Science. CRC Press.
- Muenchen, R. A. and J. M. Hilbe (2010). *R for Stata Users*. Statistics and Computing. Springer.
- Pole, A., M. West, and J. Harrison (1994). *Applied Bayesian Forecasting and Time Series Analysis*. Springer.
- R Development Core Team (2008). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna. www.R-project.org.
- Smithson, M. and E. C. Merkle (2013). *Generalized Linear Models for Categorical and Continuous Limited Dependent Variables*. Chapman & Hall/CRC Statistics in the Social and Behavioral Sciences. Taylor & Francis.
- Suess, E. A. and B. E. Trumbo (2010). *Introduction to Probability Simulation and Gibbs Sampling with R*. Use R! Series. Springer.
- Team, Stan (2016). *Stan Modeling Language Users Guide and Reference Manual, Version 2.14.0*. <http://mc-stan.org/>.
- Teator, P. (2011). *R Cookbook*. O'Reilly Media.
- Weisberg, H. (2014). *Willful Ignorance: The Mismeasure of Uncertainty*. Wiley.
- Zuur, A. F., J. M. Hilbe, and E. N. Ieno (2013). *A Beginner's Guide to GLM and GLMM with R: A Frequentist and Bayesian Perspective for Ecologists*. Highland Statistics.

Articles

- Abazajian, K. N. *et al.* (2009). "The seventh data release of the Sloan Digital Sky Survey." *Astrophys. J. Suppl.* 182, 543–558. DOI: 10.1088/0067-0049/182/2/543. arXiv:0812.0649.
- Aeschbacher, S. *et al.* (2012). "A novel approach for choosing summary statistics in approximate Bayesian computation." *Genetics* 192(3), 1027–1047. DOI: 10.1534/genetics.112.143164.
- Akaike, H. (1974). "A new look at the statistical model identification." *IEEE Trans. Automatic Control* 19(6), 716–723.
- Akeret, J. *et al.* (2015). "Approximate Bayesian computation for forward modeling in cosmology." *J. Cosmology Astroparticle Phys.* 8, 043. DOI: 10.1088/1475-7516/2015/08/043. arXiv:1504.07245.
- Alam, S. *et al.* (2015). "The eleventh and twelfth data releases of the Sloan Digital Sky Survey: final data from SDSS-III." *Astrophys. J. Suppl.* 219, 12. DOI: 10.1088/0067-0049/219/1/12. arXiv: 1501.00963 [astro-ph.IM].
- Almeida, L. A. *et al.* (2015). "Discovery of the massive overcontact binary VFTS352: evidence for enhanced internal mixing." *Astrophys. J.* 812, 102. DOI: 10.1088/0004-637X/812/2/102. arXiv:1509.08940[astro-ph.SR].
- Andreon, S. (2011). "Understanding better (some) astronomical data using Bayesian methods." *ArXiv e-prints*. arXiv: 1112.3652 [astro-ph.IM].
- Andreon, S. and M. A. Hurn (2010). "The scaling relation between richness and mass of galaxy clusters: a Bayesian approach." *Mon. Not. Roy. Astronom. Soc.* 404, 1922–1937. DOI: 10.1111/j.1365-2966.2010.16406.x. arXiv: 1001.4639 [astro-ph.CO].
- Baldwin, J. A. *et al.* (1981). "Classification parameters for the emission-line spectra of extragalactic objects." *Publ. Astronom. Soc. Pacific* 93, 5–19. DOI: 10.1086/130766.
- Bamford, S. P. *et al.* (2009). "Galaxy zoo: the dependence of morphology and colour on environment." *Mon. Not. Roy. Astronom. Soc.* 393, 1324–1352. DOI: 10.1111/j.1365-2966.2008.14252.x. arXiv: 0805.2612.
- Bastian, N. *et al.* (2010). "A universal stellar initial mass function? a critical look at variations." *Ann. Rev. Astron. Astrophys.* 48, 339–389. DOI: 10.1146/annurev-astro-082708-101642. arXiv: 1001.2965.
- Beaumont, M. A. *et al.* (2009). "Adaptive approximate Bayesian computation." *Biometrika*, asp052.
- Benson, A. J. (2010). "Galaxy formation theory." *Phys. Rep.* 495, 33–86. DOI: 10.1016/j.physrep.2010.06.001. arXiv: 1006.5394 [astro-ph.CO].
- Betancourt, M. (2015). "Continuing sampling." <https://groups.google.com/forum/#msg/stan-users/t1IZW78M3za/ZHUNqR815MKJ> (visited on 07/04/2016).
- Betancourt, M. (2016). "Some Bayesian modeling techniques in stan." www.youtube.com/watch?v=uSjsJg8fcwY (visited on 06/18/2016).
- Betoule, M. *et al.* (2014). "Improved cosmological constraints from a joint analysis of the SDSS-II and SNLS supernova samples." *Astron. Astrophys.* 568, A22. DOI: 10.1051/0004-6361/201423413. arXiv: 1401.4064.
- Bett, P. *et al.* (2007). "The spin and shape of dark matter haloes in the Millennium simulation of a cold dark matter universe." *Mon. Not. Roy. Astronom. Soc.* 376, 215–232. DOI: 10.1111/j.1365-2966.2007.11432.x. eprint: arXiv: astro-ph/0608607.
- Biagi, V. and U. Maio (2013). "Statistical properties of mass, star formation, chemical content and rotational patterns in early $\gtrsim 9$ structures." *Mon. Not. Roy. Astronom. Soc.* 436, 1621–1638. DOI: 10.1093/mnras/stt1678. arXiv: 1309.2283[astro-ph.CO].
- Blanton, M. R. *et al.* (2011). "Improved background subtraction for the Sloan Digital Sky Survey images." *Astronom. J.* 142, 31 DOI: 10.1088/0004-6256/142/1/31. arXiv: 1105.1960 [astro-ph.IM].
- Bleem, L. E. *et al.* (2015). "Galaxy clusters discovered via the Sunyaev–Zel'dovich effect in the 2500-square-degree SPT-SZ survey." *Astrophys. J. Suppl.* 216, 27. DOI: 10.1088/0067-0049/216/2/27. arXiv: 1409.0850.
- Bonnarel, F. *et al.* (2000). "The ALADIN interactive sky atlas. A reference tool for identification of astronomical sources." *A&AS* 143, 33–40. DOI: 10.1051/aas:2000331.
- Bradford, J. D. *et al.* (2015). "A study in blue: the baryon content of isolated low-mass galaxies." *Astrophys. J.* 809, 146. DOI: 10.1088/0004-637X/809/2/146. arXiv: 1505.04819.
- Burkert, A. and S. Tremaine (2010). "A correlation between central supermassive black holes and the globular cluster systems of early-type galaxies." *Astrophys. J.* 720, 516–521. DOI: 10.1088/0004-637X/720/1/516. arXiv: 1004.0137 [astro-ph.CO].
- Burr, T. and A. Skurikhin (2013). "Selecting summary statistics in approximate bayesian computation for calibrating stochastic models." *BioMed Res. Int.* 2013.
- Cameron, E. (2011). "On the estimation of confidence intervals for binomial population proportions in astronomy: the simplicity and superiority of the Bayesian approach." *Publ. Astronom. Soc. Australia* 28, 128–139. DOI: 10.1071/AS10046. arXiv: 1012.0566 [astro-ph.IM].
- Cameron, E. and A. N. Pettitt (2012). "Approximate Bayesian computation for astronomical model analysis: a case study in galaxy demographics and morphological transformation at high redshift." *Mon. Not. Roy. Astronom. Soc.* 425, 44–65. DOI: 10.1111/j.1365-2966.2012.21371.x. arXiv: 1202.1426 [astro-ph.IM].
- Chabrier, G. (2003). "Galactic stellar and substellar initial mass function." *Publ. Astronom. Soc. Pacific* 115, 763–795. DOI: 10.1086/376392. eprint:arXiv:astro-ph/0304382.
- Chattopadhyay, G. and S. Chattopadhyay (2012). "Monthly sunspot number time series analysis and its modeling through autoregressive artificial neural network." *Europ. Physical J. Plus* 127, 43. DOI: 10.1140/epjp/i2012-12043-9. arXiv: 1204.3991 [physics.gen-ph].
- Conley, A. *et al.* (2011). "Supernova constraints and systematic uncertainties from the first three years of the Supernova

- Legacy Survey.” *Astrophys. J. Suppl.* 192, 1. DOI: 10.1088/0067-0049/192/1/1. arXiv: 1104.1443[astro-ph.CO].
- Consul, P. C. and F. Famoye (1992). “Generalized poisson regression model.” *Commun. Statistics – Theory Meth.* 21(1), 89–109. DOI: 10.1080/03610929208830766.
- Cortese, L. and T. M. Hughes (2009). “Evolutionary paths to and from the red sequence: star formation and HI properties of transition galaxies at $z \sim 0$.” *Mon. Not. Roy. Astronom. Soc.* 400, 1225–1240. DOI: 10.1111/j.1365-2966.2009.15548.x. arXiv: 0908.3564.
- de Souza, R. S. and B. Ciardi (2015). “AMADA—Analysis of multidimensional astronomical datasets.” *Astron. Comput.* 12, 100–108. DOI: 10.1016/j.ascom.2015.06.006. arXiv: 1503.07736 [astro-ph.IM].
- de Souza, R. S. et al. (2013). “Dark matter halo environment for primordial star formation.” *Mon. Not. Roy. Astronom. Soc.* 428, 2109–2117. DOI: 10.1093/mnras/sts181. arXiv: 1209.0825 [astro-ph.CO].
- de Souza, R. S. et al. (2014). “Robust PCA and MIC statistics of baryons in early minihaloes.” *Mon. Not. Roy. Astronom. Soc.* 440, 240–248. DOI: 10.1093/mnras/stu274. arXiv: 1308.6009[astro-ph.co].
- de Souza, R. S. et al. (2015a). “The overlooked potential of generalized linear models in astronomy – I: Binomial regression.” *Astron. Comput.* 12, 21–32. ISSN: 2213-1337. DOI: <http://dx.doi.org/10.1016/j.ascom.2015.04.002>. URL: www.sciencedirect.com/science/article/pii/S2213133715000360.
- de Souza, R. S. et al. (2015b). “The overlooked potential of generalized linear models in astronomy – III. Bayesian negative binomial regression and globular cluster populations.” *Mon. Not. Roy. Astronom. Soc.* 453, 1928–1940. DOI: 10.1093/mnras/stv1825. arXiv: 1506.04792 [astro-ph.IM].
- de Souza, R. S. et al. (2016). “Is the cluster environment quenching the Seyfert activity in elliptical and spiral galaxies?” *Mon. Not. Roy. Astronom. Soc.* 461, 2115–2125. DOI: 10.1093/mnras/stw1459. arXiv: 1603.06256.
- Djorgovski, S. and M. Davis (1987). “Fundamental properties of elliptical galaxies.” *Astrophys. J.* 313, 59–68. DOI: 10.1086/164948.
- Eisenstein, D. J. et al. (2011). “SDSS-III: massive spectroscopic surveys of the distant universe, the Milky Way, and extra-solar planetary systems.” *Astronom. J.* 142, 72. DOI: 10.1088/0004-6256/142/3/72. arXiv: 1101.1529 [astro-ph.IM].
- Elliott, J. et al. (2015). “The overlooked potential of generalized linear models in astronomy – II: Gamma regression and photometric redshifts”. *Astron. Comput.* 10, 61–72. DOI: 10.1016/j.ascom.2015.01.002. arXiv: 1409.7699 [astro-ph.IM].
- Fabian, A. C. (2012). “Observational evidence of active galactic nuclei feedback.” *Ann. Rev. Astron. Astrophys.* 50, 455–489. DOI: 10.1146/annurev-astro-081811-125521. arXiv: 1204.4114.
- Famoye, F. and K. P. Singh (2006). “Zero-inflated generalized poisson regression model with an application to domestic violence data.” *J. Data Sci.* 4(1), 117–130. ISSN: 1683-8602.
- Feehrer, C. E. (2000). “Dances with Wolfs: a short history of sunspot indices.” www.aavso.org/dances-wolfs-short-history-sunspot-indices (visited on 06/18/2016).
- Feigelson, E. D. and G. J. Babu (1992). “Linear regression in astronomy. II.” *Astrophys. J.* 397, 55–67. DOI: 10.1086/171766.
- Feroz, F. and M. P. Hobson (2008). “Multimodal nested sampling: an efficient and robust alternative to Markov Chain Monte Carlo methods for astronomical data analyses.” *Mon. Not. Roy. Astronom. Soc.* 384, 449–463. DOI: 10.1111/j.1365-2966.2007.12353.x. arXiv: 0704.3704.
- Ferrarese, L. and D. Merritt (2000). “A fundamental relation between supermassive black holes and their host galaxies.” *Astrophys. J.* 539, L9–L12. DOI: 10.1086/312838. eprint: astro-ph/0006053.
- Fontanot, F. (2014). “Variations of the initial mass function in semi-analytical models.” *Mon. Not. Roy. Astronom. Soc.* 442, 3138–3146. DOI: 10.1093/mnras/stu1078. arXiv: 1405.7699.
- Foreman-Mackey, D. et al. (2013). “emcee: the MCMC hammer.” *Publ. Astronom. Soc. Pacific* 125, 306–312. DOI: 10.1086/670067. arXiv: 1202.3665 [astro-ph.IM].
- Gebhardt, K. et al. (2000). “Black hole mass estimates from reverberation mapping and from spatially resolved kinematics.” *Astrophys. J.* 543, L5–L8. DOI: 10.1086/318174. eprint: astro-ph/0007123.
- Gelfand, A. E. and A. F. M. Smith (1990). “Sampling-based approaches to calculating marginal densities.” *J. Amer. Statist. Assoc.* 85(410), 398–409.
- Gelfand, A. E. et al. (1990). “Illustration of Bayesian inference in normal data models using Gibbs sampling.” *J. Amer. Statist. Assoc.* 85(412), 972–985. DOI: 10.1080/01621459.1990.10474968.
- Gelman, A. (2006). “Prior distributions for variance parameters in hierarchical models.” *Bayesian Anal.* 1(3), 515–533.
- Gelman, A. et al. (2014). “Understanding predictive information criteria for Bayesian models.” *Statist. Comput.* 24(6), 997–1016. DOI: 10.1007/s11222-013-9416-2.
- Gelman, A. et al. (2015). “Stan: a probabilistic programming language for bayesian inference and optimization.” *J. Educational and Behavioral Statist.* DOI: 10.3102/1076998615606113. eprint: <http://jeb.sagepub.com/content/early/2015/10/09/1076998615606113.full.pdf+html>.
- Geman, S. and D. Geman (1984). “Stochastic relaxation, Gibbs distribution, and the Bayesian restoration of images.” *IEEE Trans. Pattern Recognition* 6, 721–741.
- Graczyk, D. et al. (2011). “The optical gravitational lensing experiment. The OGLE-III catalog of variable stars. XII. Eclipsing binary stars in the large magellanic cloud.” *Acta Astron.* 61, 103–122. arXiv: 1108.0446 [astro-ph.SR].
- Guy, J. et al. (2007). “SALT2: using distant supernovae to improve the use of type Ia supernovae as distance indicators.”

- Astron. Astrophys.* 466, 11–21. DOI: 10.1051/0004-6361:20066930. eprint: astro-ph/0701828.
- Hadin, J. W. (2012). “Modeling underdispersed count data with generalized Poisson regression.” *Stata J.* 12(4), 736–747. www.stata-journal.com/article.html?article=st0279.
- Hahn, O. et al. (2007). “Properties of dark matter haloes in clusters, filaments, sheets and voids.” *Mon. Not. Roy. Astronom. Soc.* 375, 489–499. DOI: 10.1111/j.1365-2966.2006.11318.x. eprint: arXiv: astro-ph/0610280.
- Harris, G. L. H. and W. E. Harris (2011). “The globular cluster/central black hole connection in galaxies.” *Mon. Not. Roy. Astronom. Soc.* 410, 2347–2352. DOI: 10.1111/j.1365-2966.2010.17606.x. arXiv: 1008.4748 [astro-ph.CO].
- Harris, W. E. et al. (2013). “A catalog of globular cluster systems: what determines the size of a galaxy’s globular cluster population?” *Astrophys. J.* 772, 82. DOI: 10.1088/0004-637X/772/2/82. arXiv: 1306.2247[astro-ph.GA].
- Harris, G. L. H. et al. (2014). “Globular clusters and supermassive black holes in galaxies: further analysis and a larger sample.” *Mon. Not. Roy. Astronom. Soc.* 438, 2117–2130. DOI: 10.1093/mnras/stt2337. arXiv: 1312.5187[astro-ph.GA].
- Hastings, W. K. (1970). “Monte Carlo sampling methods using Markov chains and their applications.” *Biometrika* 57, 97–109. DOI: 10.1093/biomet/57.1.97.
- Hathaway, D. H. (2015). “The solar cycle.” *Living Rev. Solar Phys.* 12. DOI: 10.1007/lrsp-2015-4. arXiv: 1502.07020 [astro-ph.SR].
- Hilbe, J. M. and W. H. Greene (2007). Count response regression models, in *Epidemiology and Medical Statistics*, eds. C. R. Rao, J. P. Miller, and D. C. Rao, Elsevier Handbook of Statistics Series.
- Hilbe, J. M. (2016). “Astrostatistics as new statistical discipline – a historical perspective.” www.worldofstatistics.org/files/2016/05/WOS_newsletter_05252016.pdf (visited on 06/16/2016).
- Hillebrandt, W. and J. C. Niemeyer (2000). “Type Ia supernova explosion models.” *Ann. Rev. Astron. Astrophys.* 38(1), 191–230. DOI: 10.1146/annurev.astro.38.1.191. eprint: <http://dx.doi.org/10.1146/annurev.astro.38.1.191>.
- Ishida, E. E. O. and R. S. de Souza (2013). “Kernel PCA for type Ia supernovae photometric classification.” *Mon. Not. Roy. Astronom. Soc.* 430, 509–532. DOI: 10.1093/mnras/sts650. arXiv: 1201.6676.
- Ishida, E. E. O. et al. (2015). “COSMOABC: likelihood-free inference via population monte carlo approximate Bayesian computation.” *Astron. Comput.* 13, 1–11. DOI: 10.1016/j.ascom.2015.09.001. arXiv: 1504.06129.
- Isobe, T. et al. (1990). “Linear regression in astronomy.” *Astrophys. J.* 364, 104–113. DOI: 10.1086/169390.
- Jang-Condell, H. and L. Hernquist (2001). “First structure formation: a simulation of small-scale structure at high redshift.” *Astrophys. J.* 548(1), 68. <http://stacks.iop.org/0004-637X/548/i=1/a=68>
- Janson, M. et al. (2014). “The AstraLux Multiplicity Survey: extension to late M-dwarfs.” *Astrophys. J.* 789, 102. DOI: 10.1088/0004-637X/789/2/102. arXiv: 1406.0535 [astro-ph.SR].
- Kashyap, V. L. et al. (2002). “Flare heating in stellar coronae.” *Astrophys. J.* 580, 1118–1132. DOI: 10.1086/343869. eprint: astro-ph/0208546.
- Kauffmann, G. et al. (2003). “The host galaxies of active galactic nuclei.” *Mon. Not. Roy. Astronom. Soc.* 346, 1055–1077. DOI: 10.1111/j.1365-2966.2003.07154.x. eprint: astro-ph/0304239.
- Kelly, B. C. (2007). “Some aspects of measurement error in linear regression of astronomical data.” *Astrophys. J.* 665, 1489–1506. DOI: 10.1086/519947. arXiv: 0705.2774.
- Kessler, R. et al. (2010). “Results from the Supernova Photometric Classification Challenge.” *Publ. Astronom. Soc. Pacific* 122, 1415–1431. DOI: 10.1086/657607. arXiv: 1008.1024 [astro-ph.CO].
- Kewley, L. J. et al. (2001). “Theoretical modeling of starburst galaxies.” *Astrophys. J.* 556, 121–140. DOI: 10.1086/321545. eprint: astro-ph/0106324.
- Killedar, M. et al. (2015). “Weighted ABC: a new strategy for cluster strong lensing cosmology with simulations.” arXiv: 1507.05617[astro-ph].
- Kravtsov, A. V. and S. Borgani (2012). “Formation of galaxy clusters.” *Ann. Rev. Astron. Astrophys.* 50, 353–409. DOI: 10.1146/annurev-astro-081811-125502. arXiv: 1205.5556 [astro-ph.CO].
- Kroupa, P. (2001). “On the variation of the initial mass function.” *Mon. Not. Roy. Astronom. Soc.* 322, 231–246. DOI: 10.1046/j.1365-8711.2001.04022.x. eprint: arXiv: astro-ph/0009005.
- Kruijssen, J. M. D. (2014). “Globular cluster formation in the context of galaxy formation and evolution.” *Classical Quant. Grav.* 31(24), 244006. DOI: 10.1088/0264-9381/31/24/244006. arXiv: 1407.2953.
- Kuo, L. and B. Mallick (1998). “Variable selection for regression models.” *Sankhyā: Indian J. Statist., Series B (1960–2002)* 60(1), 65–81. www.jstor.org/stable/25053023.
- Lansbury, G. B. et al. (2014). “Barred S0 galaxies in the Coma cluster.” *Mon. Not. Roy. Astronom. Soc.* 439(2), 1749–1764.
- Lin, C.-A. and M. Kilbinger (2015). “A new model to predict weak-lensing peak counts II. Parameter constraint strategies.” arXiv: 1506.01076.
- Lintott, C. J. et al. (2008). “Galaxy Zoo: morphologies derived from visual inspection of galaxies from the Sloan Digital Sky Survey.” *Mon. Not. Roy. Astronom. Soc.* 389, 1179–1189. DOI: 10.1111/j.1365-2966.2008.13689.x. arXiv: 0804.4483.
- Lynden-Bell, D. (1969). “Galactic nuclei as collapsed old quasars.” *Nature* 223, 690–694. DOI: 10.1038/223690a0.
- Ma, C. et al. (2016). “Application of Bayesian graphs to SN Ia data analysis and compression.” *Mon. Not. Roy. Astronom. Soc.* (preprint). arXiv: 1603.08519.
- Macciò, A. V. et al. (2007). “Concentration, spin and shape of dark matter haloes: scatter and the dependence on mass and environment.” *Mon. Not. Roy. Astronom. Soc.* 378, 55–71. DOI: 10.1111/j.1365-2966.2007.11720.x. eprint: arXiv:

- astro-ph/0608157.
- Machida, M. N. *et al.* (2008). “Formation scenario for wide and close binary systems.” *Astrophys. J.* 677, 327–347. DOI: 10.1086/529133. arXiv: 0709.2739.
- Mahajan, S. and S. Raychaudhury (2009). “Red star forming and blue passive galaxies in clusters.” *Mon. Not. Roy. Astronom. Soc.* 400, 687–698. DOI: 10.1111/j.1365-2966.2009.15512.x. arXiv: 0908.2434.
- Maio, U. *et al.* (2010). “The transition from population III to population II-I star formation.” *Mon. Not. Roy. Astronom. Soc.* 407, 1003–1015. DOI: 10.1111/j.1365-2966.2010.17003.x. arXiv: 1003.4992 [astro-ph.CO].
- Maio, U. *et al.* (2011). “The interplay between chemical and mechanical feedback from the first generation of stars.” *Mon. Not. Roy. Astronom. Soc.* 414, 1145–1157. DOI: 10.1111/j.1365-2966.2011.18455.x. arXiv: 1011.3999[astro-ph.CO].
- Mandel, K. S. *et al.* (2011). “Type Ia supernova light curve inference: hierarchical models in the optical and near-infrared.” *Astrophys. J.* 731, 120. DOI: 10.1088/0004-637X/731/2/120. arXiv: 1011.5910.
- Maoz, D. *et al.* (2014). “Observational clues to the progenitors of type Ia supernovae.” *Ann. Rev. Astron. Astrophys.* 52(1), 107–170. DOI: 10.1146/annurev-astro-082812-141031.
- Marley, J. and M. Wand (2010). “Non-standard semiparametric regression via BRugs.” *J. Statist. Software* 37(1), 1–30. DOI: 10.18637/jss.v037.i05.
- Masters, K. L. *et al.* (2010). “Galaxy Zoo: passive red spirals.” *Mon. Not. Roy. Astronom. Soc.* 405, 783–799. DOI: 10.1111/j.1365-2966.2010.16503.x. arXiv: 0910.4113.
- McCullagh, P. (2002). “What is a statistical model?” *Ann. Statist.* 30(5), 1225–1310. DOI: 10.1214/aos/1035844977.
- Merritt, D. (2000). “Black holes and galaxy evolution.” *Dynamics of Galaxies: from the Early Universe to the Present*, eds. F. Combes, G. A. Mamon, and V. Charmandaris Vol. 197. Astronomical Society of the Pacific Conference Series, p. 221. eprint: astro-ph/9910546.
- Merritt, D. and L. Ferrarese (2001). “Black hole demographics from the $M-\sigma$ relation.” *Mon. Not. Roy. Astronom. Soc.* 320, L30–L34. DOI: 10.1046/j.1365-8711.2001.04165.x. eprint: astro-ph/0009076.
- Metropolis, N. and S. Ulam (1949). “The Monte Carlo method.” *J. Amer. Statist. Assoc.* 44(247), 335–341. www.jstor.org/stable/2280232.
- Metropolis, N. *et al.* (1953). “Equation of state calculations by fast computing machines.” *J. Chem. Phys.* 21, 1087–1092.
- Mignoli, M. *et al.* (2009). “The zCOSMOS redshift survey: the three-dimensional classification cube and bimodality in galaxy physical properties.” *Astron. Astrophys.* 493, 39–49. DOI: 10.1051/0004-6361:200810520. arXiv: 0810.2245.
- Nelder, J. A. and R. W. M. Wedderburn (1972). “Generalized linear models.” *J. Royal Statist. Soc., Series A* 135, 370–384.
- O’Hara, R. B. and D. J. Kotze (2010). “Do not log-transform count data.” *Meth. Ecology Evol.* 1(2), 118–122. DOI: 10.1111/j.2041-210X.2010.00021.x.
- O’Hara, R. B. and M. J. Sillanpää (2009). “A review of Bayesian variable selection methods: what, how and which.” *Bayesian Anal.* 4(1), 85–117. DOI: 10.1214/09-ba403.
- Oliveira, J. M. *et al.* (2005). “Circumstellar discs around solar mass stars in NGC 6611.” *Mon. Not. Roy. Astronom. Soc.* 358, L21–L24. DOI: 10.1111/j.1745-3933.2005.00020.x. eprint: astro-ph/0501208.
- Orban de Xivry, G. *et al.* (2011). “The role of secular evolution in the black hole growth of narrow-line Seyfert 1 galaxies.” *Mon. Not. Roy. Astronom. Soc.* 417, 2721–2736. DOI: 10.1111/j.1365-2966.2011.19439.x. arXiv: 1104.5023.
- Park, T. and G. Casella (2008). “The Bayesian lasso.” *J. Amer. Statist. Assoc.* 103(482), 681–686. DOI: 10.1198/016214508000000337.
- Pawlak, M. (2016). “Period-luminosity-colour relation for early-type contact binaries.” *Mon. Not. Roy. Astronom. Soc.* DOI: 10.1093/mnras/stw269. arXiv: 1602.01467 [astro-ph.SR].
- Penna-Lima, M. *et al.* (2014). “Biases on cosmological parameter estimators from galaxy cluster number counts.” *J. Cosmol. Astroparticle Phys.* 5, 039. DOI: 10.1088/1475-7516/2014/05/039. arXiv: 1312.4430.
- Perlmutter, S. *et al.* (1999) “Measurements of Ω and Λ from 42 high-redshift supernovae.” *Astrophys. J.* 517, 565–586. DOI: 10.1086/307221. eprint: astro-ph/9812133.
- Peterson, B. M. (2008). “The central black hole and relationships with the host galaxy.” *New Astron. Rev.* 52, 240–252. DOI: 10.1016/j.newar.2008.06.005.
- Pimbblet, K. A. *et al.* (2013). “The drivers of AGN activity in galaxy clusters: AGN fraction as a function of mass and environment.” *Mon. Not. Roy. Astronom. Soc.* 429, 1827–1839. DOI: 10.1093/mnras/sts470. arXiv: 1212.0261.
- Raichoor, A. and S. Andreon (2014). “Do cluster properties affect the quenching rate?” *Astron. Astrophys.* 570, A123. DOI: 10.1051/0004-6361/201424050. arXiv: 1409.4416.
- Rhode, K. L. (2012). “Exploring the correlations between globular cluster populations and supermassive black holes in giant galaxies.” *Astronom. J.* 144, 154. DOI: 10.1088/0004-6256/144/5/154. arXiv: 1210.4570 [astro-ph.CO].
- Richardson, S. and W. R. Gilks (1993). “A Bayesian approach to measurement error problems in epidemiology using conditional independence models.” *Amer. J. Epidemiology* 138(6), 430–442. eprint: https://aje.oxfordjournals.org/content/138/6/430.full.pdf+html.
- Riess, A. G. *et al.* (1998). “Observational evidence from supernovae for an accelerating universe and a cosmological constant.” *Astronom. J.* 116, 1009–1038. DOI: 10.1086/300499. eprint: astro-ph/9805201.
- Robin, A. C. *et al.* (2014). “Constraining the thick disc formation scenario of the Milky Way.” *Astron. Astrophys.* 569. arXiv: 1406.5384.
- Rubin, D. B. (1984). “Bayesianly justifiable and relevant frequency calculations for the applied statistician.” *Ann. Statist.*

- 12(4), 1151–1172. www.jstor.org/stable/2240995.
- Rubin, D. *et al.* (2015). “UNITY: Confronting supernova cosmology’s statistical and systematic uncertainties in a unified Bayesian framework.” *Astrophys. J.* 813, 137. DOI: 10.1088/0004-637X/813/2/137. arXiv: 1507.01602.
- Rucinski, S. M. (2004). “Contact binary stars of the W UMa-type as distance tracers.” *New Astron. Rev.* 48, 703–709. DOI: 10.1016/j.newar.2004.03.005. eprint: astro-ph/0311085.
- Rue, H. *et al.* (2009). “Approximate Bayesian inference for latent Gaussian models by using integrated nested Laplace approximations.” *J. Royal Statist. Soc. Series B* 71(2), 319–392. DOI: 10.1111/j.1467-9868.2008.00700.x.
- Sako, M. *et al.* (2014). “The data release of the Sloan Digital Sky Survey – II Supernova Survey.” arXiv: 1401.3317 [astro-ph.CO].
- Salpeter, E. E. (1955). “The luminosity function and stellar evolution.” *Astrophys. J.* 121, 161. DOI: 10.1086/145971.
- Sana, H. *et al.* (2012). “Binary interaction dominates the evolution of massive stars.” *Science* 337, 444. DOI: 10.1126/science.1223344. arXiv:1207.6397 [astro-ph.SR].
- Schafer, C. M. and P. E. Freeman (2012). “Likelihood-free inference in cosmology: potential for the estimation of luminosity.” *Statistical Challenges in Modern Astronomy V*, eds. E. D. Feigelson and B. G. Jogesh, pp. 3–19. Springer.
- Schawinski, K. *et al.* (2007). “Observational evidence for AGN feedback in early-type galaxies.” *Mon. Not. Roy. Astronom. Soc.* 382, 1415–1431. DOI: 10.1111/j.1365-2966.2007.12487.x. arXiv: 0709.3015.
- Schwarz, G. (1978). “Estimating the dimension of a model.” *Ann. Statist.* 6(2), 461–464.
- Shariff, H. *et al.* (2015). “BAHAMAS: new SNIa analysis reveals inconsistencies with standard cosmology.” arXiv: 1510.05954.
- Shimizu, T. T. and R. F. Mushotzky (2013). “The first hard X-ray power spectral density functions of active galactic nucleus.” *Astrophys. J.* 770, 60. DOI: 10.1088/0004-637X/770/1/60. arXiv: 1304.7002 [astro-ph.HE].
- Snyder, G. F. *et al.* (2011). “Relation between globular clusters and supermassive black holes in ellipticals as a manifestation of the black hole fundamental plane.” *Astrophys. J.* 728, L24. DOI: 10.1088/2041-8205/728/1/L24. arXiv: 1101.1299 [astro-ph.CO].
- Somerville, R. S. *et al.* (2008). “A semi-analytic model for the co-evolution of galaxies, black holes and active galactic nuclei.” *Mon. Not. Roy. Astronom. Soc.* 391, 481–506. DOI: 10.1111/j.1365-2966.2008.13805.x. arXiv: 0808.1227.
- Spiegelhalter, D. J. *et al.* (2002). “Bayesian measures of model complexity and fit.” *J. Royal Statist. Soc., Series B* 64(4), 583–639. DOI: 10.1111/1467-9868.00353.
- Stan (2016). “Prior choice recommendations.” <https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations> (visited on 06/27/2016).
- Sunyaev, R. A. and Y. B. Zeldovich (1972). “The observations of relic radiation as a test of the nature of X-ray radiation from the clusters of galaxies.” *Comm. Astrophys. Space Phys.* 4, 173.
- Tanner, M. A. and W. H. Wong (1987). “The calculation of posterior distributions by data augmentation.” *J. Amer. Statist. Assoc.* 82, 528–540.
- Tibshirani, R. (1996). “Regression shrinkage and selection via the lasso.” *J. Royal Statist. Soc. Series B* 58, 267–288.
- Tremaine, S. *et al.* (2002). “The slope of the black hole mass versus velocity dispersion correlation.” *Astrophys. J.* 574, 740–753. DOI: 10.1086/341002. eprint: astro-ph/0203468.
- Uemura, M. *et al.* (2015). “Variable selection for modeling the absolute magnitude at maximum of Type Ia supernovae.” *PASJ* 67, 55. DOI: 10.1093/pasj/psv031. arXiv: 1504.01470 [astro-ph.SR].
- Uttley, P. *et al.* (2002). “Measuring the broad-band power spectra of active galactic nuclei with RXTE.” *Mon. Not. Roy. Astronom. Soc.* 332, 231–250. DOI: 10.1046/j.1365-8711.2002.05298.x. eprint: astro-ph/0201134.
- Vaquero, J. M. (2007). “Historical sunspot observations: a review.” *Adv. Space Res.* 40, 929–941. DOI: 10.1016/j.asr.2007.01.087. eprint: astro-ph/0702068.
- Vehtari, A. and J. Ojanen (2012). “A survey of Bayesian predictive methods for model assessment, selection and comparison.” *Statist. Surv.* 6, 142–228. DOI: 10.1214/12-SS102.
- Vitenti, S. D. P. and M. Penna-Lima (2014). “NumCosmo: numerical cosmology.” ASCL: 1408.013.
- Wang, H. *et al.* (2011). “Internal properties and environments of dark matter haloes.” *Mon. Not. Roy. Astronom. Soc.* 413, 1973–1990. DOI: 10.1111/j.1365-2966.2011.18301.x. arXiv: 1007.0612 [astro-ph.CO].
- Weyant, A. *et al.* (2013). “Likelihood-free cosmological inference with Type Ia supernovae: approximate Bayesian computation for a complete treatment of uncertainty.” *Astrophys. J.* 764, 116. DOI: 10.1088/0004-637X/764/2/116. arXiv: 1206.2563 [astro-ph.CO].
- White, L. A. (2014). “The rise of astrostatistics.” www.symmetrymagazine.org/article/november-2014/the-rise-of-astrostatistics (visited on 06/16/2016).
- Wolf, R. C. *et al.* (2016). “SDSS-II Supernova Survey: an analysis of the largest sample of Type Ia supernovae and correlations with host-galaxy spectral properties.” *Astrophys. J.* 821, 115. DOI: 10.3847/0004-637X/821/2/115. arXiv: 1602.02674.
- Zaninetti, L. (2013). “The initial mass function modeled by a left truncated beta distribution.” *Astrophys. J.* 765, 128. DOI: 10.1088/0004-637X/765/2/128. arXiv: 1303.5597 [astro-ph.SR].
- Zuur, A. F., J. M. Hilbe, and E. N. Ieno (2013), *A Beginner’s Guide to GLM and GLMM with R: A frequentist and Bayesian perspective for ecologists*, Newburgh, UK: Highlands.

Index

- active galactic nuclei (AGN), 325–332
Akaike information criterion (AIC), 101, 106, 137, 224, 262–266, 379
approximate Bayesian computation (ABC), 276, 355–361
astronomical applications, 276–363
astrophysics, 1–5, 8, 364
Astrostatistics and Astroinformatics Portal (ASAIP), 5, 364
- Bayes' theorem, 27–29
Bayesian hierarchical models, 215–261
Bayesian information criterion (BIC), 101, 106, 137, 224, 262, 263
Bernoulli model, 7, 8, 32–36, 54, 71, 72, 74, 118, 125, 134, 140, 184, 186, 196, 202, 212, 234, 307–312, 325–334
Bernoulli trials, 148
beta binomial model, 7, 125–134
beta model, 7, 32–38, 75, 92–98, 302–307
beta prior, 32–38, 94, 267
binaries, 290–297
binomial models, 98–117, 235–240
black hole, 277, 278, 313, 325
boundary likelihood ratio test, 139
- canonical link, 72, 84, 150
censored binary component, 197, 202
classification of statistical models, 7
complementary loglog link, 99, 104, 111, 118, 125, 126, 134, 202, 206
confidence interval, 27, 29–31, 45, 49, 142
continuous models, 46–67, 74–98, 283–302
correlation, 2, 6, 23, 72, 106, 125, 149, 161, 190, 215, 219, 253, 267–269, 277, 284, 291, 303, 307, 333
cosmology, 283, 333, 347, 361
count models, 6, 44, 70, 71, 77, 134, 135, 140, 142, 149, 153, 184, 206, 240, 369, 376
CRAN, 10, 88, 95, 101, 136, 140, 152, 174, 220, 240, 258
credible interval, 12, 29–31, 43, 45, 49, 53, 92, 101, 106, 122, 125, 129, 142, 161, 186, 222, 263, 287, 300, 310, 336, 345, 356
- deviance information criterion (DIC), 106, 111, 176, 197, 224, 264–266, 319, 366
deviance statistic (Bayesian), 106, 202, 263–265
dispersion
 overdispersion, 44, 72, 125, 139, 141, 142, 149, 150, 152, 153, 161, 171, 180, 197, 220, 240, 253, 317, 323
 parameter, 6, 44, 69–72, 139, 149–152, 154, 158, 161, 164–165, 167, 176, 180, 191, 193, 202, 253, 257–259, 315, 317, 320, 321, 323, 324, 366, 368, 373, 374
 Pearson, 140, 167, 253
 statistic, 106, 125, 139, 140, 149, 153, 165, 167, 171, 240, 253, 258, 316, 323, 367, 379
 underdispersion, 125, 139–142, 149, 180, 241, 379
distribution, probability
 Bernoulli, 7, 33, 34, 71, 98, 99, 112, 118, 196, 234, 307, 308, 311, 325, 327, 333
 beta, 34, 37, 75, 92, 94, 303
 beta binomial, 126, 127
 binomial, 71, 98, 99, 118, 307

gamma, 20, 82, 83, 148, 154, 157
Gaussian, 46–48, 68, 271, 284, 356–358
generalized Poisson, 77, 134, 136, 164, 165
geometric, 71, 72
inverse Gaussian, 75, 87, 88
lognormal, 75, 297, 298, 300, 334
NB-P, 180
negative binomial, 71, 148, 150, 154, 176, 182, 323, 346
Poisson, 54, 71, 135, 141, 149, 150, 154, 165, 170, 171, 184–190, 219, 253
Poisson inverse Gaussian (PIG), 149, 253
double-exponential, 274

errors in measurements, 62, 277, 314, 353, 363

frequentist statistics, 2, 3, 6–8, 23, 29, 43, 44, 46, 49, 73, 75, 77, 153, 217, 219, 220, 248, 262–265, 377

galaxies, 34, 106, 277, 278, 298, 302, 303, 307, 308, 313, 314, 325–327, 329, 333
gamma-logit model, 206
gamma models, 82, 149, 206
generalized
 additive models (GAM), 366, 368
 linear mixed models (GLMMs), 217, 222, 230
 linear models (GLMs), 68–75, 111, 113, 120, 136, 238, 313, 340, 366
 Poisson models, 134, 136, 139, 140, 149, 161, 164–167, 206, 253, 368, 369, 374, 376
globular clusters, 35, 37, 135, 313, 314
goodness-of-fit, 2, 106, 137, 263

halos, 196, 313, 329, 333
hierarchical GLMM models, 217
 binary logistic, 228, 229
 binomial logistic, 235, 238
 Gaussian, 219
 negative binomial, 252
 Poisson, 240
Hubble residuals, 283, 284, 287
hurdle model, 141, 170, 185, 190, 196, 197, 202, 210, 332, 333, 337
generalized Poisson, 202
log-gamma hurdle, 206
log-gamma-logit hurdle, 206
log-normal-logit hurdle, 333, 334
negative binomial, 190, 202
Poisson, 170, 207
Poisson-logit, 197, 200, 206
two-part, 134, 141, 206

initial mass function, 297, 298
integrated nested Laplace approximation (INLA), 9, 10, 31, 366, 368
International Astronomical Union (IAU), 3, 5, 341, 364
International Astrostatistics Association (IAA), i, 5, 364
inverse Gaussian model, 71, 87, 88, 265
inverse link, 70, 104, 113, 121, 126, 130, 134, 137, 218

JAGS software, 12–14

Kuo and Mallick selection method, 267–269

least absolute shrinkage and selection operator (LASSO), 274–275
linear regression, 3, 11, 16, 24, 41
log-gamma model, 71, 75, 82
lognormal model, 75, 210, 298

Markov chain Monte Carlo (MCMC), 4, 31, 43, 53, 101, 200, 361, 366

maximum likelihood estimate, 25, 33, 140, 262, 370

multivariate normal model, 58, 292

negative binomial model, 44, 68, 136, 148, 150, 154, 164, 167, 176, 180, 317, 373

negative binomial model (GLMMs), 252

negative binomial three-parameter model, 179, 313

normal models, 38, 41, 46–48, 68, 75, 219, 277, 350

odds ratio, 111, 329

offsets, 369

ordinary differential equation (ODE), 349

parameter

alpha, 156, 158, 161, 202, 257, 258

delta, 165, 259, 368

dispersion, 44, 69, 149, 150, 180, 193, 257, 259, 315, 317, 323, 373

scale, 6, 30, 33, 44, 52, 68, 71, 75, 94, 226, 298, 357

variance, 44, 68, 80, 90, 317

pD, 265

Pearson χ^2 statistic, 106, 139, 240, 253

Pearson dispersion, 140, 167, 253

Poisson model, 44, 71, 135, 139, 314

Poisson-logit hurdle model, 206

posterior distribution, 2, 7, 19, 27, 43, 48, 49, 68, 79, 140, 215, 219, 250, 263, 316, 329, 336, 356

predicted values, 52, 74, 98, 113, 129, 137, 218, 377

priors

diffuse, 30, 33, 38, 44, 82, 89, 265, 370, 372, 374

flat, 33, 38

gamma, 89, 298

informative (explanation of), 30, 82, 89, 96, 265, 267, 271, 368

inverse gamma, 327

non-informative (explanation of), 30, 32, 33, 38, 44, 45, 142, 327

uniform, 33, 52, 321

Probit model, 110, 124

Python, 14

quadrature, 49

R, 10

R2jags, 112

Rate parameter, 112, 369

residuals, 23, 139, 150, 152, 166, 218, 222, 253, 340, 341, 344, 377

Schwarz criterion (SC), 262

sensitivity test, 89

software for modeling, 4

spatial analysis, 368

Stan software, 17

standard errors, 106, 140, 153

standardized residuals, 12

supernovae, 283, 347

time series model, 340

truncated model, 169

Uniform prior, 19, 25, 38, 50, 89

VGAM, 373

zero-altered models, 196

zero-inflated models, 184
negative binomial, 190
Poisson, 184
zero-truncated models, 169
negative binomial, 176
Poisson, 170