

Bayesian Analysis with Python

Second Edition

Introduction to statistical modeling and probabilistic programming using PyMC3 and ArviZ



Packt

www.packt.com

Osvaldo Martin

Bayesian Analysis with Python

Second Edition

Introduction to statistical modeling and probabilistic
programming using PyMC3 and ArviZ

Osvaldo Martin

Packt

BIRMINGHAM - MUMBAI

Bayesian Analysis with Python

Second Edition

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Pavan Ramchandani

Acquisition Editor: Joshua Nadar

Content Development Editor: Unnati Guha

Technical Editor: Sayli Nikalje

Copy Editor: Safis Editing

Project Coordinator: Manthan Patel

Proofreader: Safis Editing

Indexer: Priyanka Dhadke

Graphics: Jisha Chirayil

Production Coordinator: Arvindkumar Gupta

First published: November 2016

Second edition: December 2018

Production reference: 2261219

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78934-165-2

www.packtpub.com

I dedicate this book to Abril.



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Foreword

Probabilistic programming is a framework that allows you to flexibly build Bayesian statistical models in computer code. Once built, powerful inference algorithms that work independently of the model you formulated can be used to fit your model to data. This combination of flexible model specification and automatic inference provides a powerful tool for the researcher to quickly build, analyze, and iteratively improve novel statistical models. This iterative approach is in stark contrast to the way Bayesian models were fitted to data before: previous inference algorithms usually only worked for one specific model. Not only did this require strong mathematical skills to formulate the model and devise an inference scheme, it also considerably slowed down the iterative cycle: change the model, re-derive your inference. Probabilistic programming thus democratizes statistical modeling by considerably lowering the mathematical understanding and time required to successfully build novel models and gain unique insights into your data.

The idea behind probabilistic programming is not new: BUGS, the first of its kind, was first released in 1989. The kinds of model that could be fitted successfully were extremely limited and inference was slow, rendering these first-generation languages not very practical. Today, there are a multitude of probabilistic programming languages that are widely used in academia and at companies such as Google, Microsoft, Amazon, Facebook, and Uber to solve large and complex problems. What has changed? The key factor in lifting probabilistic programming from being a cute toy to the powerful engine that can solve complex large-scale problems is the advent of Hamiltonian Monte Carlo samplers, which are several orders of magnitude more powerful than previous sampling algorithms. While originally devised in 1987, only the more recent probabilistic programming systems named Stan and PyMC3 made these samplers widely available and usable.

This book will give you a practical introduction to this extremely powerful and flexible tool. It will have a big impact on how you think about and solve complex analytical problems. There are few people better suited to have written it than PyMC3 core developer Osvaldo Martin. Osvaldo has the rare talent of breaking complex topics down to make them easily digestible. His deep practical understanding, gained through hard-won experience, allows him to take you, the reader, on the most efficient route through this terrain, which could otherwise easily seem impenetrable. The visualizations and code examples make this book an eminently practicable resource through which you will gain an intuitive understanding of their theoretical underpinnings.

I also would like to commend you, dear reader, for having picked up this book. It is not the fast and easy route. In a time where headlines advertise deep learning as the technique to solve all current and future analytical problems, the more careful and deliberate approach of building a custom model for a specific purpose might not seem quite as attractive. However, you will be able to solve problems that can hardly be solved any other way.

This is not to say that deep learning is not an extremely exciting technique. In fact, probabilistic programming itself is not constrained to classic statistical models. Reading the current machine learning literature, you will find that Bayesian statistics is emerging as a powerful framework to express and understand next-generation deep neural networks. This book will thus equip you not only with the skills to solve hard analytical problems, but also to have a front-row seat in humanity's perhaps greatest endeavor: the development of artificial intelligence. Enjoy!

Thomas Wiecki, PhD

Head of Research at Quantopian.

Contributors

About the author

Osvaldo Martin is a researcher at The National Scientific and Technical Research Council (CONICET), in Argentina. He has worked on structural bioinformatics of protein, glycans, and RNA molecules. He has experience using Markov Chain Monte Carlo methods to simulate molecular systems and loves to use Python to solve data analysis problems.

He has taught courses about structural bioinformatics, data science, and Bayesian data analysis. He was also the head of the organizing committee of PyData San Luis (Argentina) 2017. He is one of the core developers of PyMC3 and ArviZ.

I would like to thank Romina for her continuous support. I also want to thank Walter Lapadula, Bill Engels, Eric J Ma, and Austin Rochford for providing invaluable feedback and suggestions on my drafts. A special thanks goes to the core developers and all contributors of PyMC3 and ArviZ. This book was possible only because of the dedication, love, and hard work they have put into these libraries and into building a great community around them.

About the reviewer

Eric J MA is a data scientist at the Novartis Institutes for Biomedical Research. There, he conducts biomedical data science research, with a focus on using Bayesian statistical methods in the service of making medicines for patients. Prior to Novartis, he was an Insight Health Data Fellow in the summer of 2017, and defended his doctoral thesis in the spring of 2017.

Eric is also an open source software developer, and has led the development of nxviz, a visualization package for NetworkX, and pyjanitor, a clean API for cleaning data in Python. In addition, he has made contributions to a range of open source tools, including PyMC3, Matplotlib, bokeh, and CuPy.

His personal life motto is found in the Luke 12:48.

Austin Rochford is a principal data scientist at Monetate Labs, where he develops products that allow retailers to personalize their marketing across billions of events a year. He is a mathematician by training and is a passionate advocate of Bayesian methods.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
Chapter 1: Thinking Probabilistically	6
Statistics, models, and this book's approach	7
Working with data	8
Bayesian modeling	9
Probability theory	10
Interpreting probabilities	10
Defining probabilities	11
Probability distributions	12
Independently and identically distributed variables	16
Bayes' theorem	18
Single-parameter inference	21
The coin-flipping problem	21
The general model	22
Choosing the likelihood	22
Choosing the prior	24
Getting the posterior	26
Computing and plotting the posterior	27
The influence of the prior and how to choose one	30
Communicating a Bayesian analysis	32
Model notation and visualization	32
Summarizing the posterior	33
Highest-posterior density	33
Posterior predictive checks	34
Summary	36
Exercises	38
Chapter 2: Programming Probabilistically	40
Probabilistic programming	41
PyMC3 primer	42
Flipping coins the PyMC3 way	43
Model specification	43
Pushing the inference button	44
Summarizing the posterior	45
Posterior-based decisions	47
ROPE	47
Loss functions	49
Gaussians all the way down	52
Gaussian inferences	52
Robust inferences	58

Student's t-distribution	59
Groups comparison	
Cohen's d	64
Probability of superiority	66
The tips dataset	67
Hierarchical models	
Shrinkage	68
One more example	72
Summary	76
Exercises	80
Chapter 3: Modeling with Linear Regression	84
Simple linear regression	85
The machine learning connection	88
The core of the linear regression models	89
Linear models and high autocorrelation	94
Modifying the data before running	95
Interpreting and visualizing the posterior	97
Pearson correlation coefficient	100
Pearson coefficient from a multivariate Gaussian	101
Robust linear regression	104
Hierarchical linear regression	109
Correlation, causation, and the messiness of life	115
Polynomial regression	116
Interpreting the parameters of a polynomial regression	119
Polynomial regression – the ultimate model?	119
Multiple linear regression	120
Confounding variables and redundant variables	125
Multicollinearity or when the correlation is too high	128
Masking effect variables	133
Adding interactions	135
Variable variance	136
Summary	140
Exercises	140
Chapter 4: Generalizing Linear Models	143
Generalized linear models	144
Logistic regression	145
The logistic model	146
The Iris dataset	147
The logistic model applied to the iris dataset	150
Multiple logistic regression	153
The boundary decision	153
Implementing the model	154
Interpreting the coefficients of a logistic regression	155
Dealing with correlated variables	158

Dealing with unbalanced classes	160
Softmax regression	162
Discriminative and generative models	164
Poisson regression	167
Poisson distribution	167
The zero-inflated Poisson model	169
Poisson regression and ZIP regression	171
Robust logistic regression	174
The GLM module	175
Summary	176
Exercises	177
Chapter 5: Model Comparison	179
Posterior predictive checks	180
Occam's razor – simplicity and accuracy	185
Too many parameters leads to overfitting	187
Too few parameters leads to underfitting	189
The balance between simplicity and accuracy	189
Predictive accuracy measures	190
Cross-validation	191
Information criteria	192
Log-likelihood and deviance	192
Akaike information criterion	193
Widely applicable Information Criterion	194
Pareto smoothed importance sampling leave-one-out cross-validation	194
Other Information Criteria	195
Model comparison with PyMC3	195
A note on the reliability of WAIC and LOO computations	198
Model averaging	198
Bayes factors	202
Some remarks	203
Computing Bayes factors	204
Common problems when computing Bayes factors	207
Using Sequential Monte Carlo to compute Bayes factors	208
Bayes factors and Information Criteria	209
Regularizing priors	212
WAIC in depth	213
Entropy	214
Kullback-Leibler divergence	216
Summary	219
Exercises	220
Chapter 6: Mixture Models	221
Mixture models	222
Finite mixture models	223
The categorical distribution	225

The Dirichlet distribution	226
Non-identifiability of mixture models	230
How to choose K	232
Mixture models and clustering	237
Non-finite mixture model	237
Dirichlet process	238
Continuous mixtures	246
Beta-binomial and negative binomial	246
The Student's t-distribution	247
Summary	248
Exercises	249
Chapter 7: Gaussian Processes	250
Linear models and non-linear data	251
Modeling functions	252
Multivariate Gaussians and functions	254
Covariance functions and kernels	254
Gaussian processes	257
Gaussian process regression	258
Regression with spatial autocorrelation	265
Gaussian process classification	271
Cox processes	279
The coal-mining disasters	279
The redwood dataset	282
Summary	286
Exercises	286
Chapter 8: Inference Engines	288
Inference engines	289
Non-Markovian methods	290
Grid computing	290
Quadratic method	293
Variational methods	295
Automatic differentiation variational inference	298
Markovian methods	298
Monte Carlo	300
Markov chain	302
Metropolis-Hastings	302
Hamiltonian Monte Carlo	307
Sequential Monte Carlo	308
Diagnosing the samples	311
Convergence	312
Monte Carlo error	317
Autocorrelation	317
Effective sample sizes	318

Table of Contents

Divergences	319
Non-centered parameterization	322
Summary	323
Exercises	323
Chapter 9: Where To Go Next?	325
Other Books You May Enjoy	329
Index	332

Preface

Bayesian statistics has been developing for more than 250 years. During this time, it has enjoyed as much recognition and appreciation as it has faced disdain and contempt. Throughout the last few decades, it has gained more and more attention from people in statistics and almost all the other sciences, engineering, and even outside the boundaries of the academic world. This revival has been possible due to theoretical and computational advancements developed mostly throughout the second half of the 20th century. Indeed, modern Bayesian statistics is mostly computational statistics. The necessity for flexible and transparent models and a more intuitive interpretation of statistical models and analysis has only contributed to the trend.

In this book we will adopt a pragmatic approach to Bayesian statistics and we will not care too much about other statistical paradigms and their relationships with Bayesian statistics. The aim of this book is to learn how to do Bayesian data analysis; philosophical discussions are interesting, but they have already been undertaken elsewhere in a richer way that is simply outside the scope of these pages.

We will take a modeling approach to statistics, learn how to think in terms of probabilistic models, and apply Bayes' theorem to derive the logical consequences of our models and data. The approach will also be computational; models will be coded using PyMC3, a library for Bayesian statistics that hides most of the mathematical details and computations from the user, and ArviZ, a Python package for exploratory analysis of Bayesian models.

Bayesian methods are theoretically grounded in probability theory, and so it's no wonder that many books about Bayesian statistics are full of mathematical formulas requiring a certain level of mathematical sophistication. Learning the mathematical foundations of statistics could certainly help you build better models and gain intuition about problems, models, and results. Nevertheless, libraries such as PyMC3 allow us to learn and do Bayesian statistics with only a modest amount of mathematical knowledge, as you will be able to verify yourself throughout this book.

Who this book is for

If you are a student, data scientist, researcher in the natural or social sciences, or a developer looking to get started with Bayesian data analysis and probabilistic programming, this book is for you. The book is introductory, so no previous statistical knowledge is required, although some experience in using Python and NumPy is expected.

What this book covers

Chapter 1, *Thinking Probabilistically*, covers the basic concepts of Bayesian statistics and its implications for data analysis. This chapter contains most of the foundational ideas used in the rest of the book.

Chapter 2, *Programming Probabilistically*, revisits the concepts from the previous chapter from a more computational perspective. The PyMC3 probabilistic programming library is introduced, as well as ArviZ, a Python library for exploratory analysis of Bayesian models. Hierarchical models are explained with a couple of examples.

Chapter 3, *Modeling with Linear Regression*, covers the basic elements of linear regression, a very widely used model and the building block of more complex models.

Chapter 4, *Generalizing Linear Models*, covers how to expand linear models with other distributions than the Gaussian, opening the door to solving many data analysis problems.

Chapter 5, *Model Comparison*, discusses how to compare, select, and average models using WAIC, LOO, and Bayes factors. The general caveats of these methods are discussed.

Chapter 6, *Mixture Models*, discusses how to add flexibility to models by mixing simpler distributions to build more complex ones. The first non-parametric model in the book is also introduced: the Dirichlet process.

Chapter 7, *Gaussian Processes*, cover the basic idea behind Gaussian processes and how to use them to build non-parametric models over functions for a wide array of problems.

Chapter 8, *Inference Engines*, provides an introduction to methods for numerically approximating the posterior distribution, as well as a very important topic from the practitioner's perspective: how to diagnose the reliability of the approximated posterior.

Chapter 9, *Where To Go Next?*, provides a list of resources for you to keep learning from beyond this book, and a very short *farewell speech*.

To get the most out of this book

The code in the book was written using Python version 3.6. To install Python and Python libraries, I recommend using Anaconda, a scientific computing distribution. You can read more about Anaconda and download it at <https://www.anaconda.com/download/>. This will install many useful Python packages on your system. You will need to install two more packages. To install PyMC3 please use conda:

```
conda install -c conda-forge pymc3
```

And for ArviZ you can do it with the following command:

```
pip install arviz
```

An alternative way to install the necessary packages, once Anaconda is installed in your system, is to go to <https://github.com/aloctavodia/BAP> and download the environment file named `bap.yml`. Using it, you can install all the necessary packages by doing the following:

```
conda env create -f bap.yml
```

The Python packages used to write this book are listed here:

- IPython 7.0
- Jupyter 1.0 (or Jupyter-lab 0.35)
- NumPy 1.14.2
- SciPy 1.1
- pandas 0.23.4
- Matplotlib 3.0.2
- Seaborn 0.9.0
- ArviZ 0.3.1
- PyMC3 3.6

The code presented in each chapter assumes that you have imported at least some of these packages. Instead of copying and pasting the code from the book, I recommend downloading the code from <https://github.com/aloctavodia/BAP> and running it using Jupyter Notebook (or Jupyter Lab). I will keep this repository updated for new releases of PyMC3 or ArviZ. If you find a technical problem running the code in this book, a typo in the text, or any other mistake, please fill an issue in that repository and I will try to solve it as soon as possible.

Most figures in this book are generated using code. A common pattern you will find in this book is the following: a block of code immediately followed by a figure (generated from that code). I hope this pattern will look familiar to those of you using Jupyter Notebook/Lab, and I hope it does not appear annoying or confusing to anyone.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packt.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code and figures for the book are hosted on GitHub at <https://github.com/PacktPublishing/Bayesian-Analysis-with-Python-Second-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://www.packtpub.com/sites/default/files/downloads/9781789341652_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, filenames, or name of functions. Here is an example: "Most of the preceding code is for plotting; the *probabilistic part* is performed by the `y = stats.norm(mu, sd).pdf(x)` line."

A block of code is set as follows:

```
μ = 0.  
σ = 1.  
X = stats.norm(μ, σ)  
x = X.rvs(3)
```

Bold: Indicates a new term, an important word, or words that you see onscreen.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packt.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

For more information about Packt, please visit packt.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

Thinking Probabilistically

"Probability theory is nothing but common sense reduced to calculation."

- Pierre Simon Laplace

In this chapter, we will learn about the core concepts of Bayesian statistics and some of the instruments in the Bayesian toolbox. We will use some Python code, but this chapter will be mostly theoretical; most of the concepts we will see here will be revisited many times throughout this book. This chapter, being *heavy* on the theoretical side, may be a little anxiogenic for the coder in you, but I think it will ease the path in effectively applying Bayesian statistics to your problems.

In this chapter, we will cover the following topics:

- Statistical modeling
- Probabilities and uncertainty
- Bayes' theorem and statistical inference
- Single-parameter inference and the classic coin-flip problem
- Choosing priors and why people often don't like them, but should
- Communicating a Bayesian analysis

Statistics, models, and this book's approach

Statistics is about collecting, organizing, analyzing, and interpreting data, and hence statistical knowledge is essential for data analysis. Two main statistical methods are used in data analysis:

- **Exploratory Data Analysis (EDA):** This is about numerical summaries, such as the mean, mode, standard deviation, and interquartile ranges (this part of EDA is also known as **descriptive statistics**). EDA is also about visually inspecting the data, using tools you may be already familiar with, such as histograms and scatter plots.
- **Inferential statistics:** This is about making statements beyond the current data. We may want to understand some particular phenomenon, or maybe we want to make predictions for future (as yet unobserved) data points, or we need to choose among several competing explanations for the same observations. Inferential statistics is the set of methods and tools that will help us to answer these types of questions.



The focus of this book is upon how to perform Bayesian inferential statistics, and then how to use EDA to summarize, interpret, check, and communicate the results of Bayesian inference.

Most introductory statistical courses, at least for non-statisticians, are taught as a collection of recipes that more or less go like this: go to the statistical pantry, pick one tin can and open it, add data to taste, and stir until you obtain a consistent p-value, preferably under 0.05. The main goal in this type of course is to teach you how to pick the proper can. I never liked this approach, mainly because the most common result is a bunch of confused people unable to grasp, even at the conceptual level, the unity of the different *learned* methods. We will take a different approach: we will also learn some recipes, but this will be homemade rather than canned food; we will learn how to mix fresh ingredients that will suit different gastronomic occasions and, more importantly, that will let you to apply concepts far beyond the examples in this book.

Taking this approach is possible for two reasons:

- **Ontological:** Statistics is a form of modeling unified under the mathematical framework of probability theory. Using a probabilistic approach provides a unified view of what may seem like very disparate methods; statistical methods and **machine learning (ML)** methods look much more similar under the probabilistic lens.
- **Technical:** Modern software, such as PyMC3, allows practitioners, just like you and me, to define and solve models in a relative easy way. Many of these models were unsolvable just a few years ago or required a high level of mathematical and technical sophistication.

Working with data

Data is an essential ingredient in statistics and data science. Data comes from several sources, such as experiments, computer simulations, surveys, and field observations. If we are the ones in charge of generating or gathering the data, it is always a good idea to first think carefully about the questions we want to answer and which methods we will use, and only then proceed to get the data. In fact, there is a whole branch of statistics dealing with data collection, known as **experimental design**. In the era of the data deluge, we can sometimes forget that gathering data is not always cheap. For example, while it is true that the **Large Hadron Collider (LHC)** produces hundreds of terabytes a day, its construction took years of manual and intellectual labor.

As a general rule, we can think of the process generating the data as stochastic, because there is ontological, technical, and/or epistemic uncertainty, that is, the system is intrinsically stochastic, there are technical issues adding noise or restricting us from measuring with arbitrary precision, and/or there are conceptual limitations veiling details from us. For all these reasons, we always need to interpret data in the context of models, including mental and formal ones. Data does not speak but through models.

In this book, we will assume that we already have collected the data. Our data will also be clean and tidy, something rarely true in the *real world*. We will make these assumptions in order to focus on the subject of this book. I just want to emphasize, especially for newcomers to data analysis, that even when not covered in this book, these are important skills that you should learn and practice in order to successfully work with data.

A very useful skill when analyzing data is knowing how to write code in a programming language, such as Python. Manipulating data is usually necessary given that we live in a messy world with even messier data, and coding helps to get things done. Even if you are lucky and your data is very clean and tidy, coding will still be very useful since modern Bayesian statistics is done mostly through programming languages such as Python or R.

If you want to learn how to use Python for cleaning and manipulating data, I recommend reading the excellent book, *Python Data Science Handbook*, by Jake VanderPlas.

Bayesian modeling

Models are simplified descriptions of a given system or process that, for some reason, we are interested in. Those descriptions are deliberately designed to capture only the most relevant aspects of the system and not to explain every minor detail. This is one reason a more complex model is not always a better one.

There are many different kinds of models; in this book, we will restrict ourselves to Bayesian models. We can summarize the Bayesian modeling process using three steps:

1. Given some data and some assumptions on how this data could have been generated, we design a model by combining building blocks known as **probability distributions**. Most of the time these models are crude approximations, but most of the time is all we need.
2. We use Bayes' theorem to add data to our models and derive the logical consequences of combining the data and our assumptions. We say we are conditioning the model on our data.
3. We criticize the model by checking whether the model makes sense according to different criteria, including the data, our expertise on the subject, and sometimes by comparing several models.

In general, we will find ourselves performing these three steps in an iterative non-linear fashion. We will retrace our steps at any given point: maybe we made a silly coding mistake, or we found a way to change the model and improve it, or we realized that we need to add more data or collect a different kind of data.

Bayesian models are also known as **probabilistic models** because they are built using probabilities. Why probabilities? Because probabilities are the correct mathematical tool to model uncertainty, so let's take a walk through the garden of forking paths.

Probability theory

The title of this section may be a little bit pretentious as we are not going to learn probability theory in just a few pages, but that is not my intention. I just want to introduce a few general and important concepts that are necessary to better understand Bayesian methods, and that should be enough for understanding the rest of the book. If necessary, we will expand or introduce new probability-related concepts as we need them. For a detailed study of probability theory, I highly recommend the book, *Introduction to Probability* by Joseph K Blitzstein and Jessica Hwang. Another useful book could be *Mathematical Theory of Bayesian Statistics* by Sumio Watanabe, as the title says, the book is more Bayesian-oriented than the first, and also heavier on the mathematical side.

Interpreting probabilities

While probability theory is a mature and well-established branch of mathematics, there is more than one interpretation of probability. From a Bayesian perspective, a probability is a measure that quantifies the uncertainty level of a statement. Under this definition of probability, it is totally valid and *natural* to ask about the probability of life on Mars, the probability of the mass of the electron being 9.1×10^{-31} kg, or the probability of the 9th of July of 1816 being a sunny day in Buenos Aires. Notice, for example, that life on Mars exists or does not exist; the outcome is binary, a yes-no question. But given that we are not sure about that fact, a sensible course of action is trying to find out how likely life on Mars is. Since this definition of probability is related to our epistemic state of mind, people often call it the subjective definition of probability. But notice that any scientific-minded person will not use their personal beliefs, or the *information provided by an angel* to answer such a question, instead they will use all the relevant geophysical data about Mars, and all the relevant biochemical knowledge about necessary conditions for life, and so on. Therefore, Bayesian probabilities, and by extension Bayesian statistics, is as subjective (or objective) as any other well-established scientific method we have.

If we do not have information about a problem, it is reasonable to state that every possible event is equally likely, formally this is equivalent to assigning the same probability to every possible event. In the absence of information, our uncertainty is maximum. If we know instead that some events are more likely, then this can be formally represented by assigning higher probability to those events and less to the others. Notice than when we talk about events in *stats-speak*, we are not restricting ourselves to *things that can happen*, such as an asteroid crashing into Earth or my auntie's 60th birthday party; an event is just any of the possible values (or subset of values) a variable can take, such as the *event* that you are older than 30, or the price of a *Sachertorte*, or the number of bikes sold last year around the world.

The concept of probability it is also related to the subject of logic. Under Aristotelian or classical logic, we can only have statements that take the values of *true* or *false*. Under the Bayesian definition of probability, certainty is just a special case: a true statement has a probability of 1, a false statement has probability of 0. We would assign a probability of 1 to the statement, *There is Martian life*, only after having conclusive data indicating something is growing, and reproducing, and doing other activities we associate with living organisms. Notice, however, that assigning a probability of 0 is harder because we can always think that there is some Martian spot that is unexplored, or that we have made mistakes with some experiment, or several other reasons that could lead us to falsely believe life is absent on Mars even when it is not. Related to this point is Cromwell's rule, stating that we should reserve the use of the prior probabilities of 0 or 1 to logically true or false statements. Interestingly enough, Richard Cox mathematically proved that if we want to extend logic to include uncertainty, we must use probabilities and probability theory. Bayes' theorem is just a logical consequence of the rules of probability, as we will see soon. Hence, another way of thinking about Bayesian statistics is as an extension of logic when dealing with uncertainty, something that clearly has nothing to do with subjective reasoning in the pejorative sense—people often used the term *subjective*.

To summarize, using probability to model uncertainty is not necessarily related to the debate about whether nature is deterministic or random at its most fundamental level, nor is related to *subjective personal beliefs*. Instead, it is a purely methodological approach to model uncertainty. We recognize most phenomena are difficult to grasp because we generally have to deal with incomplete and/or noisy data, we are intrinsically limited by our evolution-sculpted primate brain, or any other sound reason you could add. As a consequence, we use a modeling approach that explicitly takes uncertainty into account.



From a practical point of view, the most relevant piece of information from this section is that Bayesian's use probabilities as a tool to quantify uncertainty.

Now that we've discussed the Bayesian interpretation of probability, let's learn about a few of the mathematical properties of probabilities.

Defining probabilities

Probabilities are numbers in the $[0, 1]$ interval, that is, numbers between 0 and 1, including both extremes. Probabilities follow a few rules; one of these rules is the product rule:

$$p(A, B) = p(A \mid B)p(B) \quad (1.1)$$

We read this as follows: the probability of A and B is equal to the probability of A given B , times the probability of B . The $p(A, B)$ expression represents the **joint probability** of A and B . The $p(A|B)$ expression is used to indicate a **conditional probability**; the name refers to the fact that the probability of A is *conditioned on* knowing B . For example, the probability that the pavement is wet is different from the probability that the pavement is wet if we know (or given that) it's raining. A conditional probability can be larger than, smaller than, or equal to the unconditioned probability. If knowing B does not provide us with information about A , then $p(A|B) = p(A)$. This will be true only if A and B are independent of each other. On the contrary, if knowing B gives us useful information about A , then the conditional probability could be larger or smaller than the unconditional probability, depending on whether knowing B makes A less or more likely. Let's see a simple example using a fair six-sided die. What is the probability of getting the number 3 if we roll the die, $p(\text{die} = 3)$? This is $1/6$ since each of the six numbers has the same chance for a fair six-sided die. And what is the probability of getting the number 3 given that we have obtained an odd number, $p(\text{die} = 3 | \text{die} = \text{odd})$? This is $1/3$, because if we know we have an odd number, the only possible numbers are $\{1, 3, 5\}$ and each of them has the same chance. Finally, what is the probability of $p(\text{die} = 3 | \text{die} = \text{even})$? This is 0, because if we know the number is even, then the only possible ones are $\{2, 4, 6\}$, and thus getting a 3 is not possible.

As we can see from this simple example, by conditioning on observed data we effectively change the probability of events, and with it, the uncertainty we have about them. Conditional probabilities are the heart of statistics, irrespective of your problem being rolling dice or building self-driving cars.

Probability distributions

A probability distribution is a mathematical object that describes how likely different events are. In general, these events are restricted somehow to a set of possible events, such as $\{1, 2, 3, 4, 5, 6\}$ for a die (we exclude undefined). A common and useful conceptualization in statistics is to think data is generated from some *true* probability distribution with unknown parameters. Then, inference is the process of finding out the values of those parameters using just a sample (also known as a **dataset**) from the *true* probability distribution. In general, we do not have access to the *true* probability distribution and thus we must resign ourselves and create a model in an attempt to somehow approximate that distribution. Probabilistic models are built by properly combining probability distributions.



Notice that, in general, we cannot be sure of whether our model is correct and thus we need to evaluate and criticize the models in order to gain confidence and convince others about our models being suitable for the problem we want to explore or solve.

If a variable, X , can be described by a probability distribution, then we call X a **random variable**. As a general rule, people use capital letters, such as X , to indicate the objects **random variable**, and x , to indicate an instance of that random variable. x can be a vector and thus contains many elements or individual values $x = (x_1, x_2, \dots, x_n)$. Let's see an example using Python; our true probability distribution will be a Normal (or Gaussian) distribution with means of $\mu = 0$ and $\sigma = 1$; these two parameters completely and unambiguously define a Normal distribution. Using SciPy, we can define the random variable, X , by writing `stats.norm(mu, sigma)` and we can get an instance, x , from it using the `rvs` (random variates) method. In the following example, we ask for three values:

```
μ = 0.  
σ = 1.  
X = stats.norm(μ, σ)  
x = X.rvs(3)
```

You will notice that each time you execute this code (in *stats-speak*: every *trial*), you will get a different *random* result. Notice that once the values of the parameters of a distributions are known, the probability of each x value is also known; what is random is the exact x values we will get at each trial. A common misconception with the meaning of random is that you can get any possible value from a random variable or that all values are equally likely. The allowed values of a random variable and their probabilities are strictly controlled by a probability distribution, and the randomness arises only from the fact that we can not predict the exact values we will get in each trial. Every time we execute the preceding code, we will get three different numbers, but if we iterate the code thousands of times, we will be able to empirically check that the mean of the sampled values is around zero and that 95% of the samples will have values within the $[-1.96, +1.96]$ range. Please do not trust me, use your Pythonic powers and verify it for yourself. We can arrive at this same conclusion if we study the mathematical properties of the Normal distribution.

A common notation used in statistics to indicate that a variable is distributed as a normal distribution with parameters μ and σ is:

$$x \sim \mathcal{N}(\mu, \sigma) \tag{1.2}$$

In this context, when you find the \sim (tilde) symbol, say *is distributed as*.



In many texts, it is common to see the normal distribution expressed in terms of the variance and not the standard deviation. Thus, people write $\mathcal{N}(\mu, \sigma^2)$. In this book, we are going to parameterize the Normal distribution using the standard deviation, first because it is easier to interpret, and second because this is how PyMC3 works.

We will meet several probability distributions in this book; every time we discover one, we will take a moment to describe it. We started with the normal distribution because it is like the Zeus of the probability distributions. A variable, X , follows a Gaussian distribution if its values are dictated by the following expression:

$$p(x | \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (1.3)$$

This is the probability density function for the Normal distribution; you do not need to memorize expression 1.3, I just like to show it to you so you can see where the numbers come from. As we have already mentioned, μ and σ are the parameters of the distribution, and by specifying those values we totally define the distribution; you can see this from expression 1.3 as the other terms are all constants. μ can take any real value, that is, $\mu \in \mathbb{R}$, and dictates the mean of the distribution (and also the median and mode, which are all equal). σ is the standard deviation, which can only be positive and dictates the spread of the distribution, the larger the value of σ , the more spread out the distribution. Since there are an infinite number of possible combinations of μ and σ , there is an infinite number of *instances* of the Gaussian distribution and all of them belong to the same *Gaussian family*.

Mathematical formulas are concise and unambiguous and some people say even beautiful, but we must admit that meeting them for the first time can be intimidating, specially for those not very mathematically inclined; a good way to break the ice is to use Python to explore them:

```
mu_params = [-1, 0, 1]
sd_params = [0.5, 1, 1.5]
x = np.linspace(-7, 7, 200)
_, ax = plt.subplots(len(mu_params), len(sd_params), sharex=True,
                     sharey=True,
                     figsize=(9, 7), constrained_layout=True)
for i in range(3):
    for j in range(3):
        mu = mu_params[i]
        sd = sd_params[j]
```

```

y = stats.norm(mu, sd).pdf(x)
ax[i,j].plot(x, y)
ax[i,j].plot([], label=" $\mu = {:.3.2f}$ \n $\sigma = {:.3.2f}$ ".format(mu,
                                                               sd), alpha=0)
ax[i,j].legend(loc=1)
ax[2,1].set_xlabel('x')
ax[1,0].set_ylabel('p(x)', rotation=0, labelpad=20)
ax[1,0].set_yticks([])

```

Most of the preceding code is for plotting, the *probabilistic part* is performed by the `y = stats.norm(mu, sd).pdf(x)` line. With this line, we are evaluating the probability density function of the normal distribution, given the `mu` and `sd` parameters for a set of `x` values. The preceding code generates *Figure 1.1*. In each subplot we have a blue (dark gray) curve representing a Gaussian distribution with specific parameters, μ and σ , included in each subplot's legend:

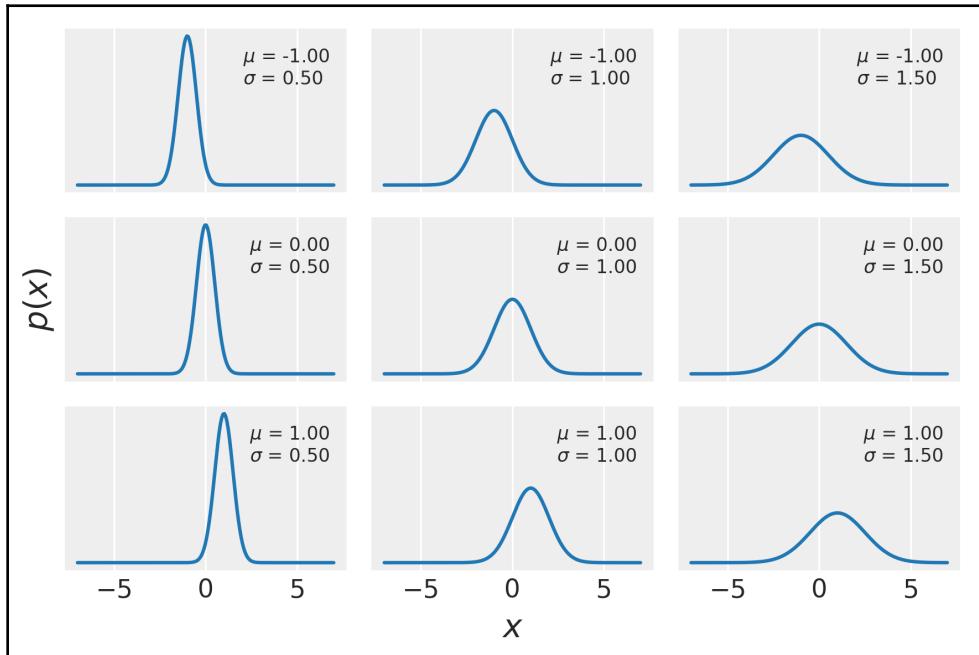


Figure 1.1



Most of the figures in this book are generated directly from the code preceding them, many times even without a leading phrase connecting the code to the figure. This pattern should be familiar to those of you working with Jupyter Notebooks or Jupyter Lab.

There are two types of random variables: continuous and discrete. **Continuous variables** can take any value from some interval (we can use Python floats to represent them), and **discrete variables** can take only certain values (we can use Python integers to represent them). The Normal distribution is a continuous distribution.

Notice that in the preceding plot, the *y*ticks are omitted; this is a feature not a bug. The reason to omit them is that, in my experience, those values do not add too much relevant information and could be potentially confusing to some people. Let me explain myself: the actual numbers on the *y* axis do not really matter—what is relevant is their relative values. If you take two values from x , let's say x_i and x_j , and you find that $p(x_i) = 2p(x_j)$ (two times higher in the plot), you can safely say the probability of the value of x_i is twice the probability of x_j . This is something that most people will understand intuitively, and fortunately is the correct interpretation. The only tricky part when dealing with continuous distributions is that the values plotted on the *y* axis are not probabilities, but **probability densities**. To get a proper probability, you have to integrate between a given interval, that is, you have to compute the area below the curve (for that interval). While probabilities cannot be greater than one, probability densities can, and is the total area under the probability density curve the one restricted to be 1. Understanding the difference between probabilities and probability densities is crucial from a mathematical point of view. For the practical approach used in this book, we can be a little bit more sloppy, since that difference is not *that* important as long as you get the correct intuition on how to interpret the previous plot in terms of relative values.

Independently and identically distributed variables

Many models assume that successive values of random variables are all sampled from the same distribution and those values are independent of each other. In such a case, we will say that the variables are **independently and identically distributed (iid)** variables for short. Using mathematical notation, we can see that two variables are independent if $p(x, y) = p(x)p(y)$ for every value of x and y .

A common example of non-iid variables are **temporal series**, where a temporal dependency in the random variable is a key feature that should be taken into account. Take, for example, the following data coming from <http://cdiac.esd.ornl.gov>. This data is a record of atmospheric CO₂ measurements from 1959 to 1997. We are going to load the data (including the accompanying code) and plot it:

```
data = np.genfromtxt('../data/mauna_loa_CO2.csv', delimiter=',')
plt.plot(data[:,0], data[:,1])
plt.xlabel('year')
plt.ylabel('$CO_2$ (ppmv)')
plt.savefig('B11197_01_02.png', dpi=300)
```

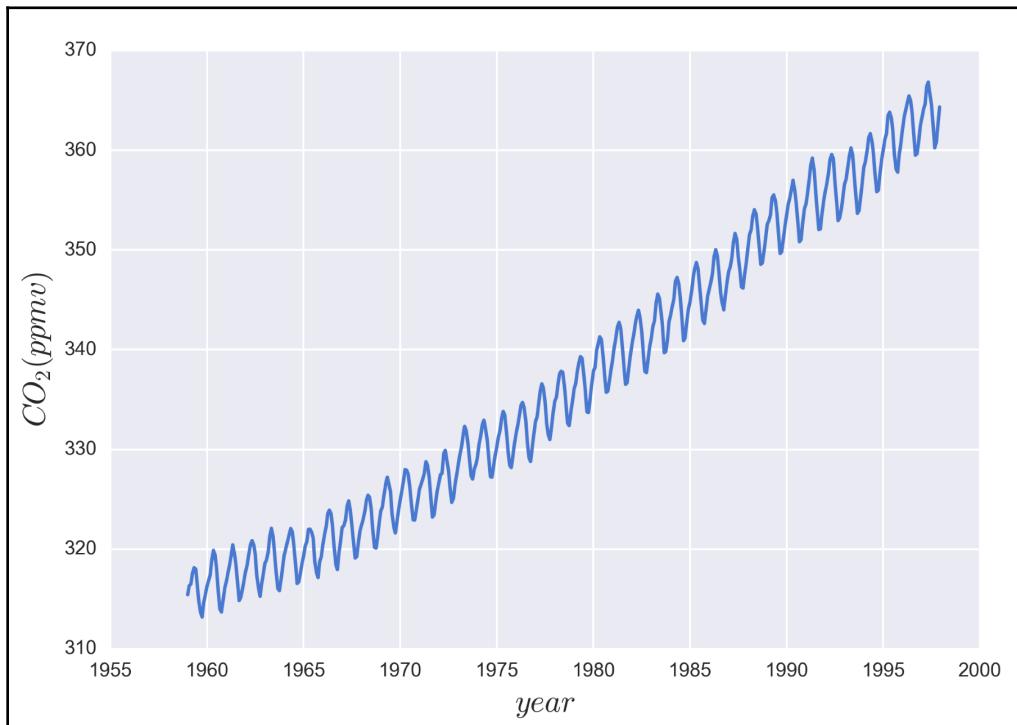


Figure 1.2

Each data point corresponds to the measured levels of atmospheric CO₂ per month. The temporal dependency of data points is easy to see in this plot. In fact, we have two trends here: a seasonal one (this is related to cycles of vegetation growth and decay), and a global one, indicating an increasing concentration of atmospheric CO₂.

Bayes' theorem

Now that we have learned some of the basic concepts and jargon from probability theory, we can move to the moment everyone was waiting for. Without further ado, let's contemplate, in all its majesty, Bayes' theorem:

$$p(\theta | y) = \frac{p(y | \theta)p(\theta)}{p(y)} \quad (1.4)$$

Well, it's not that impressive, is it? It looks like an elementary school formula, and yet, paraphrasing Richard Feynman, *this is all you need to know about Bayesian statistics.*

Learning where Bayes' theorem comes from will help us to understand its meaning.

According to the product rule we have:

$$p(\theta, y) = p(\theta | y)p(y) \quad (1.5)$$

This can also be written as:

$$p(\theta, y) = p(y | \theta)p(\theta) \quad (1.6)$$

Given that the terms on the left are equal for equations 1.5 and 1.6, we can combine them and write:

$$p(\theta | y)p(y) = p(y | \theta)p(\theta) \quad (1.7)$$

And if we reorder 1.7, we get expression 1.4, thus is Bayes' theorem.

Now, let's see what formula 1.4 implies and why it is important. First, it says that $p(\theta | y)$ is not necessarily the same as $p(y | \theta)$. This is a very important fact, one that is easy to miss in daily situations even for people trained in statistics and probability. Let's use a simple example to clarify why these quantities are not necessarily the same. The probability of a person being the Pope given that this person is Argentinian is not the same as the probability of being Argentinian given that this person is the Pope. As there are around 44,000,000 Argentinians alive and a single one of them is the current Pope, we have $p(\text{Pope} | \text{Argentinian}) \cong 1/44000000$ and we also have $p(\text{Argentinian} | \text{Pope}) = 1$.

If we replace θ with *hypothesis* and y with *data*, Bayes' theorem tells us how to compute the probability of a hypothesis, θ , given the data, y , and that's the way you will find Bayes' theorem explained in a lot of places. But, how do we turn a hypothesis into something that we can put inside Bayes' theorem? Well, we do it by using probability distributions. So, in general, our hypothesis is a *hypothesis* in a very, very, very narrow sense; we will be more precise if we talk about finding a suitable value for parameters in our models, that is, parameters of probability distributions. By the way, don't try to set θ to statements such as *unicorns are real*, unless you are willing to build a realistic probabilistic model of unicorn existence!

Bayes' theorem is central to Bayesian statistics, as we will see in Chapter 2, *Programming Probabilistically* using tool such as PyMC3 free ourselves of the need to explicitly write Bayes' theorem every time we build a Bayesian model. Nevertheless, it is important to know the name of its *parts* because we will constantly refer to them and it is important to understand what each *part* means because this will help us to conceptualize models, so let's do it:

- $p(\theta)$: Prior
- $p(y | \theta)$: Likelihood
- $p(\theta | y)$: Posterior
- $p(y)$: Marginal likelihood

The **prior distribution** should reflect what we know about the value of the θ parameter before seeing the data, y . If we know nothing, like Jon Snow, we could use flat priors that do not convey too much information. In general, we can do better than flat priors, as we will learn in this book. The use of priors is why some people still talk about Bayesian statistics as subjective, even when priors are just another assumption that we made when modeling and hence are just as subjective (or objective) as any other assumption, such as likelihoods.

The **likelihood** is how we will introduce data in our analysis. It is an expression of the plausibility of the data given the parameters. In some texts, you will find people call this term *sampling model*, *statistical model*, or just *model*. We will stick to the name likelihood and we will call the combination of priors and likelihood model.

The **posterior distribution** is the result of the Bayesian analysis and reflects all that we know about a problem (given our data and model). The posterior is a probability distribution for the θ parameters in our model and not a single value. This distribution is a balance between the prior and the likelihood. There is a well-known joke: *A Bayesian is one who, vaguely expecting a horse, and catching a glimpse of a donkey, strongly believes he has seen a mule.* One excellent way to kill the mood after hearing this joke is to explain that if the likelihood and priors are both vague, you will get a posterior reflecting *vague beliefs* about seeing a mule rather than strong ones. Anyway, I like the joke, and I like how it captures the idea of a posterior being somehow a compromise between prior and likelihood.

Conceptually, we can think of the posterior as the updated prior in the light of (new) data. In fact, the posterior from one analysis can be used as the prior for a new analysis. This makes Bayesian analysis particularly suitable for analyzing data that becomes available in sequential order. Some examples could be an early warning system for natural disasters that processes *online* data coming from meteorological stations and satellites. For more details, read about online machine learning methods.

The last term is the **marginal likelihood**, also known as **evidence**. Formally, the marginal likelihood is the probability of observing the data averaged over all the possible values the parameters can take (as prescribed by the prior). Anyway, for most of this book, we will not care about the marginal likelihood, and we will think of it as a simple normalization factor. We can do this because when analyzing the posterior distribution, we will only care about the relative values of the parameters and not their absolute values. You may remember that we mentioned this when we talked about how to interpret plots of probability distributions in the previous section. If we ignore the marginal likelihood, we can write Bayes' theorem as a proportionality:

$$p(\theta | y) \propto p(y | \theta)p(\theta) \quad (1.8)$$

Understanding the exact role of each term in Bayes' theorem will take some time and practice, and it will require we work through a few examples, and that's what the rest of this book is for.

Single-parameter inference

In the last two sections, we learned several important concepts, but two of them are essentially the core of Bayesian statistics, so let's restate them in a single sentence.



Probabilities are used to measure the uncertainty we have about parameters, and Bayes' theorem is the mechanism to correctly update those probabilities in light of new data, hopefully reducing our uncertainty.

Now that we know what Bayesian statistics is, let's learn how to do Bayesian statistics with a simple example. We are going to begin inferring a single, unknown parameter.

The coin-flipping problem

The coin-flipping problem, or the beta-binomial model if you want to sound fancy at parties, is a classical problem in statistics and goes like this: we toss a coin a number of times and record how many heads and tails we get. Based on this data, we try to answer questions such as, *is the coin fair?* Or, more generally, *how biased is the coin?* While this problem may sound dull, we should not underestimate it. The coin-flipping problem is a great example to learn the basics of Bayesian statistics because it is a simple model that we can solve and compute with ease. Besides, many real problems consist of binary, mutually-exclusive outcomes such as 0 or 1, positive or negative, odds or evens, spam or ham, hotdog or not hotdog, cat or dog, safe or unsafe, and healthy or unhealthy. Thus, even when we are talking about coins, this model applies to any of those problems.

In order to estimate the bias of a coin, and in general to answer any questions in a Bayesian setting, we will need data and a probabilistic model. For this example, we will assume that we have already tossed a coin a number of times and we have a record of the number of observed heads, so the data-gathering part is already done. Getting the model will take a little bit more effort. Since this is our first model, we will explicitly write Bayes' theorem and do all the necessary math (don't be afraid, I promise it will be painless) and we will proceed very slowly. From Chapter 2, *Programming Probabilistically*, onward, we will use PyMC3 and our computer to do the math for us.

The general model

The first thing we will do is generalize the concept of bias. We will say that a coin with a bias of 1 will always land heads, one with a bias of 0 will always land tails, and one with a bias of 0.5 will land half of the time heads and half of the time tails. To represent the bias, we will use the θ parameter, and to represent the total number of heads for a N number of tosses, we will use the y variable. According to Bayes' theorem (equation 1.4), we have to specify the prior, $p(\theta)$, and likelihood, $p(y | \theta)$, we will use. Let's start with the likelihood.

Choosing the likelihood

Let's assume that only two outcomes are possible—heads or tails—and let's also assume that a coin toss does not affect other tosses, that is, we are assuming coin tosses are independent of each other. We will further assume all coin tosses come from the same distribution. Thus the random variable *coin toss* is an example of an **iid** variable. I hope you agree these are very reasonable assumptions to make for our problem. Given these assumptions a good candidate for the likelihood is the binomial distribution:

$$p(y | \theta, N) = \frac{N!}{y!(N-y)!} \theta^y (1-\theta)^{N-y} \quad (1.9)$$

This is a discrete distribution returning the probability of getting y heads (or in general, successes) out of N coin tosses (or in general, trials or experiments) given a fixed value of θ :

```
n_params = [1, 2, 4] # Number of trials
p_params = [0.25, 0.5, 0.75] # Probability of success

x = np.arange(0, max(n_params)+1)
f,ax = plt.subplots(len(n_params), len(p_params), sharex=True,
                    sharey=True,
                    figsize=(8, 7), constrained_layout=True)

for i in range(len(n_params)):
    for j in range(len(p_params)):
        n = n_params[i]
        p = p_params[j]

        y = stats.binom(n=n, p=p).pmf(x)

        ax[i,j].vlines(x, 0, y, colors='C0', lw=5)
        ax[i,j].set_ylim(0, 1)
        ax[i,j].plot(0, 0, label="N = {:.2f}\nθ = {:.2f}".format(n,p), alpha=0)
        ax[i,j].legend()
```

```
ax[2,1].set_xlabel('y')
ax[1,0].set_ylabel('p(y | θ, N)')
ax[0,0].set_xticks(x)
```

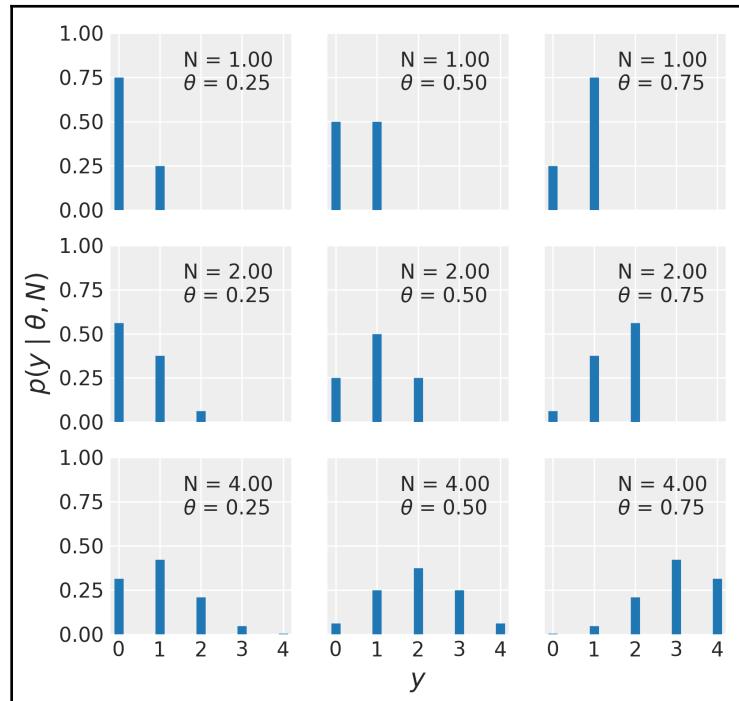


Figure 1.3

The preceding figure shows nine binomial distributions; each subplot has its own legend indicating the values of the parameters. Notice that for this plot I did not omit the values on the y axis. I did this so you can check for yourself that if you sum the height of all bars you will get 1, that is, for discrete distributions, the height of the bars represents actual probabilities.

The binomial distribution is a reasonable choice for the likelihood. We can see that θ indicates how likely it is to obtain a head when tossing a coin (this is easier to see when $N = 1$, but is valid for any value of N)—just compare the value of θ with the height of the bar for $y = 1$ (heads).

OK, if we know the value of θ , the binomial distribution will tell us the expected distribution of heads. The only problem is that we do not know θ ! But do not despair; in Bayesian statistics, every time we do not know the value of a parameter, we put a prior on it, so let's move on and choose a prior for θ .

Choosing the prior

As a prior, we will use a **beta distribution**, which is a very common distribution in Bayesian statistics and looks as follows:

$$p(\theta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^{\alpha-1} (1 - \theta)^{\beta-1} \quad (1.10)$$

If we look carefully, we will see that the beta distribution looks similar to the binomial except for the first term, the one with all those Γ . The first term is a normalizing constant that ensures the distribution integrates to 1, and Γ is the Greek uppercase gamma letter and represents what is known as **gamma function**. We can see from the preceding formula that the beta distribution has two parameters, α and β . Using the following code, we will explore our third distribution so far:

```
params = [0.5, 1, 2, 3]
x = np.linspace(0, 1, 100)
f, ax = plt.subplots(len(params), len(params), sharex=True,
                     sharey=True,
                     figsize=(8, 7), constrained_layout=True)
for i in range(4):
    for j in range(4):
        a = params[i]
        b = params[j]
        y = stats.beta(a, b).pdf(x)
        ax[i, j].plot(x, y)
```

```
ax[i,j].plot(0, 0, label="α = {:2.1f}\nβ = {:2.1f}".format(a,
    b), alpha=0)
ax[i,j].legend()
ax[1,0].set_yticks([])
ax[1,0].set_xticks([0, 0.5, 1])
f.text(0.5, 0.05, 'θ', ha='center')
f.text(0.07, 0.5, 'p(θ)', va='center', rotation=0)
```

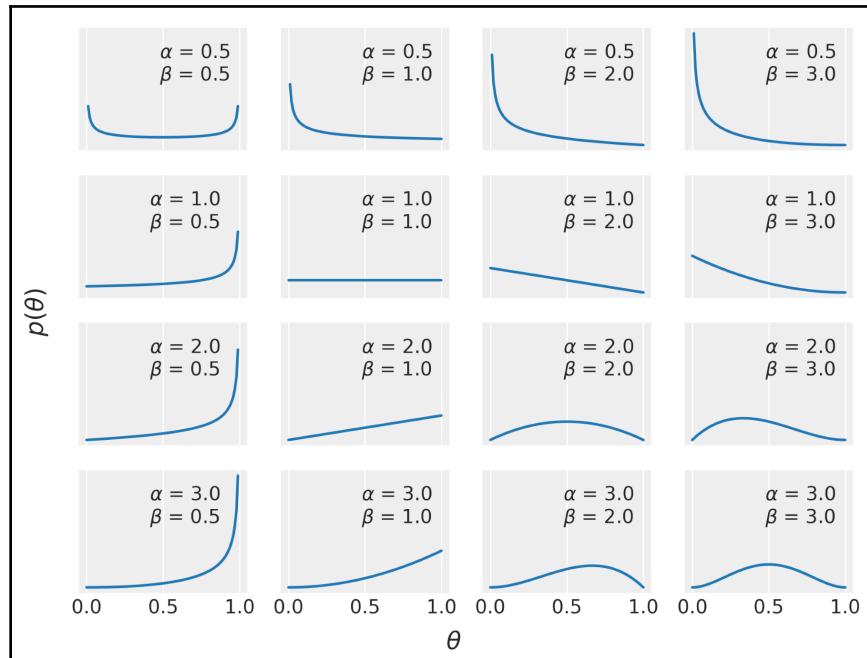


Figure 1.4

I really like the beta distribution and all the *shapes* we can get from it, but why are we using it for our model? There are many reasons to use a beta distribution for this and other problems. One of them is that the beta distribution is restricted to be between 0 and 1, in the same way our θ parameter is. In general, we use the beta distribution when we want to model proportions of a binomial variable. Another reason is for its versatility. As we can see in the preceding figure, the distribution adopts several shapes (all restricted to the $[0, 1]$ interval), including a uniform distribution, Gaussian-like distributions, and U-like distributions. As a third reason, the beta distribution is the conjugate prior of the binomial distribution (which we are using as the likelihood). A conjugate prior of a likelihood is a prior that, when used in combination with a given likelihood, returns a posterior with the same functional form as the prior. Untwisting the tongue, every time we use a beta distribution as the prior and a binomial distribution as the likelihood, we will get a beta as the posterior distribution. There are other pairs of conjugate priors; for example, the Normal distribution is the conjugate prior of itself. For many years, Bayesian analysis was restricted to the use of conjugate priors. Conjugacy ensures mathematical tractability of the posterior, which is important given that a common problem in Bayesian statistics is ending up with a posterior we cannot solve analytically. This was a deal breaker before the development of suitable computational methods to solve probabilistic methods. From Chapter 2, *Programming Probabilistically* onwards, we will learn how to use modern computational methods to solve Bayesian problems, whether we choose conjugate priors or not.

Getting the posterior

Let's remember that Bayes' theorem (*equation 1.4*) says the posterior is proportional to the likelihood times the prior. So, for our problem, we have to multiply the binomial and the beta distributions:

$$p(\theta | y) \propto \frac{N!}{y!(N-y)!} \theta^y (1-\theta)^{N-y} \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^{\alpha-1} (1-\theta)^{\beta-1} \quad (1.11)$$

We can simplify this expression. For our practical concerns, we can drop all the terms that do not depend on θ and our results will still be valid. Accordingly, we can write:

$$p(\theta | y) \propto \theta^y (1-\theta)^{N-y} \theta^{\alpha-1} (1-\theta)^{\beta-1} \quad (1.12)$$

Reordering it, we get:

$$p(\theta | y) \propto \theta^{y+\alpha-1} (1-\theta)^{N-y+\beta-1} \quad (1.13)$$

If we pay attention, we will see that this expression has the same functional form of a beta distribution (except for the normalization term) with $\alpha_{posterior} = \alpha_{prior} + y$ and $\beta_{posterior} = \beta_{prior} + N - y$. In fact, the posterior distribution for our problem is the beta distribution:

$$p(\theta | y) \propto \text{Beta}(\alpha_{prior} + y, \beta_{prior} + N - y) \quad (1.14)$$

Computing and plotting the posterior

Now we will use Python to compute and plot the posterior distribution based on the analytical expression we have just derived. In the following code, you will see there is actually one line that computes the results while the others are there just to get a nice plot:

```
plt.figure(figsize=(10, 8))

n_trials = [0, 1, 2, 3, 4, 8, 16, 32, 50, 150]
data = [0, 1, 1, 1, 1, 4, 6, 9, 13, 48]
theta_real = 0.35

beta_params = [(1, 1), (20, 20), (1, 4)]
dist = stats.beta
x = np.linspace(0, 1, 200)

for idx, N in enumerate(n_trials):
    if idx == 0:
        plt.subplot(4, 3, 2)
        plt.xlabel('θ')
    else:
        plt.subplot(4, 3, idx+3)
        plt.xticks([])
    y = data[idx]
    for (a_prior, b_prior) in beta_params:
        p_theta_given_y = dist.pdf(x, a_prior + y, b_prior + N - y)
        plt.fill_between(x, 0, p_theta_given_y, alpha=0.7)

plt.axvline(theta_real, ymax=0.3, color='k')
plt.plot(0, 0, label=f'{N:4d} trials\n{y:4d} heads', alpha=0)
plt.xlim(0, 1)
plt.ylim(0, 12)
plt.legend()
```

```
plt.yticks([])
plt.tight_layout()
```

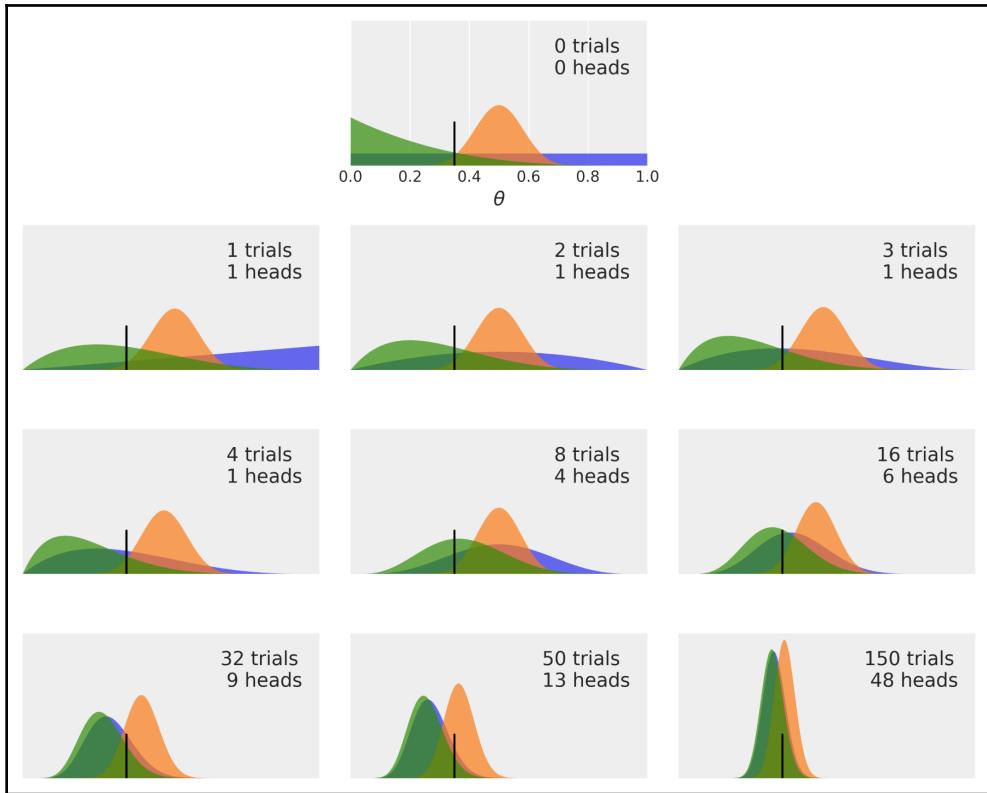


Figure 1.5

On the first subplot of *Figure 1.5*, we have zero trials, thus the three curves represent our priors:

- The uniform (blue) prior. This represent all the possible values for the bias being equally probable a priori.
- The Gaussian-like (orange) prior is centered and concentrated around 0.5, so this prior is compatible with information indicating that the coin has more or less about the same chance of landing heads or tails. We could also say this prior is compatible with the *belief* that most coins are fair. While *belief* is commonly used in Bayesian discussions, we think is better to talk about models and parameters that are informed by data.
- The skewed (green) prior puts the most weight on a tail-biased outcome.

The rest of the subplots show posterior distributions for successive trials. The number of trials (or coin tosses) and the number of heads are indicated in each subplot's legend. There is also a black vertical line at 0.35 representing the true value for θ . Of course, in real problems, we do not know this value, and it is here just for pedagogical reasons. This figure can teach us a lot about Bayesian analysis, so grab your coffee, tea, or favorite drink and let's take a moment to understand it:

- The result of a Bayesian analysis is a posterior distribution, not a single value but a distribution of plausible values given the data and our model.
- The most probable value is given by the mode of the posterior (the peak of the distribution).
- The spread of the posterior is proportional to the uncertainty about the value of a parameter; the more spread out the distribution, the less certain we are.
- Intuitively, we are more confident in a result when we have observed more data supporting that result. Thus, even when numerically $\frac{1}{2} = \frac{4}{8} = 0.5$, seeing four heads out of eight trials gives us more confidence that the bias is 0.5 than observing one head out of two trials. This intuition is reflected in the posterior, as you can check for yourself if you pay attention to the (blue) posterior in the third and sixth subplots; while the mode is the same, the spread (uncertainty) is larger in the third subplot than in the sixth subplot.
- Given a sufficiently large amount of data, two or more Bayesian models with different priors will tend to converge to the same result. In the limit of infinite data, no matter which prior we use, all of them will provide the same posterior. Remember that infinite is a limit and not a number, so from a practical point of view, we could get *practically* indistinguishably posteriors for a finite and rather small number of data points.
- How fast posteriors converge to the same distribution depends on the data and the model. In the preceding figure, we can see that the posteriors coming from the blue prior (uniform) and green prior (biased towards tails) converge faster to *almost the same* distribution, while it takes longer for the orange posterior (the one coming from the concentrated prior). In fact, even after 150 trials, it is somehow easy to recognize the orange posterior as a different distribution from the two others.
- Something not obvious from the figure is that we will get the same result if we update the posterior sequentially than if we do it all at once. We can compute the posterior 150 times, each time adding one more observation and using the obtained posterior as the new prior, or we can just compute one posterior for the 150 tosses at once. The result will be exactly the same. This feature not only makes perfect sense, it also leads to a natural way of updating our estimations when we get new data, a situation common in many data-analysis problems.

The influence of the prior and how to choose one

From the preceding example, it is clear that priors can influence inferences. This is totally fine, priors are supposed to do this. Newcomers to Bayesian analysis (as well as detractors of this paradigm) are generally a little nervous about how to choose priors, because they do not want the prior to act as a censor that does not let the data speak for itself! That's OK, but we have to remember that data does not really speak; at best, data murmurs. Data only makes sense in the context of our models, including mathematical and mental models. There are plenty of examples in the history of science where the same data leads people to think differently about the same topics, and this can happen even if you base your opinions on formal models.

Some people like the idea of using non-informative priors (also known as *flat, vague, or diffuse priors*); these priors have the least possible amount of impact on the analysis. While it is possible to use them, in general, we can do better. Throughout this book, we will follow the recommendations of Gelman, McElreath, Kruschke, and many others, and we will prefer **weakly-informative priors**. For many problems, we often know something about the values a parameter can take, we may know that a parameter is restricted to being positive, or we may know the approximate range it can take, or whether we expect the value to be close to zero or below/above some value. In such cases, we can use priors to put some weak information in our models without being afraid of being too pushy. Because these priors work to keep the posterior distribution within certain reasonable bounds, they are also known as **regularizing priors**. Using informative priors is also a valid option if we have good-quality information to define those priors. Informative priors are very strong priors that convey a lot of information. Depending on your problem, it could be easy or not to find this type of prior. For example, in my field of work (structural bioinformatics), people have been using, in Bayesian and non-Bayesian ways, all the prior information they could get to study and especially predict the structure of proteins. This is reasonable because we have been collecting data from thousands of carefully-designed experiments for decades and hence we have a great amount of trustworthy prior information at our disposal. Not using it would be absurd! So, the take-home message is this: if you have reliable prior information, there is no reason to discard that information, including the nonsensical argument that being objective means throwing away valuable information. Imagine if every time an automotive engineer had to design a new car, they had to start from scratch and reinvent the combustion engine, the wheel, and for that matter, the whole concept of a car. That's not the way things should work.

Knowing we can classify priors into categories according to their relative strength does not make us less nervous about choosing from them. Maybe it would be better to not have priors at all—that would make modeling easier, right? Well, not necessarily. Priors can make models behave better, have better generalization properties, and can help us convey useful information. Also, every model, Bayesian or not, has some kind of prior in one way or another, even if the prior is not set explicitly. In fact, many result from *frequentist statistics*, and can be seen as special cases of a Bayesian model under certain circumstances, such as flat priors. One common *frequentist* method to estimate parameters is known as maximum likelihood; this methods avoids setting a prior and works just by finding the value of θ that maximizes the likelihood. This value is usually notated by adding a little *hat* on top of the symbol of the parameter we are estimating, such as $\hat{\theta}$ or sometimes θ_{mle} (or even both). $\hat{\theta}$ is a point estimate (a number) and not a distribution. For the coin-flipping problem we can compute this analytically:

$$\hat{\theta} = \frac{y}{N} \quad (1.15)$$

If you go back to *Figure 1.5*, you will be able to check for yourself that the mode of the blue posterior (the one corresponding to the uniform/flat prior) agrees with the values of $\hat{\theta}$, computed for each subplot. So, at least for this example, we can see that even when the maximum likelihood method does not explicitly invoke any prior, it can be considered a special case of a Bayesian model, one with a uniform prior.

We cannot really avoid priors, but if we include them in our analysis, we will get several benefits, including a distribution of plausible values and not only the most probable one. Another advantage of being explicit about priors is that we get more transparent models, meaning they're easier to criticize, debug (in a broad sense of the word), and hopefully improve. Building models is an iterative process; sometimes the iteration takes a few minutes, sometimes it could take years. Sometimes it will only involve you, and sometimes it will involve people you do not even know. Reproducibility matters and transparent assumptions in a model contribute to it. Besides, we are free to use more than one prior (or likelihood) for a given analysis if we are not sure about any special one; exploring the effect of different priors can also bring valuable information to the table. Part of the modeling process is about questioning assumptions, and priors (and likelihood) are just that. Different assumptions will lead to different models and probably different results. By using data and our domain knowledge of the problem, we will be able to compare models and, if necessary, decide on a winner. *Chapter 5, Model Comparison*, will be devoted to this issue. Since priors have a central role in Bayesian statistics, we will keep discussing them as we face new problems. So if you have doubts and feel a little bit confused about this discussion, just keep calm and don't worry, people have been confused for decades and the discussion is still going on.

Communicating a Bayesian analysis

Creating reports and communicating results is central to the practice of statistics and data science. In this section, we will briefly discuss some of the peculiarities of this task when working with Bayesian models. In future chapters, we will keep looking at examples about this important matter.

Model notation and visualization

If you want to communicate the results of an analysis, you should also communicate the model you used. A common notation to succinctly represent probabilistic models is:

$$\begin{aligned}\theta &\sim \text{Beta}(\alpha, \beta) \\ y &\sim \text{Bin}(n = 1, p = \theta)\end{aligned}\tag{1.16}$$

This is just the model we use for the coin-flip example. As you may remember, the \sim symbol indicates that the variable, on the left of it, is a random variable distributed according to the distribution on the right. In many contexts, this symbol is used to indicate that a variable takes *approximately* some value, but when talking about probabilistic models, we will read this symbol out loud saying *is distributed as*. Thus, we can say θ is distributed as a beta distribution with α and β parameters, and y is distributed as a binomial with $n = 1$ and $p = \theta$ parameters. The very same model can be represented graphically using Kruschke's diagrams:

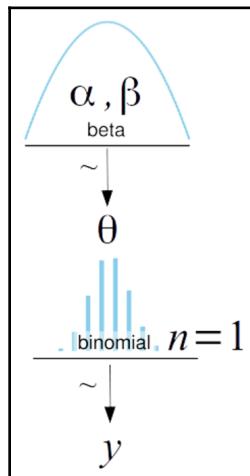


Figure 1.6

On the first level, we have the prior that generates the values for θ , then the likelihood, and on the last line the data, y . Arrows indicate the relationship between variables, and the \sim symbol indicates the stochastic nature of the variables. All Kruschke's diagrams in the book were made using the templates provided by Rasmus Bååth (<http://www.sumsar.net/blog/2013/10/diy-kruschke-style-diagrams/>).

Summarizing the posterior

The result of a Bayesian analysis is a posterior distribution, and all the information about the parameters given a dataset and a model is contained in the posterior distribution. Thus, by summarizing the posterior, we are summarizing the logical consequences of a model and data. A common practice is to report, for each parameter, the mean (or mode or median) to have an idea of the location of the distribution and some measure, such as the standard deviation, to have an idea of the dispersion and hence the uncertainty in our estimate. The standard deviation works well for normal-like distributions but can be misleading for other type of distributions, such as skewed ones. So, an alternative is to use the following measure.

Highest-posterior density

A commonly-used device to summarize the spread of a posterior distribution is to use a **Highest-Posterior Density (HPD)** interval. An HPD is the shortest interval containing a given portion of the probability density. One of the most commonly-used is the 95% HPD, often accompanied by the 50% HPD. If we say that the 95% HPD for some analysis is [2-5], we mean that according to our data and model, we think the parameter in question is between 2 and 5 with a probability of 0.95.



There is nothing special about choosing 95%, 50%, or any other value. They are just arbitrary commonly-used values; we are free to choose the 91.37% HPD interval if we like. If you want to use the 95% value, that's OK; just remember it is a default value. Ideally, justifications should be context-dependent and not automatic.

ArviZ is a Python package for exploratory data analysis for Bayesian models. ArviZ has many functions to help us summarize the posterior, for example, `az.plot_posterior` can be used to generate a plot with the mean and HPD of a distribution. In the following example, instead of a posterior from a real analysis, we are generating a random sample from a beta distribution:

```
np.random.seed(1)
az.plot_posterior({'θ':stats.beta.rvs(5, 11, size=1000)})
```

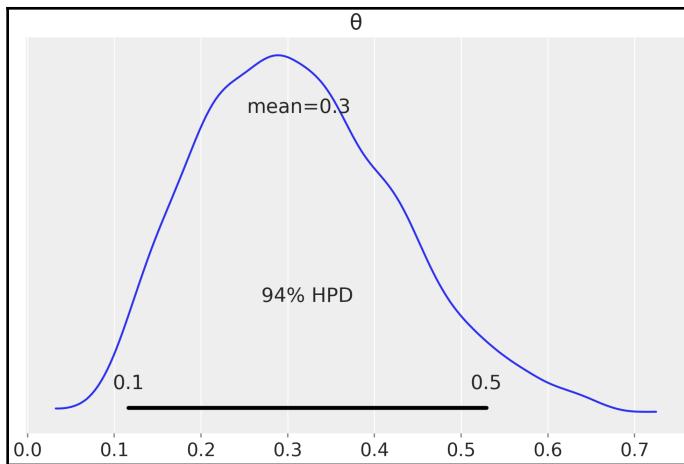


Figure 1.7

Note that in *Figure 1.7*, the reported HPD is 94%. This is a friendly remainder of the arbitrary nature of the 95% value. Every time ArviZ computes and reports a HPD, it will use, by default, a value of 0.94 (corresponding to 94%). You can change this by passing a different value to the `credible_interval` argument.



If you are familiar with the frequentist paradigm, please note that HPD intervals are not the same as confidence intervals. The HPD has a very intuitive interpretation, to the point that people often misinterpret frequentist confidence intervals as if they were Bayesian credible intervals. Performing a fully-Bayesian analysis enables us to talk about the probability of a parameter having some value. This is not possible in the frequentist framework since parameters are fixed by design; a frequentist confidence interval contains or does not contain the true value of a parameter.

Posterior predictive checks

One of the nice elements of the Bayesian toolkit is that once we have a posterior, it is possible to use the posterior, $p(\theta | y)$, to generate predictions, \hat{y} , based on the data, y , and the estimated parameters, θ . The posterior predictive distribution is:

$$p(\hat{y} | y) = \int p(\hat{y} | \theta)p(\theta | y)d\theta \quad (1.17)$$

Thus, the posterior predictive distribution is an average of conditional predictions over the posterior distribution of θ . Conceptually (and computationally), we approximate this integral 1.17 as an iterative two-step process:

1. We sample a value of θ from the posterior, $p(\theta | y)$
2. We feed that value of θ to the likelihood (or sampling distribution if you wish), thus obtaining a data point, \tilde{y}



Notice how this process combines two sources of uncertainty: the parameters uncertainty; as captured by the posterior; and the sampling uncertainty; as captured by the likelihood.

The generated predictions, \tilde{y} , can be used when we need to make, ahem, predictions. But also we can use them to criticize the models by comparing the observed data, y , and the predicted data, \tilde{y} , to spot differences between these two sets, this is known as **posterior predictive checks**. The main goal is to check for auto-consistency. The generated data and the observed data should look more or less similar, otherwise there was some problem during the modeling or some problem feeding the data to the model. But even in the absence of mistakes, differences could arise. Trying to understand the mismatch could lead us to improve models or at least to understand their limitations. Knowing which parts of our problem/data the model is capturing well and which it is not is valuable information even if we do not know how to improve the model. Maybe the model captures the mean behavior of our data well but fails to predict rare values. This could be problematic for us, or maybe we only care about the mean, so this model will be OK to us. The general aim is not to declare that a model is false. We just want to know which part of the model we can trust, and try to test whether the model is a good fit for our specific purpose. How confident one can be about a model is certainly not the same across disciplines. Physics can study systems under highly-controlled conditions using high-level theories, so models are often seen as good descriptions of reality. Other disciplines, such as sociology and biology, study complex, difficult-to-isolate systems, and thus models usually have a weaker epistemological status. Nevertheless, independent of which discipline you are working in, models should always be checked, and posterior predictive checks together with ideas from exploratory data analysis are a good way to check our models.

Summary

We began our Bayesian journey with a very brief discussion about statistical modeling, probability theory, and the introduction of Bayes' theorem. We then used the coin-flipping problem as an excuse to introduce basic aspects of Bayesian modeling and data analysis. We used this classic example to convey some of the most important ideas of Bayesian statistics, such as using probability distributions to build models and represent uncertainties. We tried to demystify the use of priors and put them on an equal footing with other elements that are part of the modeling process, such as the likelihood, or even more *meta-questions*, such as why we are trying to solve a particular problem in the first place. We ended the chapter by discussing the interpretation and communication of the results of a Bayesian analysis.

Figure 1.8 is based on one from Sumio Watanabe and summarizes the Bayesian workflow as described in this chapter:

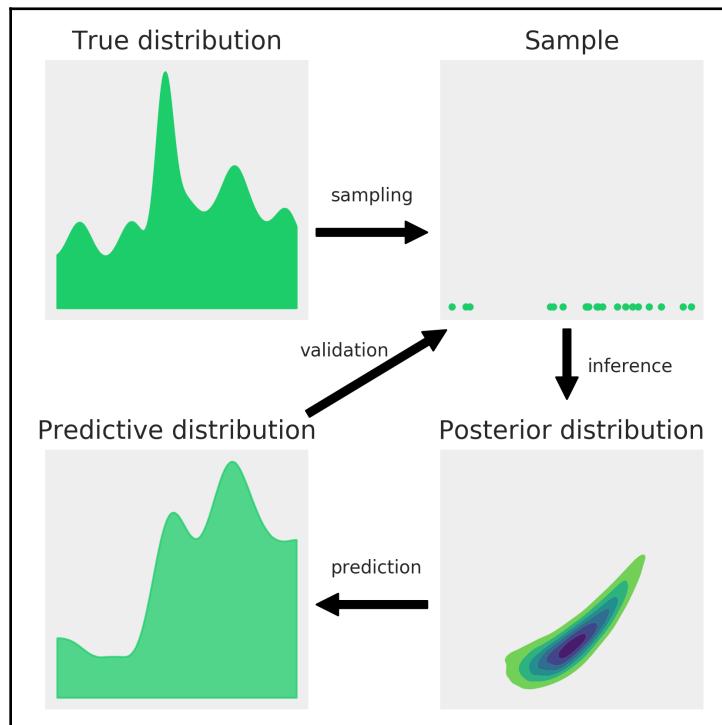


Figure 1.8

We assume there is a **True distribution** that in general is unknown (and in principle also unknowable), from which we get a finite **sample**, either by doing an experiment, a survey, an observation, or a simulation. In order to learn something from the True distribution, given that we have only observed a **sample**, we build a probabilistic model. A probabilistic model has two basic ingredients: a prior and a likelihood. Using the model and the sample, we perform Bayesian Inference and obtain a **Posterior distribution**; this distribution encapsulates all the information about a problem, given our model and data. From a Bayesian perspective, the posterior distribution is the main object of interest and everything else is derived from it, including predictions in the form of a **Posterior Predictive Distribution**.

Distribution. As the Posterior distribution (and any other derived quantity from it) is a consequence of the model and data, the usefulness of Bayesian inferences are restricted by the quality of models and data. One way to evaluate our model is by comparing the Posterior Predictive Distribution with the finite sample we got in the first place. Notice that the Posterior distribution is a distribution of the parameters in a model (conditioned on the observed samples), while the Posterior Predictive Distribution is a distribution of the predicted samples (averaged over the posterior distribution). The process of model validation is of crucial importance not because we want to be sure we have *the right model*, but because we know we almost never have *the right model*. We check models to evaluate whether they are *useful enough* in a specific context and, if not, to gain insight into how to improve them.

In this chapter, we have briefly summarized the main aspects of doing Bayesian data analysis. Throughout the rest of this book, we will revisit these ideas to really absorb them and use them as the scaffold of more advanced concepts. In the next chapter, we will introduce PyMC3, which is a Python library for Bayesian modeling and Probabilistic Machine Learning, and ArviZ, which is a Python library for the exploratory analysis of Bayesian models.

Exercises

We do not know whether the brain really works in a Bayesian way, in an approximate Bayesian fashion, or maybe some evolutionary (more or less) optimized heuristics. Nevertheless, we know that we learn by exposing ourselves to data, examples, and exercises... well you may say that humans never learn, given our record as a species on subjects such as wars or economic systems that prioritize profit and not people's well-being... Anyway, I recommend you do the proposed exercises at the end of each chapter:

1. From the following expressions, which one corresponds to the sentence, *The probability of being sunny given that it is 9th of July of 1816?*
 - $p(\text{sunny})$
 - $p(\text{sunny} \mid \text{July})$
 - $p(\text{sunny} \mid 9^{\text{th}} \text{ of July of 1816})$
 - $p(9^{\text{th}} \text{ of July of 1816} \mid \text{sunny})$
 - $p(\text{sunny}, 9^{\text{th}} \text{ of July of 1816}) / p(9^{\text{th}} \text{ of July of 1816})$
2. Show that the probability of choosing a human at random and picking the Pope is not the same as the probability of the Pope being human. In the animated series Futurama, the (Space) Pope is a reptile. How does this change your previous calculations?
3. In the following definition of a probabilistic model, identify the prior and the likelihood:

$$\begin{aligned} y_i &\sim \text{Normal}(\mu, \sigma) \\ \mu &\sim \text{Normal}(0, 10) \\ \sigma &\sim \text{HalfNormal}(25) \end{aligned}$$
4. In the previous model, how many parameters will the posterior have? Compare it with the model for the coin-flipping problem.
5. Write the Bayes' theorem for the model in exercise 3.
6. Let's suppose that we have two coins; when we toss the first coin, half of the time it lands tails and half of the time on heads. The other coin is a *loaded coin* that always lands on heads. If we take one of the coins at random and get a head, what is the probability that this coin is the unfair one?
7. Modify the code that generated *Figure 1.5* in order to add a dotted vertical line showing the observed rate head/(number of tosses), compare the location of this line to the mode of the posteriors in each subplot.
8. Try re-plotting *Figure 1.5* using other priors (`beta_params`) and other data (trials and data).

9. Explore different parameters for the Gaussian, binomial, and beta plots (*Figure 1.1*, *Figure 1.3*, and *Figure 1.4*, respectively). Alternatively, you may want to plot a single distribution instead of a grid of distributions.
10. Read about the Cromwel rule on Wikipedia:
https://en.wikipedia.org/wiki/Cromwell%27s_rule.
11. Read about probabilities and the Dutch book on Wikipedia:
https://en.wikipedia.org/wiki/Dutch_book.

2

Programming Probabilistically

"Our golems rarely have a physical form, but they too are often made of clay living in silicon as computer code."

- Richard McElreath

Now that we have a basic understanding of Bayesian statistics, we are going to learn how to build probabilistic models using computational tools. Specifically, we are going to learn about **probabilistic programming** with **PyMC3**. The basic idea is to specify models using code and then solve them in a more or less automatic way. It is not that we are too lazy to learn the mathematical way, nor are we elitist-hardcore-hackers-in-code. One important reason behind this choice is that many models do not lead to an analytic closed form, and thus we can only solve those models using numerical techniques.

Another reason to learn probabilistic programming is that modern Bayesian statistics is mainly done by writing code, and since we already know Python, why would we do it in another way? Probabilistic programming offers an effective way to build and solve complex models and allows us to focus more on model design, evaluation, and interpretation, and less on mathematical or computational details. In this chapter, and through the rest of this book, we are going to use PyMC3, a very flexible Python library for probabilistic programming, as well as **ArviZ**, a new Python library that will help us interpret the results of probabilistic models. Knowing PyMC3 and ArviZ will also help us to learn advanced Bayesian concepts in a more practical way.

In this chapter, we will cover the following topics:

- Probabilistic programming
- PyMC3 primer
- The coin-flipping problem revisited
- Summarizing the posterior

- The Gaussian and student's t models
- Comparing groups and the effect size
- Hierarchical models and shrinkage

Probabilistic programming

Bayesian statistics is conceptually very simple; we have *the knows* and *the unknowns*; we use Bayes' theorem to condition the latter on the former. If we are lucky, this process will reduce the uncertainty about the *unknowns*. Generally, we refer to the *knows* as **data** and treat it like a constant, and the *unknowns* as **parameters** and treat them as probability distributions. In more formal terms, we assign probability distributions to unknown quantities. Then, we use Bayes' theorem to transform the prior probability distribution $p(\theta)$ into a posterior distribution $p(\theta | y)$. Although conceptually simple, fully probabilistic models often lead to analytically intractable expressions. For many years, this was a real problem and was probably one of the main issues that hindered the wide adoption of Bayesian methods.

The arrival of the computational era and the development of numerical methods that, at least in principle, can be used to solve any inference problem, has dramatically transformed the Bayesian data analysis practice. We can think of these numerical methods as universal inference engines or as Thomas Wiecki, core developer of PyMC3, likes to call it, the inference-button. The possibility of automating the inference process has led to the development of **probabilistic programming languages (PPL)**, which allows for a clear separation between model creation and inference.

In the PPL framework, users specify a full probabilistic model by writing a few lines of code, and then inference follows automatically. It is expected that probabilistic programming will have a major impact on data science and other disciplines by enabling practitioners to build complex probabilistic models in a less time-consuming and less error-prone way. I think one good analogy for the impact that programming languages can have on scientific computing is the introduction of the Fortran programming language more than six decades ago. While nowadays Fortran has lost its shine, at one time, it was considered to be very revolutionary. For the first time, scientists moved away from computational details and began focusing on building numerical methods, models, and simulations in a more natural way. In a similar fashion, we now have PPL, which hides details on how probabilities are manipulated and how the inference is performed from users, allowing users to focus on model specification and the analysis of results.

In this chapter, we will learn how to use PyMC3 to define and solve models. We will treat the inference-button as a black box that gives us proper samples from the posterior distribution. The methods we will be using are stochastic, and so the samples will vary every time we run them. However, if the inference process works as expected, the samples will be representative of the posterior distribution and thus we will obtain the same conclusion from any of those samples. The details of what happens under the hood when we push the inference-button and how to check if the samples are indeed trustworthy will be explained in Chapter 8, *Inference Engines*.

PyMC3 primer

PyMC3 is a Python library for probabilistic programming. The last version at the moment of writing is 3.6. PyMC3 provides a very simple and intuitive syntax that is easy to read and that is close to the syntax used in the statistical literature to describe probabilistic models. PyMC3's base code is written using Python, and the computationally demanding parts are written using NumPy and Theano.

Theano is a Python library that was originally developed for deep learning and allows us to define, optimize, and evaluate mathematical expressions involving multidimensional arrays efficiently. The main reason PyMC3 uses Theano is because some of the sampling methods, such as NUTS, need gradients to be computed, and Theano knows how to compute gradients using what is known as **automatic differentiation**. Also, Theano compiles Python code to C code, and hence PyMC3 is really fast. This is all the information about Theano we need to have to use PyMC3. If you still want to learn more about it, start reading the official Theano tutorial

at <http://deeplearning.net/software/theano/tutorial/index.html#tutorial>.



You may have heard that Theano is no longer developed, but that's no reason to worry. PyMC devs will take over Theano maintenance, ensuring that Theano will keep serving PyMC3 for several years to come. At the same time, PyMC devs are moving quickly to create the successor to PyMC3. This will probably be based on TensorFlow as a backend, although other options are being analyzed as well. You can read more about this at the following blog post: https://medium.com/@pymc_devs/theano-tensorflow-and-the-future-of-pymc-6c9987bb19d5.

Flipping coins the PyMC3 way

Let's revisit the coin-flipping problem from Chapter 1, *Thinking Probabilistically*, but this time using PyMC3. We will use the same synthetic data we used in that chapter. Since we are generating the data, we know the true value of θ , called `theta_real`, in the following code. Of course, for a real dataset, we will not have this knowledge:

```
np.random.seed(123)
trials = 4
theta_real = 0.35 # unknown value in a real experiment
data = stats.bernoulli.rvs(p=theta_real, size=trials)
```

Model specification

Now that we have the data, we need to specify the model. Remember that this is done by specifying the likelihood and the prior using probability distributions. For the likelihood, we will use the binomial distribution with $n = 1$ and $p = \theta$, and for the prior, a beta distribution with the parameters $\alpha = \beta = 1$. A beta distribution with such parameters is equivalent to a uniform distribution in the interval $[0, 1]$. We can write the model using mathematical notation:

$$\begin{aligned}\theta &\sim \text{Beta}(\alpha, \beta) \\ y &\sim \text{Bern}(p = \theta)\end{aligned}\tag{2.1}$$

This statistical model has an almost one-to-one translation to PyMC3:

```
with pm.Model() as our_first_model:
    theta = pm.Beta('theta', alpha=1., beta=1.)
    y = pm.Bernoulli('y', p=theta, observed=data)
    trace = pm.sample(1000, random_seed=123)
```

The first line of the code creates a container for our model. Everything inside the `with-block` will be automatically added to `our_first_model`. You can think of this as syntactic sugar to ease model specification as we do not need to manually assign variables to the model. The second line specifies the prior. As you can see, the syntax follows the mathematical notation closely.



Please note that we use the name `theta` twice, first as a Python variable and then as the first argument of the `Beta` function; using the same name is a good practice to avoid confusion. The `theta` variable is a random variable; it is not a number, but an object representing a probability distribution from which we can compute random numbers and probability densities.

The third line specifies the likelihood. The syntax is almost the same as for the prior, except that we pass the data using the `observed` argument. This is the way in which we tell PyMC3 that we want to condition for the unknown on the knows (`data`). The observed values can be passed as a Python list, a tuple, a NumPy array, or a pandas DataFrame.

Now, we are finished with the model's specification! Pretty neat, right?

Pushing the inference button

The last line is the *inference button*. We are asking for 1,000 samples from the posterior and will store them in the `trace` object. Behind this innocent line, PyMC3 has hundreds of *oompa loompas* singing and baking a delicious Bayesian inference just for you! Well, not exactly, but PyMC3 is automating a lot of tasks. If you run the code, you will get a message like this:

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [θ]
100%|██████████| 3000/3000 [00:00<00:00, 3695.42it/s]
```

The first and second lines tell us that PyMC3 has automatically assigned the `NUTS` sampler (one inference engine that works very well for continuous variables), and has used a method to initialize that sampler. The third line says that PyMC3 will run two chains in parallel, thus we will get two independent samples from the posterior for the price of one.

The exact number of chains is computed taking into account the number of processors in your machine, you can change it using the `chains` argument for the `sample` function. The next line is telling us which variables are being sampled by which sampler. For this particular case, this line is not adding new information. Because `NUTS` is used to sample the only variable we have `θ`. However, this is not always the case as PyMC3 can assign different samplers to different variables. This is done automatically by PyMC3 based on properties of the variables that ensures that the best possible sampler is used for each variable. Users can manually assign samplers using the `step` argument of the `sample` function.

Finally, the last line is a progress bar, with several related metrics indicating how fast the sampler is working, including the number of iterations per second. If you run the code, you will see the progress-bar get updated really fast. Here, we are seeing the last stage when the sampler has finished its work. The numbers are 3000/3000, where the first number is the running sampler number (this starts at 1), and the last is the total number of samples. You will notice that we have asked for 1,000 samples, but PyMC3 is computing 3,000 samples. We have 500 samples per chain to auto-tune the sampling algorithm (NUTS, in this example). This sample will be discarded by default. We also have 1,000 productive draws per-chain, thus a total of 3,000 samples are generated. The tuning phase helps PyMC3 provide a reliable sample from the posterior. We can change the number of tuning steps with the `tune` argument of the `sample` function.

Summarizing the posterior

Generally, the first task we will perform after sampling from the posterior is check what the results look like. The `plot_trace` function from ArviZ is ideally suited to this task:

```
az.plot_trace(trace)
```

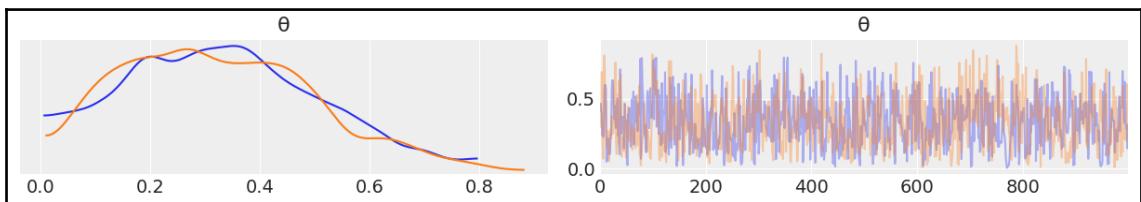


Figure 2.1

By using `az.plot_trace`, we get two subplots for each unobserved variable. The only unobserved variable in our model is θ . Notice that y is an observed variable representing the data; we do not need to sample that because we already know those values. Thus, in *Figure 2.1*, we have two subplots. On the left, we have a **Kernel Density Estimation (KDE)** plot; this is like the smooth version of the histogram. On the right, we get the individual sampled values at each step during the sampling. From the trace plot, we can visually get the plausible values from the posterior. You should compare this result using PyMC3 with those from the previous chapter, which were obtained analytically.

ArviZ provides several other plots to help interpret the trace, and we will see them in the following pages. We may also want to have a numerical summary of the trace. We can get that using `az.summary`, which will return a pandas DataFrame:

```
az.summary(trace)
```

	mean	sd	mc error	hpd 3%	hpd 97 %	eff_n	r_hat
θ	0.33	0.18	0.0	0.02	0.64	847.0	1.0

We get the mean, standard deviation (sd), and 94% HPD interval (hpd 3% and hpd 97%). As we discussed in Chapter 1, *Thinking Probabilistically*, we can use these numbers to interpret and report the results of a Bayesian inference. The last two metrics are related to diagnosing samples. For details, see Chapter 8, *Inference Engines*.

Another way to visually summarize the posterior is to use the `plot_posterior` function that comes with ArviZ. We have already used this distribution in the previous chapter for a fake posterior. We are going to use it now for a real posterior. By default, `plot_posterior` shows a histogram for discrete variables and KDEs for continuous variables. We also get the mean of the distribution (we can ask for the median or mode using the `point_estimate` argument) and the 94% HPD as a black line at the bottom of the plot. Different interval values can be set for the HPD with the `credible_interval` argument. This type of plot was introduced by John K. Kruschke in his great book *Doing Bayesian Data Analysis*:

```
az.plot_posterior(trace)
```

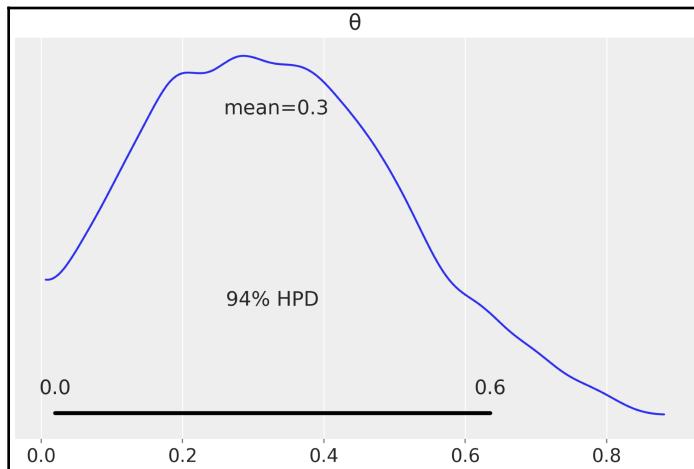


Figure 2.2

Posterior-based decisions

Sometimes, describing the posterior is not enough. Sometimes, we need to make decisions based on our inferences. We have to reduce a continuous estimation to a dichotomous one: yes-no, health-sick, contaminated-safe, and so on. We may need to decide if the coin is fair or not. A fair coin is one with a θ value of exactly 0.5. We can compare the value of 0.5 against the HPD interval. In *Figure 2.2*, we can see that the HPD goes from ≈ 0.02 to ≈ 0.71 and hence 0.5 is included in the HPD. According to our posterior, the coin seems to be tail-biased, but we cannot completely rule out the possibility that the coin is fair. If we want a sharper decision, we will need to collect more data to reduce the spread of the posterior or maybe we need to find out how to define a more informative prior.

ROPE

Strictly speaking, the chance of observing exactly 0.5 (that is, with infinite trailing zeros) is zero. Also, in practice, we generally do not care about exact results, but results within a certain margin. Accordingly, in practice, we can relax the definition of fairness and we can say that a fair coin is one with a value of θ around 0.5. For example, we could say that any value in the interval [0.45, 0.55] will be, for our purposes, practically equivalent to 0.5. We call this interval a **Region Of Practical Equivalence (ROPE)**. Once the ROPE is defined, we compare it against the **Highest-Posterior Density (HPD)**. We can get at least three scenarios:

- The ROPE does not overlap with the HPD; we can say the coin is not fair
- The ROPE contains the entire HPD; we can say the coin is fair
- The ROPE partially overlaps with HPD; we cannot say the coin is fair or unfair

If we choose a ROPE in the interval [0, 1], we will always say we have a fair coin. Notice that we do not need to collect data to perform any type of inference. Of course, this is a trivial, unreasonable, and dishonest choice and probably nobody is going to agree with our ROPE definition. I am just mentioning it to highlight the fact that the definition of the ROPE is context-dependent; there is no auto-magic rule that will fit everyone's intentions.

Decisions are inherently subjective and our mission is to take the most informed possible decisions according to our goals.



A ROPE is an arbitrary interval we choose based on background knowledge. Any value inside this interval is assumed to be of practical equivalence.

We can use the `plot_posterior` function to plot the posterior with the HPD interval and the ROPE. The ROPE appears as a semi-transparent thick (green) line:

```
az.plot_posterior(trace, rope=[0.45, .55])
```

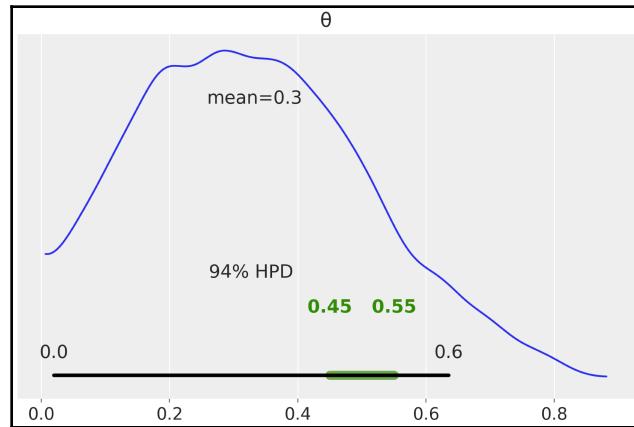


Figure 2.3

Another tool we can use to help us make a decision is to compare the posterior against a reference value. We can do this using `plot_posterior`. As you can see, we get a vertical (orange) line and the proportion of the posterior above and below our reference value:

```
az.plot_posterior(trace, ref_val=0.5)
```

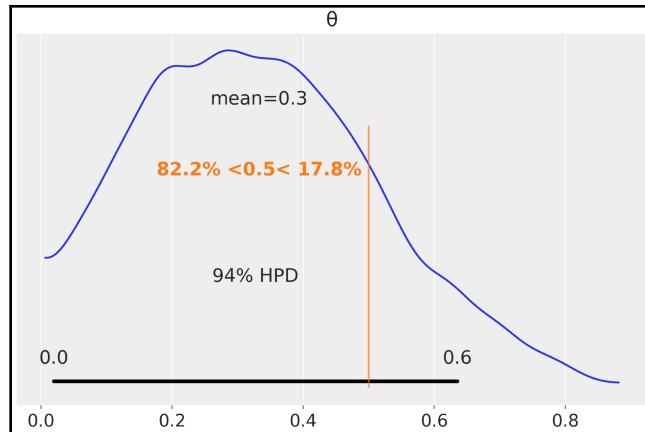


Figure 2.4

For a more detailed discussion on the use of the ROPE you could read *Chapter 12* of the great book *Doing Bayesian Data Analysis* by John Kruschke. This chapter also discusses how to perform hypothesis testing in a Bayesian framework and the caveats of hypothesis testing, whether in a Bayesian or non-Bayesian setting.

Loss functions

If you think these ROPE rules sound a little bit clunky and you want something more formal, **loss functions** are what you are looking for! To make a good decision, it is important to have the highest possible level of precision for the estimated value of the relevant parameters, but it is also important to take into account the cost of making a mistake. The cost/benefit trade-off can be mathematically formalized using loss functions. The names for loss functions or its inverses vary across different fields, and we could find names such as cost functions, objective functions, fitness functions, utility functions, and so on. No matter the name, the key idea is to use a function that captures how different the true value and the estimated value of a parameter are. The larger the value of the loss function, the worse the estimation is (according to the loss function). Some common examples of loss functions are:

- The quadratic loss $(\theta - \hat{\theta})^2$
- The absolute loss $|\theta - \hat{\theta}|$
- The 0-1 loss $I(\theta \neq \hat{\theta})$, where I is the indicator function

In practice, we generally do not have the value of the true parameter θ at hand. Instead, we have an estimation in the form of a posterior distribution. Thus, what we can do is find out the value of $\hat{\theta}$ that minimizes the **expected loss function**. By expected loss function, we mean the loss function averaged over the whole posterior distribution. In the following block of code, we have two loss functions: the absolute loss (`lossf_a`) and the quadratic loss (`lossf_b`). We will explore the value of $\hat{\theta}$ over a grid of 200 points. We will then plot those curves and we will also include the value of $\hat{\theta}$ that minimizes each loss function:

```
grid = np.linspace(0, 1, 200)
θ_pos = trace['θ']
lossf_a = [np.mean(abs(i - θ_pos)) for i in grid]
lossf_b = [np.mean((i - θ_pos)**2) for i in grid]

for lossf, c in zip([lossf_a, lossf_b], ['C0', 'C1']):
    mini = np.argmin(lossf)
    plt.plot(grid, lossf, c)
    plt.plot(grid[mini], lossf[mini], 'o', color=c)
    plt.annotate(' {:.2f}'.format(grid[mini]),
                (grid[mini], lossf[mini] + 0.03), color=c)
```

```
plt.yticks([])
plt.xlabel(r'$\hat{\theta}$')
```

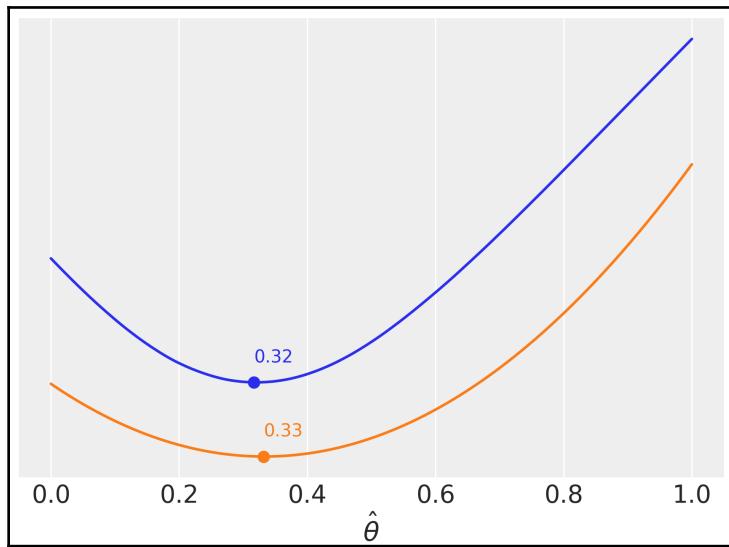


Figure 2.5

As we can see, the result looks somewhat similar for `lossf_a` is $\hat{\theta} = 0.32$ and `lossf_b` is $\hat{\theta} = 0.33$. What is interesting from this result is that the first value is equal to the median of the posterior and the last value is equal to the mean of the posterior. You can check this for yourself by computing `np.mean(theta_pos)`, `np.median(theta_pos)`. I know this is no formal proof, but the take home message is this: different loss functions are related to different point-estimates.

OK, so if we want to be formal and we want to compute a single point-estimate, we must decide which loss function we want, or in reverse, if we choose a given point-estimate, we are implicitly (and maybe even unconsciously) deciding on a loss function. The advantage of explicitly choosing a loss function is that we can tailor the function to our problem, instead of using some predefined rule that may not be suitable in our particular case. For example, in many problems, the cost of making a decision is asymmetric; it is not the same to decide whether it is safe or not to administrate a vaccine to children under five and being right, than being wrong. Making a bad decision could cost thousands of lives and produce a health crisis that could have been avoided by administrating a cheap and safe vaccine. Thus, if our problem demands it, we can construct an asymmetric loss function. It is also important to notice that, as the posterior is in the form of numerical samples, we can compute complex loss functions that don't need to be restricted by mathematical convenience or mere simplicity.

The following is just a silly example of this:

```
lossf = []
for i in grid:
    if i < 0.5:
        f = np.mean(np.pi * theta_pos / np.abs(i - theta_pos))
    else:
        f = np.mean(1 / (i - theta_pos))
    lossf.append(f)

mini = np.argmin(lossf)
plt.plot(grid, lossf)
plt.plot(grid[mini], lossf[mini], 'o')
plt.annotate(' {:.2f}'.format(grid[mini]),
             (grid[mini] + 0.01, lossf[mini] + 0.1))
plt.yticks([])
plt.xlabel(r'$\hat{\theta}$')
```

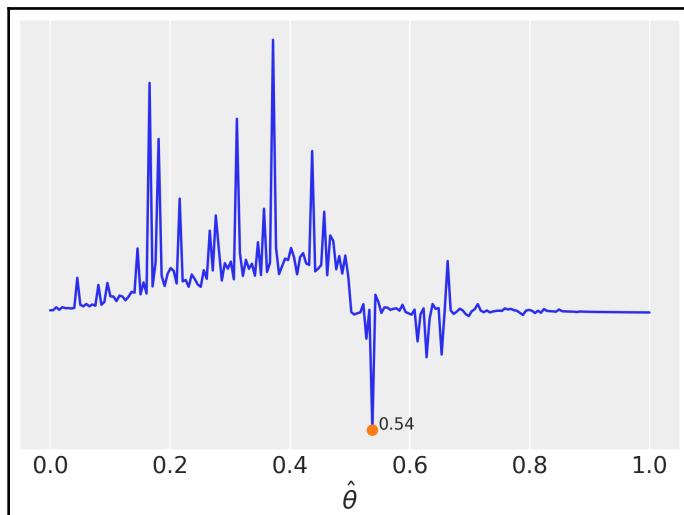


Figure 2.6

All this being said, I would like to clarify a point. It is not true that every time people uses a point-estimate they are truly thinking in terms of loss functions. In fact, loss functions are not very common in many of the scientific fields I am more or less familiar with. People often choose the median, just because it is more robust to outliers than the mean, or use the mean just because is a simple and familiar concept, or because they think their observable is truly the average of some process at some level, like molecules bouncing to each other or genes interacting with themselves and the environment.

We just saw a very brief and superficial introduction to loss functions. If you want to learn more about this, you can try reading about **decision theory**, the field that study formal decision making.

Gaussians all the way down

We introduced the main Bayesian notions using the beta-binomial model mainly because of its simplicity. Another very simple model is the Gaussian or normal model. Gaussians are very appealing from a mathematical point of view because working with them is easy; for example, we know that the **conjugate prior of the Gaussian mean is the Gaussian itself**. Besides, there are many phenomena that can be nicely approximated using Gaussians; essentially, almost every time that we measure the average of something, using a *big enough* sample size, that average will be distributed as a Gaussian. The details of when this is true, when this is not true, and when this is more or less true, are elaborated in the **central limit theorem (CLT)**; you may want to stop reading now and search about this really *central* statistical concept (very bad pun intended).

Well, we were saying that many phenomena are indeed averages. Just to follow a cliché, the height (and almost any other trait of a person, for that matter) is the result of many environmental factors and many genetic factors, and hence we get a nice Gaussian distribution for the height of adult people. Well, indeed we get a mixture of two Gaussians, which is the result of overlapping the distribution of heights of women and men, but you get the idea. In summary, Gaussians are easy to work with and they are more or less abundant in nature, and hence many of the statistical methods you may already know, or have at least heard of, are based on normality assumptions. Thus, it is important to learn how to build these models, and then it is also equally important to learn how to relax the normality assumptions, something surprisingly easy in a Bayesian framework and with modern computational tools such as PyMC3.

Gaussian inferences

Nuclear magnetic resonance (NMR) is a powerful technique that's used to study molecules and also living things such as humans or yeast (because, after all, *we are just a bunch of molecules*). NMR allows you to measure different kinds observable quantities that are related to unobservable and interesting molecular properties. One of these observables is known as **chemical shift**; we can only get chemical shifts for the nuclei of certain types of atoms.

All of this is in the domain of quantum chemistry and the details are irrelevant for this discussion, but let me explain the name chemical shift; *shifts* because we measured a signal shift from a reference value, and *chemical* because the shift is related somehow to the chemical environment on the nuclei we are measuring. For all we care at the moment, we could have been measuring the height of a group of people, the average time to travel back home, the weight of bags of oranges, or even a discrete variable like the number of sexual partners of the Tokay gecko if we are willing to approximate this number as a continuous variable, something that maybe makes sense for very promiscuous geckos! As I am not a ethologist, nor a paparazzi, I do not have a clue if this is a good approximation. But keep in mind that we do not need our data to be truly Gaussian (or any other distribution, for that matter): we only demand that a Gaussian is a *reasonable* approximation to our data. For this example, we have 48 chemical shifts values that we can load into a NumPy array and plot by using the following code:

```
data = np.loadtxt('../data/chemical_shifts.csv')
az.plot_kde(data, rug=True)
plt.yticks([0], alpha=0)
```

The KDE plot of this dataset shows a Gaussian-like distribution, except for two data points that are far away from the mean:

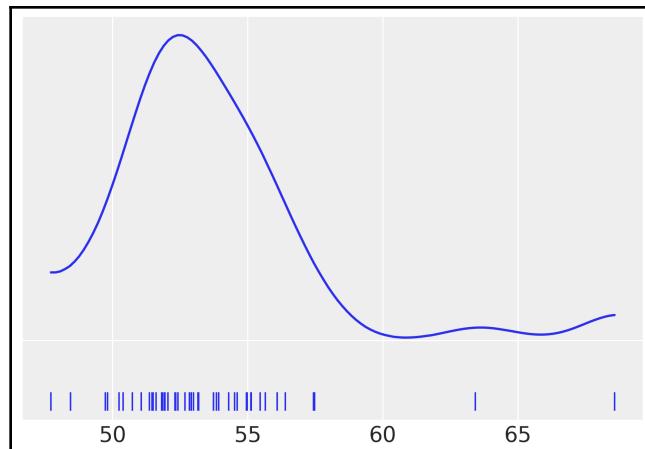


Figure 2.7

Let's forget about those two points for a moment and assume that a Gaussian distribution is a proper description of the data. Since we do not know the mean or the standard deviation, we must set priors for both of them. Therefore, a reasonable model could be:

$$\begin{aligned}\mu &\sim U(l, h) \\ \sigma &\sim |\mathcal{N}(0, \sigma_\sigma)| \\ y &\sim \mathcal{N}(\mu, \sigma)\end{aligned}\tag{2.2}$$

Thus, μ comes from a uniform distribution with boundaries l and h , which are the lower and upper bounds, respectively, and σ comes from a half-normal distribution with a standard deviation of σ_σ . A half-Normal distribution is like the regular Normal distribution but restricted to positive values (including zero). You can get samples from a half-normal by sampling from a normal distribution and then taking the absolute value of each sampled value. Finally, in our model, the data y comes from a Normal distribution with the parameters μ and σ . Using Kruschke-style diagrams:

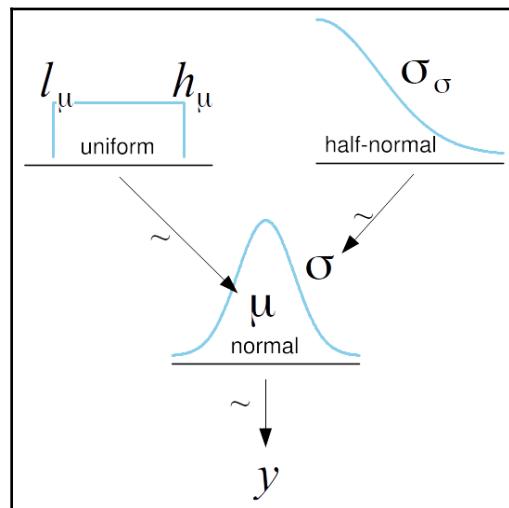


Figure 2.8

If we do not know the possible values of μ and σ , we can set priors reflecting our ignorance. One option is to set the boundaries of the uniform distribution to be $l = 40$, $h = 75$, which is a range larger than the range of the data. Alternatively, we can choose a range based on our previous knowledge. We may know that this is not physically possible to have values below 0 or above 100 for this type of measurement. In such a case, we can set the prior for the mean as a uniform with parameters $l = 0$, $h = 100$. For the half normal, we can set $\sigma_0 = 10$, just a large value for the data. Using PyMC3, we can write the model as follows:

```
with pm.Model() as model_g:
    mu = pm.Uniform('mu', lower=40, upper=70)
    sigma = pm.HalfNormal('sigma', sd=10)
    y = pm.Normal('y', mu=mu, sd=sigma, observed=data)
    trace_g = pm.sample(1000)
az.plot_trace(trace_g)
```

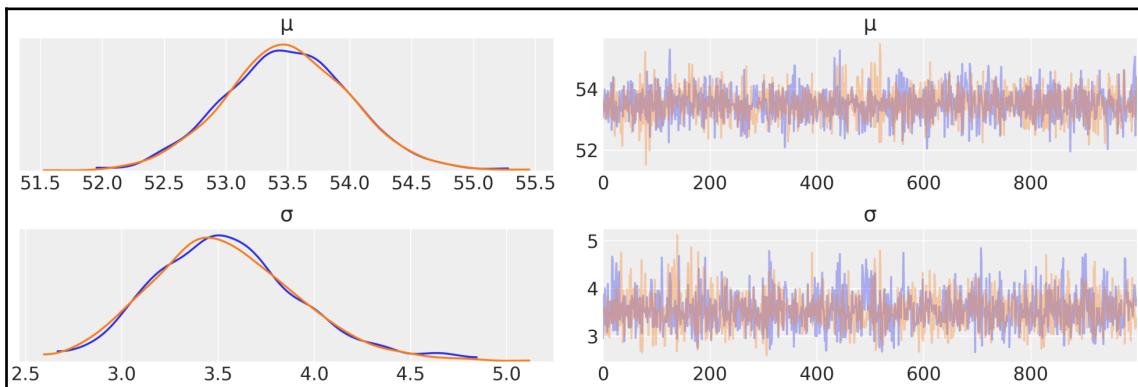


Figure 2.9

As you may have noticed, in *Figure 2.9*, which was generated with the ArviZ function `plot_trace`, it has one row for each parameter. For this model, the posterior is bi-dimensional, and so *Figure 2.9* is showing the *marginal distributions* of each parameter. We can use the `plot_joint` function from ArviZ to see what the bi-dimensional posterior looks like, together with the marginal distributions for μ and σ :

```
az.plot_joint(trace_g, kind='kde', fill_last=False)
```

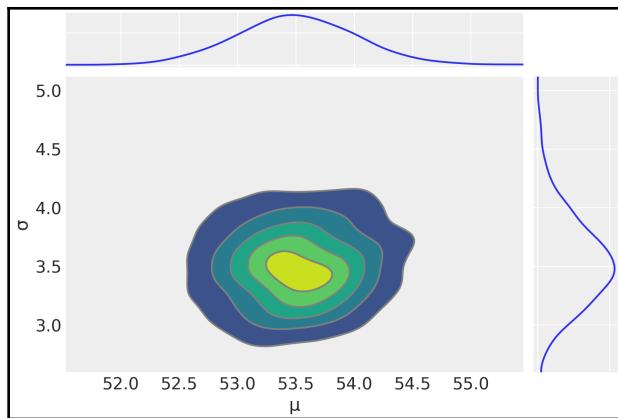


Figure 2.10

If you want to get access to the values for any of the parameters stored in a `trace` object, you can index the trace with the name of the parameter in question. As a result, you will get a NumPy array. Try doing `trace_g['σ']` or `az.plot_kde(trace_g['σ'])`. By the way, using Jupyter Notebook/lab, you can get characters such as σ by writing \sigma in a code cell and then hitting the Tab key.

We are going to print the summary for later use:

```
az.summary(trace_g)
```

	mean	sd	mc error	hpd 3%	hpd 97%	eff_n	r_hat
μ	53.49	0.50	0.00	52.5	54.39	2081.0	1.0
σ	3.54	0.38	0.01	2.8	4.22	1823.0	1.0

Now that we have computed the posterior, we can use it to simulate data and check how consistent the simulated data is with respect to the observed data. If you remember from Chapter 1, *Thinking Probabilistically*, we generically call this type of comparisons **posterior predictive checks**, because we are using the posterior to make predictions, and are using those predictions to check the model. Using PyMC3 is really easy to get posterior predictive samples if you use the `sample_posterior_predictive` function. With the following code, we are generating 100 predictions from the posterior, each one of the same size as the data. Notice that we have to pass the trace and the model to `sample_posterior_predictive`, while the other arguments are optional:

```
y_pred_g = pm.sample_posterior_predictive(trace_g, 100, model_g)
```

The `y_pred_g` variable is a dictionary, with the `keys` being the name of the observed variable in our model and the `values` an array of shape `(samples, size)`, in this case, `(100, len(data))`. We have a dictionary because we could have models with more than one observed variable. We can use the `plot_ppc` function for a visual posterior predictive check:

```
data_ppc = az.from_pymc3(trace=trace_g, posterior_predictive=y_pred_g)
ax = az.plot_ppc(data_ppc, figsize=(12, 6), mean=False)
ax[0].legend(fontsize=15)
```

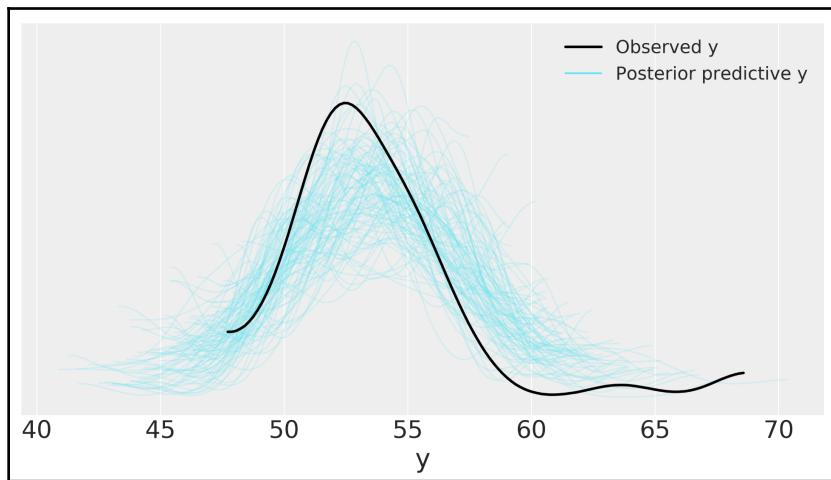


Figure 2.11

In *Figure 2.11*, the single (black) line is a KDE of the data and the many semitransparent (cyan) lines are KDEs computed from each one of the 100 posterior predictive samples. The semitransparent (cyan) lines reflect the uncertainty we have about the inferred distribution of the predicted data. Sometimes, when you have very few data points. A plot like this one could show the predicted curves as *hairy* or *wonky*; this is due to the way the KDE is implemented in ArviZ. The density is estimated within the actual range of the data passed to the `kde` function, while outside this range the density is assumed to be zero. While some could reckon this as a bug, I think it's a feature, since it's reflecting a property of the data instead of over-smoothing it.

From *Figure 2.11*, we can see that the mean of the simulated data is slightly displaced to the right and that the variance seems to be larger for the simulated data than for the actual data. This is a direct consequence of the two observations that are separated from the bulk of the data. Can we use this plot to confidently say that the model is faulty and needs to be changed? Well, as always, the interpretation of the model and its evaluation is context-dependent. Based on my experience for these type of measures and the ways I generally use this data, I would say this model is a reasonable enough representation of the data and useful for most of my analysis. Nevertheless, in the next section, we are going to learn how to refine `model_g` and get predictions that match the data even closer.

Robust inferences

One objection you may have with the `model_g` model is that we are assuming a normal distribution, but we have two data points on the tails of the distribution, making the *normally* assumption a little bit *forced*. Since the tails of the normal distribution falls quickly as we move away from the mean, the normal distribution (at least an anthropomorphized one) is *surprised* by seeing those two points and *reacts* by moving itself toward those points and increasing the standard deviation. We can imagine those points as having an *excessive weight* determining the parameters of the normal distribution. So, what can we do?

One option is to declare those points as outliers and remove them from the data. We may have a valid reason to discard those points—maybe a malfunction of the equipment or a human error while measuring those two data points. Sometimes, we can even fix those data points, for example, if we realize they are just a result of bad coding while cleaning the data. On many occasions, we may also want to automatize the outlier elimination process by using one of the many *outlier rules*. Two of them are:

- Any data point below 1.5 times the interquartile range from the lower quartile or 1.5 times the interquartile range above the upper quartile is an outlier
- Any data point below or above two times the standard deviation of the data should be declared an outlier and banished from our data

Instead of using one of these outlier rules to manipulate the data, we can change the model, as explained in the next section.

Student's t-distribution

As a general rule, Bayesians prefer to encode assumptions directly into the model by using different priors and likelihoods rather than through *ad hoc* heuristics such as outlier removal rules.

One very useful option when dealing with outliers and Gaussian distributions is to replace the Gaussian likelihood with a Student's t-distribution. This distribution has three parameters: the mean, the scale (analogous to the standard deviation), and the degrees of freedom, which is usually referred to using the Greek letter ν , that can vary in the interval $[0, \infty]$. Following Kruschke's nomenclature, we are going to call ν the **normality parameter**, since it is in charge of controlling how normal-like the Student's t-distribution is. For a value of $\nu = 1$, we get a distribution with very *heavy tails*, which is also known as **Cauchy** or **Lorentz distribution**. This last name is especially popular among physicists. By *heavy tails*, we mean that it is more probable to find values away from the mean compared to a Gaussian, or in other words values are not as concentrated around the mean as in a *lighter tail* distribution like the Gaussian. For example, 95% of the values from a Cauchy distribution are found between -12.7 and 12.7. Instead, for a Gaussian (with a standard deviation of one), this occurs between -1.96 and 1.96. On the other side of the parameter space, when ν approaches infinity, we recover the Gaussian distribution (you can't be more normal than the normal distribution, right?). A very curious feature of the Student's t-distribution is that it has no defined mean when $\nu \leq 1$. Of course, in practice, any finite sample from a Student's t-distribution is just a bunch of numbers from which it is always possible to compute an empirical mean. It's the theoretical distribution itself the one without a defined mean. Intuitively, this can be understood as follows: the tails of the distribution are so heavy that at any moment it could happen that we get a sampled value from almost anywhere from the real line, so if we keep getting numbers, we will never approximate to a fixed value. Instead, the estimate will keep wandering around. Just try the following code several times (and then change `df` to a larger number, such as 100):

```
np.mean(stats.t(loc=0, scale=1, df=1).rvs(100))
```

In a similar fashion, the variance of this distribution is only defined for values of $\nu > 2$. So, be careful that the scale of the Student's t-distribution is not the same as the standard deviation. For $\nu \leq 2$, the distribution has no defined variance and hence no defined standard deviation. The scale and the standard deviation become closer and closer as ν approaches infinity:

```
plt.figure(figsize=(10, 6))
x_values = np.linspace(-10, 10, 500)
for df in [1, 2, 30]:
    distri = stats.t(df)
    x_pdf = distri.pdf(x_values)
```

```

plt.plot(x_values, x_pdf, label=fr'$\nu = {df}$', lw=3)

x_pdf = stats.norm.pdf(x_values)
plt.plot(x_values, x_pdf, 'k--', label=r'$\nu = \infty$')
plt.xlabel('x')
plt.yticks([])
plt.legend()
plt.xlim(-5, 5)

```

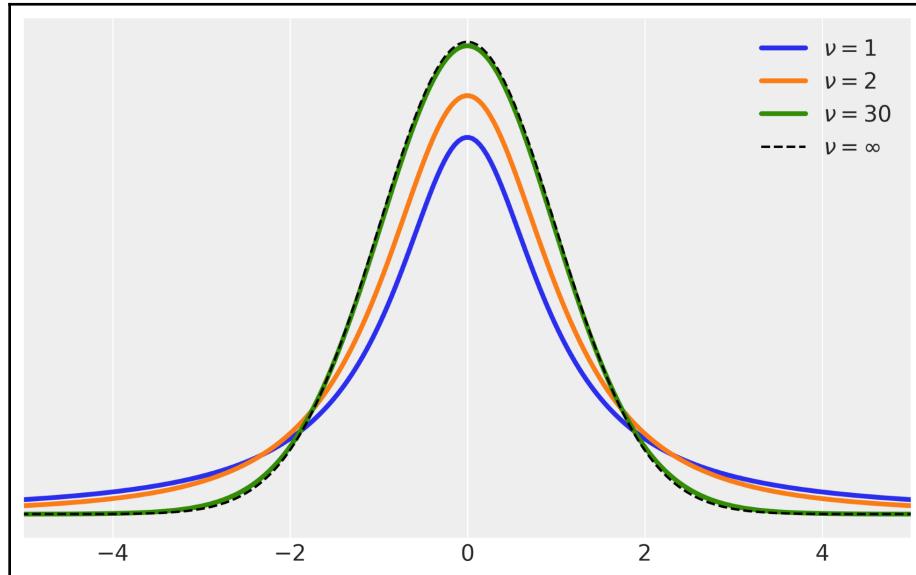


Figure 2.12

We are going to rewrite the previous model by replacing the Gaussian distribution with the Student's t-distribution:

$$\begin{aligned}
\mu &\sim U(l, h) \\
\sigma &\sim |\mathcal{N}(0, \sigma_\sigma)| \\
\nu &\sim \text{Exp}(\lambda) \\
y &\sim \mathcal{T}(\mu, \sigma, \nu)
\end{aligned} \tag{2.3}$$

Because the Student's t-distribution has one more parameter (ν) than the Gaussian, we need to specify one more prior. We are going to set ν as an exponential distribution with a mean of 30. From *Figure 2.12*, we can see that a Student's t-distribution with $\nu = 30$ looks pretty similar to a Gaussian (even when it is not). In fact, from the same diagram, we can see that *most of the action* happens for relatively small values of ν . Hence, we can say that the exponential prior with a mean of 30 is a weakly informative prior telling the model we more or less think ν should be around 30, but can move to smaller and larger values with ease. In many problems, estimating ν is of no direct interest. Graphically, we have:

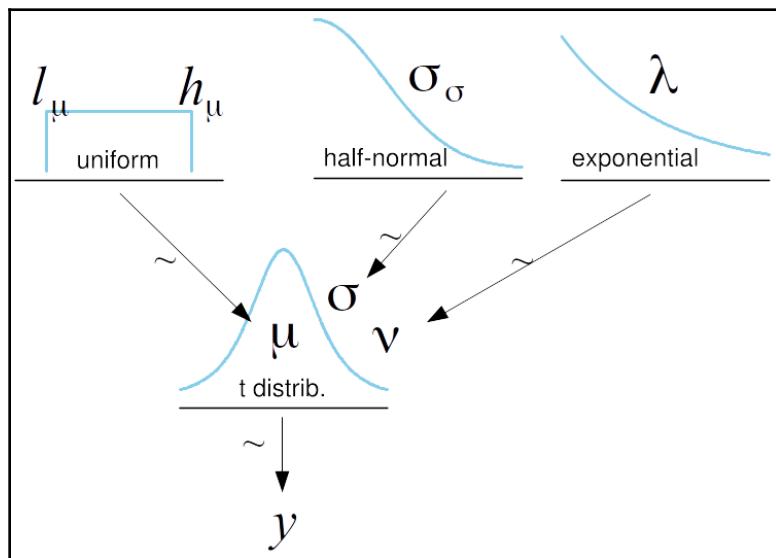


Figure 2.13

As usual, PyMC3 allows us to (re)write models just by specifying a few lines. The only cautionary word here is that the exponential in PyMC3 is parameterized with the inverse of the mean:

```
with pm.Model() as model_t:
    μ = pm.Uniform('μ', 40, 75)
    σ = pm.HalfNormal('σ', sd=10)
    ν = pm.Exponential('ν', 1/30)
    y = pm.StudentT('y', mu=μ, sd=σ, nu=ν, observed=data)
    trace_t = pm.sample(1000)
    az.plot_trace(trace_t)
```

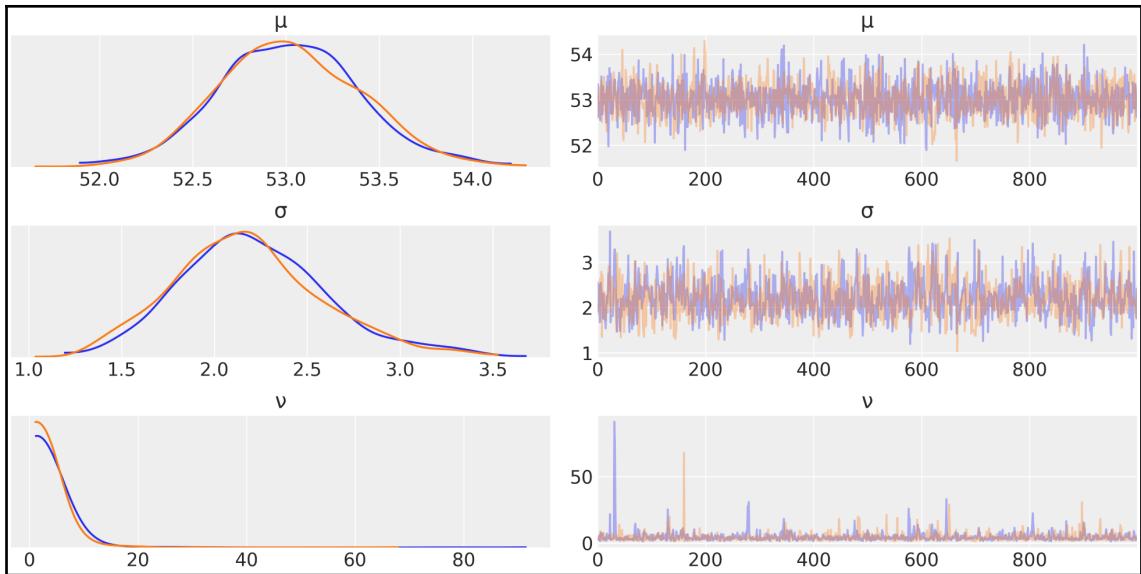


Figure 2.14

Compare the trace from `model_g` (Figure 2.9) with the trace of `model_t` (Figure 2.14). Now, print the summary of `model_t` and compare it with the one from `model_g`. Before you keep reading, take a moment to spot the difference between both results. Did you notice something interesting?

```
az.summary(trace_t)
```

	mean	sd	mc error	hpdi 3%	hpdi 97%	eff_n	r_hat
μ	53.00	0.39	0.01	52.28	53.76	1254.0	1.0
σ	2.20	0.39	0.01	1.47	2.94	1008.0	1.0
ν	4.51	3.35	0.11	1.10	9.27	898.0	1.0

The estimation of μ between both models is similar, with a difference of ≈ 0.5 . The estimation of σ changes from ≈ 3.5 to ≈ 2.1 . This is a consequence of the Student's t-distribution giving less weight (being less shocked) by values away from the mean. We can also see that $\nu \approx 4.5$, that is, we have a *not very Gaussian-like* distribution and instead one with heavier tails.

Now, we are going to do a posterior predictive check of the Student's t model and we are going to compare it to the Gaussian model:

```
y_ppc_t = pm.sample_posterior_predictive(  
    trace_t, 100, model_t, random_seed=123)  
y_pred_t = az.from_pymc3(trace=trace_t, posterior_predictive=y_ppc_t)  
az.plot_ppc(y_pred_t, figsize=(12, 6), mean=False)  
ax[0].legend(fontsize=15)  
plt.xlim(40, 70)
```

Using the Student's t-distribution in our model leads to predictive samples that seem to better fit the data in terms of the location of the peak of the distribution and also its spread. Notice how the samples extend far away from the bulk of the data, and how a few of the predictive samples look very flat. This is a direct consequence of the Student's t-distribution expecting to see data points far away from the mean or bulk of the data, and this is the reason we are setting `xlim` to [40, 70]:

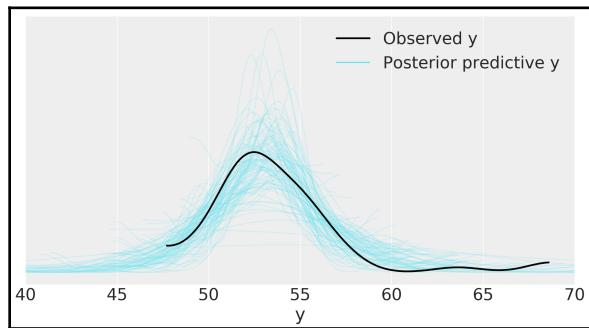


Figure 2.15

The Student's t-distribution allows us to have a more **robust estimation** because the outliers have the effect of decreasing ν , instead of pulling the mean toward them and increasing the standard deviation. Thus, the mean and the scale are estimated by weighting more data points from the bulk of the data than those apart from it. Once again, it is important to remember that the scale is not the standard deviation. Nevertheless, the scale is related to the spread of the data; the lower its value, the more concentrated the distribution. Also, for values of $\nu \gtrsim 2$, the value of the scale tends to be pretty close (at least for most practical purposes) to the value estimated after removing the outliers. So, as a rule of thumb, for values of ν not too small, and taking into account that it is not theoretically fully correct, we can consider the scale of a Student's t-distribution as a reasonable practical proxy for the standard deviation of the data after removing outliers.

Groups comparison

One pretty common statistical analysis is group comparison. We may be interested in how well patients respond to a certain drug, the reduction of car accidents by the introduction of a new traffic regulation, student performance under different teaching approaches, and so on.

Sometimes, this type of question is framed under the hypothesis testing scenario with the goal of declaring a result statistically significant. Relying only on statistical significance can be problematic for many reasons: on the one hand, statistical significance is not equivalent to practical significance; on the other hand, a really small effect can be declared significant just by collecting enough data. The idea of hypothesis testing is connected to the concept of p-values. This is not a fundamental connection but a cultural one; people are used to thinking that way mostly because that's what they learn in most introductory statistical courses. There is a long record of studies and essays showing that, more often than not, p-values are used and interpreted the wrong way, even by people who are using them on a daily basis.

Instead of doing hypothesis testing, we are going to take a different route and we are going to focus on estimating the **effect size**, that is, quantifying the difference between two groups. One advantage of thinking in terms of effect size is that we move away from the yes-no questions like; Does it work?, Is there any effect? to the more nuance type of question like; How well does it work? How large/small is the effect?



The **effect size** is just a way to quantify the size of the difference between two groups.

Sometimes, when comparing groups, people talk about a control group and a treatment group (or maybe more than one control and treatment groups). This makes sense, for example, when we want to test a new drug: because of the placebo effect and other reasons, we want to compare the new drug (the treatment) against a control group (a group not receiving the drug). In this case, we want to know how well one drug works compared to doing nothing (or, as is generally done, against the placebo effect). One interesting alternative question will be to ask how good a new drug is compared with the (already approved) most popular drug to treat that illness. In such a case, the control group cannot be a placebo; it should be the other drug. Bogus control groups are a splendid way to lie using statistics.

For example, imagine you work for a dairy product company that wants to sell overly sugared yogurts to kids by telling their dads and moms that this particular yogurt boosts the immune system or help their kids grow stronger. One way to cheat and falsely back up your claims with data is by using milk or even water as a control group, instead of another cheaper, less sugary, less marketed yogurt. It may sound silly when I put this way, but there is a lot of actual *research* done this way. In fact, I am describing actual papers, not imaginary hypothetical scenarios. When someone says something is harder, better, faster, stronger, remember to ask what the baseline used for the comparison was.

To compare groups, we must decide which feature (or features) we are going to use for the comparison. A very common feature is the mean of each group. Because we are Bayesian, we will work to obtain a posterior distribution of the differences of means between groups and not just a point-estimate of the differences. To help us see and interpret such a posterior, we are going to use three tools:

- A posterior plot with a reference value
- The Cohen's d
- The probability of superiority

In the previous chapter, we already saw an example of how to use the function `az.plot_posterior` with a reference value; we will see another example soon. The novelties here are the **Cohen's d** and the **probability of superiority**, which are two popular ways to express the effect size. Let's take a look at them.

Cohen's d

A common way to measure the effect size is **Cohen's d**, which is defined as follows:

$$\frac{\mu_2 - \mu_1}{\sqrt{\frac{\sigma_2^2 + \sigma_1^2}{2}}} \quad (2.4)$$

According to this expression, the effect size is the difference of the means with respect to the pooled standard deviation of both groups. Because we can get a posterior distribution of means and standard deviations, we can compute a posterior distribution of Cohen's d values. Of course, if we just need or want one single value, we can compute the mean of that posterior distribution and get a single Cohen's d value. Generally, when computing a pooled standard deviation, we take into account the sample size of each group explicitly, but the previous formula is omitting the sample size of both groups. The reason for this is that we are getting the values of the standard deviation from the posterior and thus we are already accounting for the standard deviations' uncertainty.



Cohen's d is a way to measure the effect size, where the difference of the means are standardized by considering the pooled standard deviations of both groups.

Cohen's d introduces the variability of each group by using their standard deviations. This is really important, as differences of one when you have a standard deviation of 0.1 seems large compared to the same difference when the standard deviation is 10. Also, a change of x units from one group to another could be explained by every individual data point changing exactly x units or by half of them not changing and the other half changing $2x$ units, and by many other combinations. Thus, including the intrinsic variations of groups is a way to put the differences in context. Re-scaling (standardizing) the differences helps us make sense of the importance of the different between groups, even when we are not very familiar with the scale used for the measurements.



A Cohen's d can be interpreted as a Z-score (a standard score). A Z-score is the signed number of standard deviations by which a value differs from the mean value of what is being observed or measured. Thus, a Cohen's d of 0.5 could be interpreted as a difference of 0.5 standard deviation of one group with respect to the other.

Even when the differences of means are standardized, we may still need to calibrate ourselves based on the context of a given problem to be able to say if a given value is *big*, *small*, *medium*, and so on. Fortunately, this calibration can be acquired with enough practice. Just as an example, if we are used to performing several analyses for more or less the same type of problems, we can get used to a Cohen's d of say 1, so when we get a Cohen's d of say 2, we know we have something important (or someone made a mistake somewhere!). If you do not have this practice yet, you can ask a domain expert for their valuable input. A very nice web page to explore what different values of Cohen's d look like is <http://rpsychologist.com/d3/cohend>. On that page, you will also find other ways to express an effect size; some of them could be more intuitive, such as the probability of superiority, which we will discuss next.

Probability of superiority

This is another way to report the effect size, and this is defined as the probability that a data point taken at random from one group has a larger value than one also taken at random from the other group. If we assume that the data we are using is distributed as a normal, we can compute the probability of superiority from the Cohen's d using the following expression:

$$ps = \Phi\left(\frac{\delta}{\sqrt{2}}\right) \quad (2.5)$$

Here, Φ is the cumulative normal distribution and δ is the Cohen's d. We can compute a point-estimate of the probability of superiority (what is usually reported), or we can compute the whole posterior distribution of values. If we are OK with the normality assumption, we can use this formula to get the probability of superiority from the Cohen's d. Otherwise, as we have samples from the posterior, we can directly compute it (see the *Exercises* section). This is a very nice advantage of using **Markov chain Monte Carlo (MCMC)** methods; once we get samples from the posterior, we can compute many quantities from it.

The tips dataset

To explore the subject matter of this section, we are going to use the `tips` dataset. This data was reported for the first time by Bryant, P. G. and Smith, M (1995) in *Practical Data Analysis: Case Studies in Business Statistics*.

We want to study the effect of the day of the week on the amount of tips at a restaurant. For this example, the different groups are the days. Notice there is no control group or treatment group. If we wish, we can arbitrarily establish one day, for example, Thursday, as the reference or *control*. For now, let's start the analysis by loading the dataset as a pandas DataFrame using just one line of code. If you are not familiar with pandas, the `tail` command is used to show the last rows of a DataFrame (you can also try using `head`):

```
tips = pd.read_csv('~/data/tips.csv')
tips.tail()
```

	total_bill	tip	sex	smoker	day	time	size
239	29.03	5.92	Male	No	Sat	Dinner	3
240	27.18	2.00	Female	Yes	Sat	Dinner	2
241	22.67	2.00	Male	Yes	Sat	Dinner	2
242	17.82	1.75	Male	No	Sat	Dinner	2
243	18.78	3.00	Female	No	Thurs	Dinner	2

From this DataFrame, we are only going to use the `day` and `tip` columns. We can plot our data using the `violinplot` function from `seaborn`:

```
sns.violinplot(x='day', y='tip', data=tips)
```

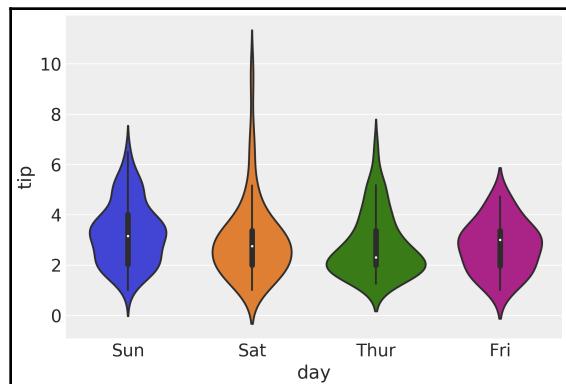


Figure 2.16

Just to simplify things, we are going to create three variables: the `y` variable, representing the `tips`, the `idx` variable, a categorical dummy variable to encode the days with numbers, that is, [0, 1, 2, 3] instead of [Thursday, Friday, Saturday, Sunday], and finally the `groups` variable, with the number of groups (4):

```
tip = tips['tip'].values
idx = pd.Categorical(tips['day'],
                     categories=['Thur', 'Fri', 'Sat', 'Sun']).codes
groups = len(np.unique(idx))
```

The model for this problem is almost the same as `model_g`; the only difference is that now μ and σ are going to be vectors instead of scalar variables. PyMC3 syntax is extremely helpful for this situation: instead of writing `for` loops, we can write our model in a vectorized way. This means that for the priors, we pass a `shape` argument and for the likelihood, we properly index the `means` and `sds` variables using the `idx` variable:

```
with pm.Model() as comparing_groups:
    mu = pm.Normal('mu', mu=0, sd=10, shape=groups)
    sigma = pm.HalfNormal('sigma', sd=10, shape=groups)

    y = pm.Normal('y', mu=mu[idx], sd=sigma[idx], observed=tip)

    trace_cg = pm.sample(5000)
az.plot_trace(trace_cg)
```

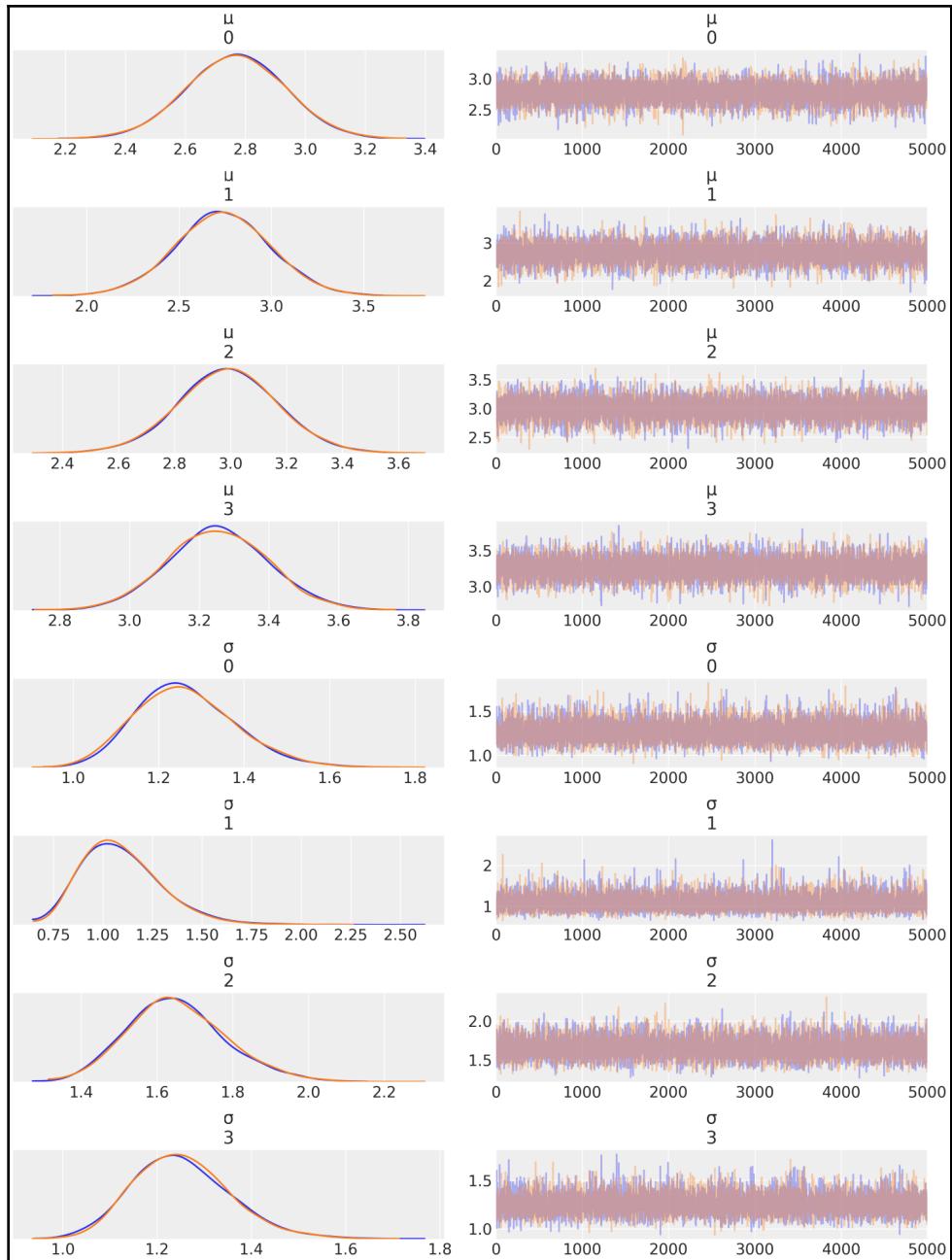


Figure 2.17

The following code is just a way of plotting the difference without repeating the comparison. Instead of plotting the *all-against-all* matrix, we are just plotting the upper triangular portion:

```
dist = stats.norm()

_, ax = plt.subplots(3, 2, figsize=(14, 8), constrained_layout=True)

comparisons = [(i, j) for i in range(4) for j in range(i+1, 4)]
pos = [(k, l) for k in range(3) for l in (0, 1)]

for (i, j), (k, l) in zip(comparisons, pos):
    means_diff = trace_cg['μ'][:, i] - trace_cg['μ'][:, j]
    d_cohen = (means_diff / np.sqrt((trace_cg['σ'][:, i]**2 +
    trace_cg['σ'][:, j]**2) / 2)).mean()
    ps = dist.cdf(d_cohen/(2**0.5))
    az.plot_posterior(means_diff, ref_val=0, ax=ax[k, l])
    ax[k, l].set_title(f'$\mu_{i}-\mu_{j}$')
    ax[k, l].plot(
        0, label=f"Cohen's d = {d_cohen:.2f}\nProb sup = {ps:.2f}",
        alpha=0)
    ax[k, l].legend()
```

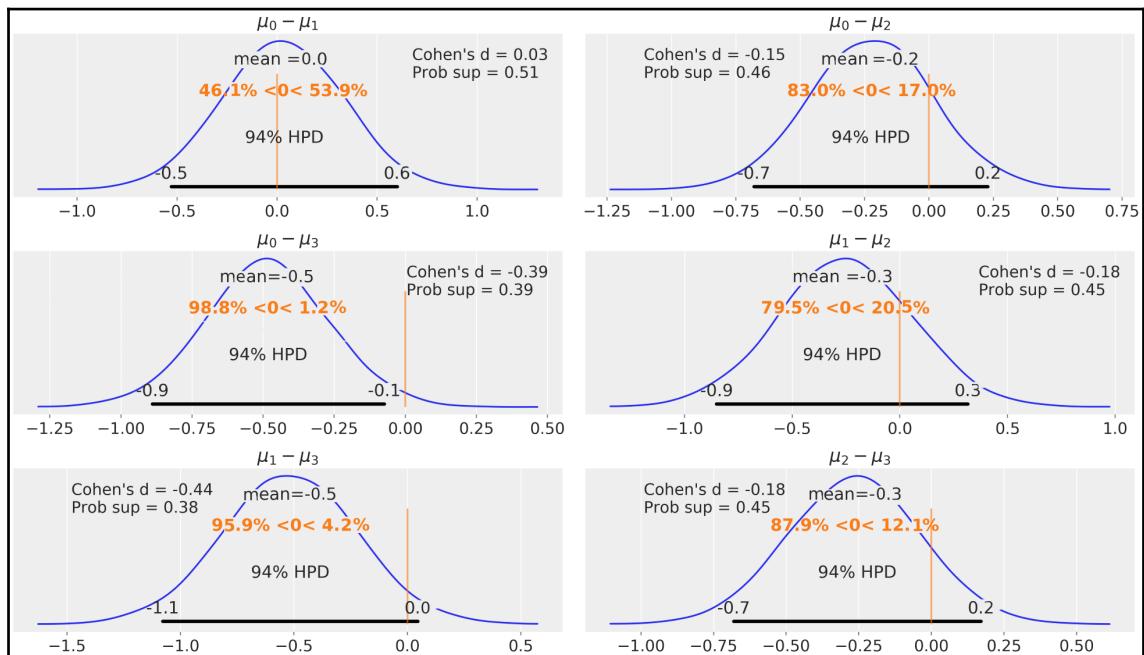


Figure 2.18

One way to interpret these results is by comparing the reference value with the HPD interval. According to the previous diagram, we have only one case when the 94% HPD excludes the reference value of zero, that is, the difference in tips between Thursday and Sunday. For all the other examples, we cannot rule out a difference of zero (according to the HPD-reference-value-overlap criteria). But even for that case, is an average difference of ≈ 0.5 dollars large enough? Is that difference enough to accept working on Sunday and missing the opportunity to spend time with family or friends? Is that difference enough to justify averaging the tips over the four days and giving every waitress and waiter the same amount of tip money? Those kinds of questions cannot be answered by statistics; they can only be informed by statistics.

Hierarchical models

Suppose we want to analyze the quality of water in a city, so we take samples by dividing the city into neighborhoods. We may think we have two options to analyze this data:

- Study each neighborhood as a separate entity
- Pool all the data together and estimate the water quality of the city as a single big group

Both options could be reasonable, depending on what we want to know. We can justify the first option by saying we obtain a more detailed view of the problem, which otherwise could become invisible or less evident if we average the data. The second option can be justified by saying that if we pool the data, we obtain a bigger sample size and hence a more accurate estimation. Both are good reasons, but we can do something else, something in-between. We can build a model to estimate the water quality of each neighborhood and, at the same time, estimate the quality of the whole city. This type of model is known as a **hierarchical model** or **multilevel model**, because we model the data using a hierarchical structure or one with multiple levels.

So, how do we build a hierarchical model? Well, in a nutshell, instead of fixing the parameters of our priors to some constant numbers, we estimate them directly from the data by placing shared priors over them. These higher-level priors are often called **hyper-priors**, and their parameters **hyperparameters**; hyper means *over* in Greek. Of course, it's also possible to put priors over the hyper-priors and create as many levels as we want; the problem is that the model rapidly becomes difficult to understand and unless the problem really demands more structure, adding more levels *than necessary* does not help to make better inferences. On the contrary, we end up entangled in a web of hyper-priors and hyperparameters without the ability to assign any meaningful interpretation to them, partially spoiling the advantages of model-based statistics. After all, the main idea of building models is to make sense of data, and thus models should reflect (and take advantage of) the structure in the data.

To illustrate the main concepts of hierarchical models, we are going to use a toy model of the water quality example we discussed at the beginning of this section and we are going to use synthetic data. Imagine we have collected water samples from three different regions of the same city and we have measured the lead content of water; samples with lead concentration above recommendations from the **World Health Organization (WHO)** are marked with zero and samples with values below the recommendations are marked with one. This is just a pedagogic example; in a more realistic example, we would have a continuous measurement of lead concentration and probably many more groups. Nevertheless, for our current purposes, this example is good enough to uncover the details of hierarchical models.

We can generate the synthetic data with the following code:

```
N_samples = [30, 30, 30]
G_samples = [18, 18, 18]

group_idx = np.repeat(np.arange(len(N_samples)), N_samples)
data = []
for i in range(0, len(N_samples)):
    data.extend(np.repeat([1, 0], [G_samples[i], N_samples[i]-
        G_samples[i]]))
```

We are simulating an experiment where we have measured three groups, each one consisting of a certain number of samples; we store the total number of samples per group in the `N_samples` list. Using the `G_samples` list, we keep a record of the number of good quality samples per group. The rest of the code is there just to generate a list of the data, filled with zeros and ones.

The model is essentially the same one we used for the coin problem, except for two important features:

- We have defined two hyper-priors that will influence the beta prior.
- Instead of putting hyper-priors on the parameters α and β , we are indirectly defining them in terms of μ , the mean of the beta distribution, and κ , the precision (the concentration) of the beta distribution. The precision is analog to the inverse of the standard deviation; the larger the value of κ , the more concentrated the beta distribution will be:

$$\begin{aligned}
 \mu &\sim \text{Beta}(\alpha_\mu, \beta_\mu) \\
 \kappa &\sim |\text{Normal}(0, \sigma_\kappa)| \\
 \alpha &= \mu * \kappa \\
 \beta &= (1 - \mu) * \kappa \\
 \theta_i &\sim \text{Beta}(\alpha_i, \beta_i) \\
 y_i &\sim \text{Bern}(\theta_i)
 \end{aligned} \tag{2.6}$$

Notice we are using the subindex i to indicate that the model has groups with different values for some of the parameters. That is, not all parameters are shared between the groups. Using Kruschke diagrams, it is evident that this new model has one additional level compared to all of the models we have seen so far:

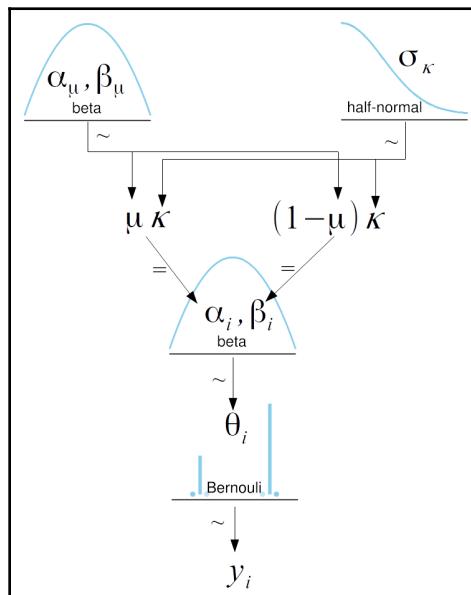


Figure 2.19

Notice how in *Figure 2.19* we have used the symbol $=$ instead of \sim to define α_i and β_i , once we know μ and κ the values of α_i and β_i becomes fully determined. Accordingly, we call this type of variable **deterministic** in opposition to **stochastic** variables such as μ , κ , or θ .

Let's talk a little bit about parameterization. Using the mean and precision is mathematically equivalent to using the α and β parameterization, implying we should get the same results. So, why are we taking this detour instead of the more direct route? There are two reasons for this:

- First, the mean and precision parameterization, while mathematically equivalent, is numerically better suited for the sampler, and thus we are more confident of the results PyMC3 is returning, we will learn about the reason for and intuition behind this statement in [Chapter 8, Inference Engines](#).
- The second reason is pedagogical. This is a concrete example showing that there is potentially more than one way to express a model. Mathematically equivalent implementations could have nevertheless practical differences; the efficiency of the sampler is one aspect to consider, while another one could be the interpretability of the model. For some specific problems or particular audiences, it could be a better choice to report the mean and precision of the beta distribution than the α and β parameters.

Let's implement and solve the model in PyMC3 so that we can keep discussing hierarchical models:

```
with pm.Model() as model_h:  
    mu = pm.Beta('mu', 1., 1.)  
    kappa = pm.HalfNormal('kappa', 10)  
  
    theta = pm.Beta('theta', alpha=mu*kappa, beta=(1.0-mu)*kappa, shape=len(N_samples))  
    y = pm.Bernoulli('y', p=theta[group_idx], observed=data)  
  
    trace_h = pm.sample(2000)  
az.plot_trace(trace_h)
```

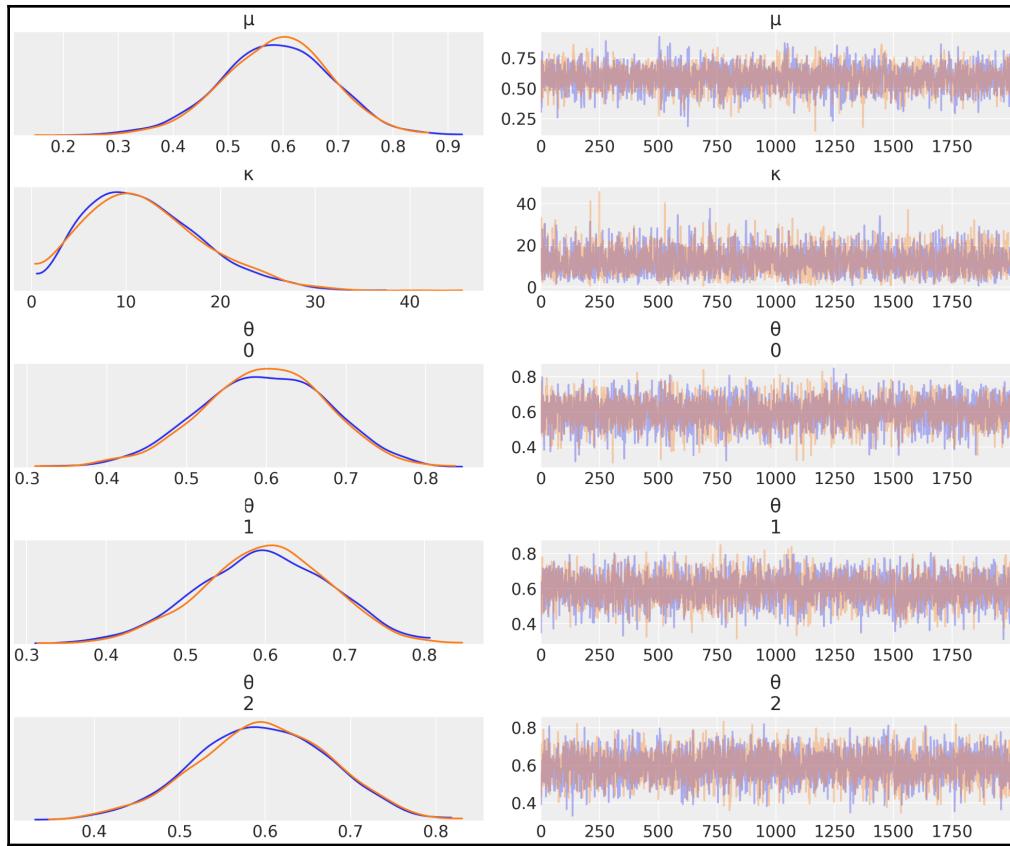


Figure 2.20

Shrinkage

To show you one of the main consequences of hierarchical models, I will require your assistance, so please join me in a brief experiment. I will need you to print and save the summary using `az.summary(trace_h)`. Then, I want you to re-run the model two more times after making small changes to the synthetic data, and always keep a record of the summary. In total, we will have three runs:

- One run setting all the elements of `G_samples` to 18
- One run setting all the elements of `G_samples` to 3
- One last run setting one element to 18 and the other two to 3

Before continuing, please take a moment to think about the outcome of this experiment. Focus on the estimated mean value of θ_i in each experiment. Based on the first two runs of the model, could you predict the outcome for the third case?

If we put the result in a table, we get something more or less like this; remember that small variations could occur due to the stochastic nature of the sampling process:

G_samples	$\theta(\text{mean})$
18, 18, 18	0.6, 0.6, 0.6
3, 3, 3	0.11, 0.11, 0.11
18, 3, 3	0.55, 0.13, 0.13

In the first row, we can see that for a dataset of 18 *good* samples out of 30 we get a mean value for θ of 0.6; remember that now the mean of θ is a vector of three elements, one per group. Then, on the second row, we have only 3 good samples out of 30 and the mean of θ is 0.11. At the end, on the last row, we get something interesting and probably unexpected. Instead of getting a mix of the mean estimates of θ from the other rows, such as 0.6, 0.11, and 0.11, we get different values, namely 0.55, 0.13, and 0.13. What on earth happened? Maybe a convergence problem or some error with the model specification? Nothing like that—we get our estimates shrunk toward the common mean. And this is totally OK, indeed this is just a consequence of our model; by using hyper-priors, we are estimating the parameters of the beta prior distribution from the data, and each group is informing the rest, which in turn is informed by the estimation of the others. Put in a more succinct way, the groups are effectively sharing information through the hyper-prior. As a result, we are observing what is known as **shrinkage**. This effect is the consequence of partially pooling the data; we are modeling the groups not as independent from each other, nor as a single big group. Instead, we have something in the middle.

Why is this useful? Because having shrinkage contributes to more stable inferences. This is, in many ways, similar to what we saw with the Student's t-distribution and the outliers, where we used heavy tail distribution results in a model that is more robust or less responsive to data points away from the mean. Introducing hyper-priors and hence inferences at a higher level results in a more conservative model (probably the first time I've used *conservative* in a flattering way), one that is less responsive to extreme values in individual groups. To illustrate this, imagine that the samples size are different from each neighborhood, some small, some larger; the smaller the sample size, the easier it is to get bogus results. At an extreme, if you take only one sample in a given neighborhood, you may just hit the only really old lead pipe in the whole neighborhood or, on the contrary, the only one made out of PVC. In one case, you will overestimate the bad quality and in the other underestimate it. Under a hierarchical model, the miss-estimation of one group will be ameliorated by the information provided by the other groups. Of course, a larger sample size will also do the trick but, more often than not, that is not an option.

The amount of shrinkage depends, of course, on the data; a group with more data will pull the estimate of the other groups harder than a group with fewer data points. If several groups are similar and one group is different, the similar groups are going to inform the others of their similarity and reinforce a common estimation, while they are going to pull toward them the estimation for the less similar group; this is exactly what we saw in the previous example.

The hyper-priors also have a part in modulating the amount of shrinkage. We can effectively use an informative prior to shrink our estimate to some reasonable value if we have trustworthy information about the group-level distribution. Nothing prevents us from building a hierarchical model with just two groups—but we would prefer to have several groups. Intuitively, the reason is that getting shrinkage is like assuming each group is a data point, and we are estimating the standard deviation at the group level. Generally, we do not trust an estimation with too few data points unless we have a strong prior to inform our estimation. Something similar is true for a hierarchical model.

We may also be interested in seeing what the estimated prior looks like. One way to do this is as follows:

```
x = np.linspace(0, 1, 100)
for i in np.random.randint(0, len(trace_h), size=100):
    u = trace_h['μ'][i]
    k = trace_h['κ'][i]
    pdf = stats.beta(u*k, (1.0-u)*k).pdf(x)
    plt.plot(x, pdf, 'C1', alpha=0.2)

u_mean = trace_h['μ'].mean()
k_mean = trace_h['κ'].mean()
dist = stats.beta(u_mean*k_mean, (1.0-u_mean)*k_mean)
```

```
pdf = dist.pdf(x)
mode = x[np.argmax(pdf)]
mean = dist.moment(1)
plt.plot(x, pdf, lw=3, label=f'mode = {mode:.2f}\nmean = {mean:.2f}')
plt.yticks([])

plt.legend()
plt.xlabel('$\theta_{prior}$')
plt.tight_layout()
```

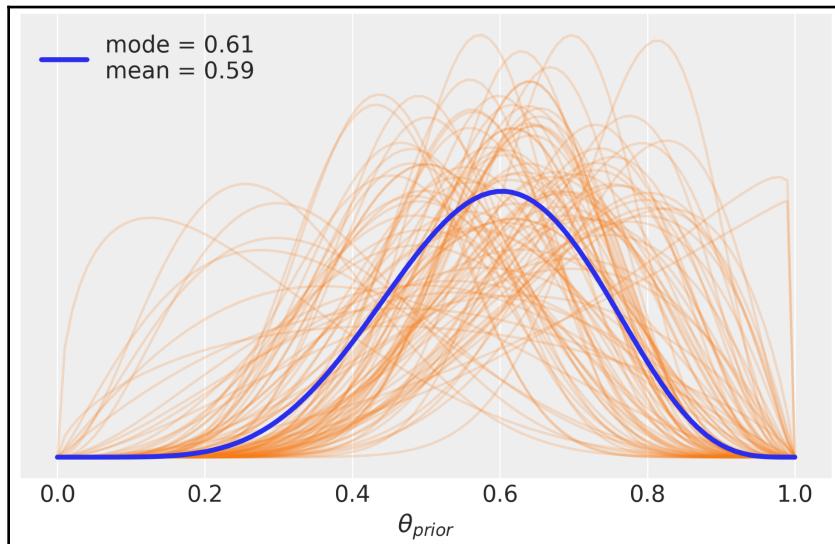


Figure 2.21

This was supposed to be the last example in this chapter, but by popular demand, I will play one more hierarchical model as an *encore*, so please join me.

One more example

Once again, we have chemical shift data. This data comes from a set of protein molecules I have personally prepared. To be precise, we should say the chemical shifts are from the

$^{13}C_\alpha$ nuclei of proteins as this is an observable we measure only for certain types of atomic nuclei. Proteins are made from sequences of 20 building blocks known as amino acidic residues. Each amino acid can appear in the sequence zero or more times, and sequences can vary from a few amino acids to hundreds or even thousands of them. Each amino acid has one and only one $^{13}C_\alpha$, so we can confidently associate each chemical shift to a particular amino acidic residue in a protein. Furthermore, each one of these 20 amino acids has different chemical properties that contribute to the biological properties of the protein; some can have net electric charges and some can only be neutral; some like to be surrounded by water molecules while others prefer the company of the same type or similar type of amino acids; and so on. The key aspect is that they are similar but not equal, and hence it may sound reasonable or even *natural* to base any chemical shift-related inference in terms of 20 groups, as defined by the amino acids types. You can learn more about proteins with this excellent video: <https://www.youtube.com/watch?v=wvTv8TqWC48>.

For the sake of this example, I am simplifying things a little bit here. In practice, experiments are messy, and there is always a chance that we do not get a complete record of chemical shifts. One common problem is signal overlapping, that is, the experiment does not have enough resolution to distinguish two or more close signals. For this example, I just removed those cases, so we will assume that the dataset is complete.

In the following code block, we load the data into a DataFrame; please take a moment to inspect the DataFrame. You will see four columns: the first one is the ID of a protein—if you feel curious, you can access a lot of information about a protein by using the ID in this page: <https://www.rcsb.org/>. The second column includes the name of the amino acid, using a standard three-letter code, while the following columns correspond to theoretical computed chemical shift values (using quantum chemical computations) and the experimentally measured chemical shifts. The motivation for this example is to compare the differences to access, among other reasons, how well the theoretical computations are reproducing the experimental measures. For that reason, we are computing the pandas' series diff:

```
cs_data = pd.read_csv('..../data/chemical_shifts_theo_exp.csv')
diff = cs_data.theo.values - cs_data.exp.values
idx = pd.Categorical(cs_data['aa']).codes
groups = len(np.unique(idx))
```

To see the difference between a hierarchical and non-hierarchical model, we are going to build two models. The first one is basically the same as the `comparing_groups` model:

```
with pm.Model() as cs_nh:  
    μ = pm.Normal('μ', mu=0, sd=10, shape=groups)  
    σ = pm.HalfNormal('σ', sd=10, shape=groups)  
  
    y = pm.Normal('y', mu=μ[idx], sd=σ[idx], observed=diff)  
  
trace_cs_nh = pm.sample(1000)
```

Now, we will build the hierarchical version of the model. We are adding two hyper-priors: one for the mean of μ and one for the standard deviation of μ . We are leaving σ without hyper-priors. This is just a model choice; I am deciding on a simpler model just for pedagogical purposes. You may face a problem where this seems unacceptable and you may consider it necessary to add a hyper-prior for σ ; feel free to do that:

```
with pm.Model() as cs_h:  
    # hyper_priors  
    μ_μ = pm.Normal('μ_μ', mu=0, sd=10)  
    σ_μ = pm.HalfNormal('σ_μ', 10)  
  
    # priors  
    μ = pm.Normal('μ', mu=μ_μ, sd=σ_μ, shape=groups)  
    σ = pm.HalfNormal('σ', sd=10, shape=groups)  
  
    y = pm.Normal('y', mu=μ[idx], sd=σ[idx], observed=diff)  
  
trace_cs_h = pm.sample(1000)
```

We are going to compare the results using ArviZ's `plot_forest` function. We can pass more than one model to this function. This is useful when we want to compare the values of parameters from different models such as with the present example. Notice that we are passing several arguments to `plot_forest` to get the plot that we want, like `combined=True` to merge results from all the chains. I invite you to explore the rest of the arguments:

```
_ , axes = az.plot_forest([trace_cs_nh, trace_cs_h],
                         model_names=['n_h', 'h'],
                         var_names='μ', combined=False, colors='cycle')
y_lims = axes[0].get_ylim()
axes[0].vlines(trace_cs_h['μ_μ'].mean(), *y_lims)
```

OK, so what we have in *Figure 2.22*? We have a plot for the 40 estimated means, one per amino acid (20) multiplied by two as we have two models. We also have their 94% credible intervals and the interquartile range (the central 50% of the distribution). The vertical (black) line is the global mean according to the hierarchical model. This value is close to zero, as expected for theoretical values reproducing experimental ones.

The most relevant part of this plot is that the estimates from the hierarchical model are pulled toward the partially-pooled mean, or equivalently they are shrunken with respect to the un-pooled estimates. You will also notice that the effect is more notorious for those groups farther away from the mean (such as 13) and that the uncertainty is on par or smaller than that from the non-hierarchical model. The estimates are partially pooled because we have one estimate for each group, but estimates for individual groups restrict each other through the hyper-prior.

Therefore, we get an intermediate situation between having a single group, all chemical shifts together, and having 20 separated groups, one per amino acid. And that is, ladies, gentlemen and non-binary-gender-fluid people, the beauty of hierarchical models:

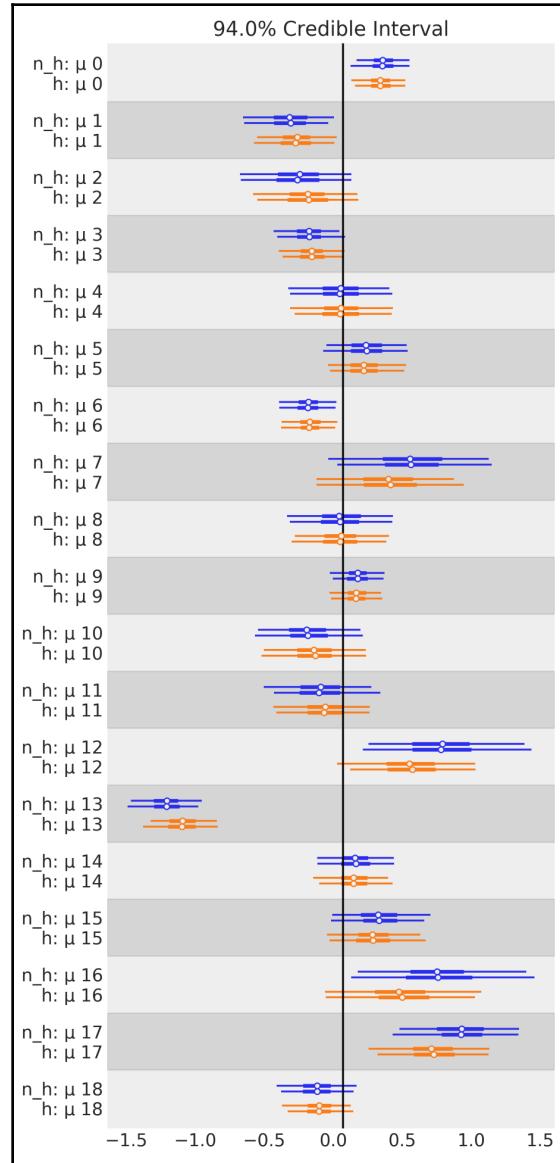


Figure 2.22

Paraphrasing the Zen of Python, we can certainly say, *hierarchical models are one honking great idea—let's do more of those!* In the following chapters, we will keep building hierarchical models and learn how to use them to build better models. We will also discuss how hierarchical models are related to the pervasive overfitting/underfitting issue in statistics and machine learning in Chapter 5, *Model Comparison*. In Chapter 8, *Inference Engines*, we will discuss some technical problems that we may find when sampling from hierarchical models and how to diagnose and fix those problems.

Summary

Although Bayesian statistics is conceptually simple, fully probabilistic models often lead to analytically intractable expressions. For many years, this was a huge barrier, hindering the wide adoption of Bayesian methods. Fortunately, math, statistics, physics, and computer science came to the rescue in the form of numerical methods that are capable—at least in principle—of solving any inference problem. The possibility of automating the inference process has led to the development of probabilistic programming languages, allowing for a clear separation between model definition and inference.

PyMC3 is a Python library for probabilistic programming with a very simple, intuitive, and easy to read syntax that is also very close to the statistical syntax used to describe probabilistic models. We introduced the PyMC3 library by revisiting the coin-flip model from Chapter 1, *Thinking Probabilistically*, this time without analytically deriving the posterior. PyMC3 models are defined inside a context manager. To add a probability distribution to a model, we just need to write a single line of code. Distributions can be combined and can be used as priors (unobserved variables) or likelihoods (observed variables). If we pass data to a distribution, it becomes a *likelihood*. Sampling can be achieved with a single line as well. PyMC3 allows us to get samples from the posterior distribution. If everything goes right, these samples will be representative of the correct posterior distribution and thus they will be a representation of the logical consequences of our model and data.

We can explore the posterior generated by PyMC3 using ArviZ, a Python library that works hand-in-hand with PyMC3 and can be used, among other tasks, to help us interpret and visualize posterior distributions. One way of using a posterior to help us make inference-driven decisions is by comparing the ROPE against the HPD interval. We also briefly mentioned the notion of loss functions, a formal way to quantify the trade-offs and costs associated to making decisions in the presence of uncertainty. We learned that loss functions and point-estimates are intimately associated.

Up to this point, the discussion was restricted to a simple one-parameter model. Generalizing to arbitrary number of parameters is trivial with PyMC3; we exemplify how to do this with the Gaussian and Student's t models. The Gaussian distribution is a special case of the Student's t-distribution and we showed you how to use the latter to perform robust inferences in the presence of outliers. In the next chapter, we will look at how these model can be used as part of linear regression models.

We used a Gaussian model to compare a common data analysis task among groups. While this is sometimes framed in the context of hypothesis testing, we take another route and frame this task as a problem of inferring the effect size, an approach we generally consider to be richer and more productive. We also explored different ways to interpret and report effect sizes.

We saved for last, as we usually do with a fine dessert, one of the most important concepts to learn from this book: **hierarchical models**. We can build hierarchical models every time we can identify subgroups in our data. In such cases, instead of treating the subgroups as separated entities or ignoring the subgroups and treating them as a single group, we can build a model to partially pool information among groups. The main effect from this **partial pooling** is that the estimates of each subgroup will be biased by the estimates of the rest of the subgroups. This effect is known as **shrinkage**, and in general is a very useful trick that helps to improve inferences by making them more conservative (as each subgroup inform the others by pulling estimates toward it) and more informative. We get estimates at the subgroup level and the group level. We will see more examples of hierarchical models in the following chapters. Each example will help us better understand them from a slightly different perspective.

Exercises

1. Using PyMC3, change the parameters of the prior beta distribution in `our_first_model` to match those of the previous chapter. Compare the results to the previous chapter. Replace the beta distribution with a uniform one in the interval [0,1]. Are the results equivalent to the $\text{Beta}(\alpha = 1, \beta = 1)$? Is the sampling slower, faster, or the same? What about using a larger interval such as [-1, 2]? Does the model run? What errors do you get?
2. Read about the coal mining disaster model that is part of the PyMC3 documentation: http://pymc-devs.github.io/pymc3/notebooks/getting_started.html#Case-study-2:-Coal-mining-disasters. Try to implement and run this model by yourself.

3. Modify `model_g`, change the prior for the mean to a Gaussian distribution centered at the empirical mean, and play with a couple of *reasonable* values for the standard deviation of this prior. How robust/sensitive are the inferences to these changes? What do you think of using a Gaussian, which is an unbounded distribution (goes from $-\infty$ to ∞), to model bounded data such as this? Remember that we said it is not possible to get values below 0 or above 100.
4. Using the data in the `chemical_shifts.csv` file, compute the empirical mean and the standard deviation with and without outliers. Compare those results to the Bayesian estimation using the Gaussian and Student's t-distribution. Repeat the exercise by adding more outliers.
5. Modify the tips example to make it robust to outliers. Try with one shared ν for all groups and also with one ν per group. Run posterior predictive checks to assess these three models.
6. Compute the probability of superiority directly from the posterior (without computing Cohen's d first). You can use the `pm.sample_posterior_predictive()` function to take a sample from each group. Is it really different from the calculation assuming normality? Can you explain the result?
7. Repeat the exercise we did with `model_h`. This time, without hierarchical structure, use a flat prior such as $\text{Beta}(\alpha = 1, \beta = 1)$. Compare the results of both models.
8. Create a hierarchical version of the tips example by partially pooling across the days of the week. Compare the results to those obtained without the hierarchical structure.
9. PyMC3 can create **directed acyclic graphs (DAGs)** from models that are very similar to Kruschke's diagrams. You can obtain them using the `pm.model_to_graphviz()` function. Generate a DAG for each model in this chapter.

Besides the exercises you will find at the end of each chapter, you can always try to (and probably should) think of problems you are interested in and how to apply what you have learned to that problem. Maybe you will need to define your problem in a different way, or maybe you will need to expand or modify the models you have learned. If you think this task is beyond your actual skills, note down the problem and come back to it after reading another chapter in this book. Eventually, if the book does not answer your questions, check the PyMC3 examples (<http://pymc-devs.github.io/pymc3/examples.html>) or ask a question at <https://discourse.pymc.io/>.

3

Modeling with Linear Regression

"In more than three centuries of science everything has changed except perhaps one thing: the love for the simple."

—Jorge Wagensberg

Music—from classical compositions to *Sheena is a Punk Rocker* by The Ramones, passing through the unrecognized *hit* from a garage band and Piazzolla's *Libertango*—is made from recurring patterns. The same scales, combinations of chords, riffs, motifs, and so on appear over and over again, giving rise to a wonderful sonic landscape capable of eliciting and modulating the entire range of emotions humans can experience. In a similar fashion, the universe of statistics and **machine learning (ML)** is built upon recurring patterns, small motifs that appear now and again. In this chapter, we are going to look at one of the most popular and useful of them, the **linear model** (or motif, if you want). This is a very useful model on its own and also the building block of many other models. If you ever took a statistics course (even a non-Bayesian one), you may have heard of simple and multiple linear regression, logistic regression, ANOVA, ANCOVA, and so on. All these methods are variations of the same underlying motif, the linear regression model. In this chapter, we will cover the following topics:

- Simple linear regression
- Robust linear regression
- Hierarchical linear regression
- Polynomial regression
- Multiple linear regression
- Interactions
- Variable variance

Simple linear regression

Many problems we find in science, engineering, and business are of the following form. We have a variable x and we want to model/predict a variable y . Importantly, these variables are paired like $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots (x_n, y_n)\}$. In the most simple scenario, known as **simple linear regression**, both x and y are uni-dimensional continuous random variables. By continuous, we mean a variable represented using real numbers (or floats, if you wish), and using NumPy, you will represent the variables x or y as one-dimensional arrays. Because this is a very common model, the variables get proper names. We call the y variables the **dependent, predicted, or outcome** variables, and the x variables the **independent, predictor, or input** variables. When X is a matrix (we have different variables), we have what is known as **multiple linear regression**. In this and the following chapter, we will explore these and other linear regression models.

Some typical situations where linear regression models can be used are:

- Model the relationship between factors like rain, soil salinity, and the presence/absence of fertilizer in crop productivity. Then, answer questions such as: is the relationship linear? How strong is this relationship? Which factors have the strongest effect?
- Find a relationship between average chocolate consumption by country and the number of Nobel laureates in that country, and then understand why this relationship could be spurious.
- Predict the gas bill (used for heating and cooking) of your house by using the sun radiation from the local weather report. How accurate is this prediction?

The machine learning connection

Paraphrasing Kevin P. Murphy, machine learning is an umbrella term for a collection of methods to automatically learn patterns in data, and then use what we learn to predict future data or to take decisions under uncertainty. ML and statistics are really intertwined subjects, and the connection becomes clear if you took a probabilistic perspective, as Kevin Murphy does in his great book *Machine learning: A probabilistic perspective*. While these domains are deeply connected at a conceptual and mathematical level, the jargon could make the connection opaque. So, let me bring the ML vocabulary to the problem in this chapter. Using the ML terminology, we say a regression problem is an example of **supervised learning**. Under the machine learning framework, we have a regression problem when we want to learn a mapping from X to Y , with Y being a continuous variable.

A machine learner usually talks about **features** instead of variables. We say that the learning process is supervised because we know the values of the X - Y pairs; in some sense, we know the correct answer, and all the remaining questions are about how to generalize these observations (or this dataset) to any possible future observation, that is, to a situation when we know X but not Y .

The core of the linear regression models

Now that we have discussed some general ideas about linear regression and we have also established a bridge between the vocabulary used in statistics and ML, let's begin to learn how we can build linear models.

The chances are high that you are already familiar with the following equation:

$$y_i = \alpha + x_i \beta \quad (3.1)$$

This equation says that there is a linear relation between the variable x and the variable y . The parameter β controls the **slope** of the linear relationship and thus is interpreted as the change in the variable y per unit change in the variable x . The other parameter, α , is known as the **intercept**, and tells us the value of y_i when $x_i = 0$. Graphically, the intercept is the value y_i of the point where the line intercepts the y axis.

There are several ways to find the parameters for a linear model; one method is known as **least squares fitting**. Least squares returns the values of α and β yielding the lowest average quadratic error between the observed y and the predicted y . Expressed in that way, the problem of estimating α and β is an optimization problem, that is, a problem where we try to find the *minima* or *maxima* of some function. An alternative to optimization is to generate a fully probabilistic model. Thinking probabilistically gives us several advantages; we can obtain the best values of α and β (the same as with optimization methods) together with an estimation of the uncertainty we have about the parameter's values. Optimization methods require extra work to provide this information. Additionally, the probabilistic approach, especially when done using tools such as PyMC3, will give us the flexibility to adapt models to particular problems, as we will see during this chapter.

Probabilistically, a linear regression model can be expressed as follows:

$$y \sim \mathcal{N}(\mu = \alpha + x\beta, \epsilon) \quad (3.2)$$

That is, the data vector y is assumed to be distributed as a Gaussian with a mean of $\alpha + \beta x$ and with a standard deviation of ϵ .



A linear regression model is an extension of the Gaussian model where the mean is not directly estimated but rather computed as a linear function of a predictor variable and some additional parameters.

Since we do not know the values of α , β , or ϵ , we have to set prior distributions for them. A reasonable generic choice would be:

$$\begin{aligned}\alpha &\sim \mathcal{N}(\mu_\alpha, \sigma_\alpha) \\ \beta &\sim \mathcal{N}(\mu_\beta, \sigma_\beta) \\ \epsilon &\sim |N(0, \sigma_\epsilon)|\end{aligned}\tag{3.3}$$

For the prior over α , we can use a very *flat* Gaussian by setting the value σ_α to a relatively high value for the scale of the data. In general, we do not know where the intercept can be, and its value can vary a lot from one problem to another and for different domain knowledge. For many problems I have worked on, α is usually centered around zero and with a σ_α no larger than 10, but this is just my experience (almost anecdotal) with a small subset of problems and not something easy to transfer to other problems. Regarding the slope, it may be easier to have a general idea of what to expect than for the intercept. For many problems, we can at least know the sign of the slope *a priori*; for example, we expect the variable weight to increase, on average, with the variable height. For epsilon, we can set σ_ϵ to a large value on the scale of the variable y , for example, ten times the value for its standard deviation. These very vague priors guarantee a very small effect of the prior on the posterior, which is easily overcome by the data.



The point-estimate obtained using the least squares method will agree with the **maximum a posteriori (MAP)** (the mode of the posterior) from a Bayesian simple linear regression with flat priors.

A couple of alternatives to the Half-Gaussian distribution for ϵ are the Uniform or the half-Cauchy distributions. The half-Cauchy distribution generally works well as a good regularizing prior (see chapter 6, *Model Comparison*, for details) and the Uniform distributions are generally not a very good choice unless you know that the parameter is truly restricted by hard boundaries. If we want to use really strong priors around some specific value for the standard deviation, we can use the gamma distribution. The default parameterization of the gamma distribution in many packages can be a little bit confusing at first, but fortunately PyMC3 allows us to define it using the shape and rate (probably the most common parameterization) or the mean and standard deviation (probably a more intuitive parameterization, at least for newcomers).

To see how the gamma and other distributions look like, you can check out the PyMC3 documentation here: <https://docs.pymc.io/api/distributions/continuous.html>.

Going back to the linear regression model, we can use the nice and easy to interpret Kruschke diagrams to represent them, as in *Figure 3.1*. You may remember from the previous chapter that in Kruschke diagrams we use the symbol $=$ to define deterministic variables (such as μ) and \sim to define stochastic variables such as α, β , and ϵ :

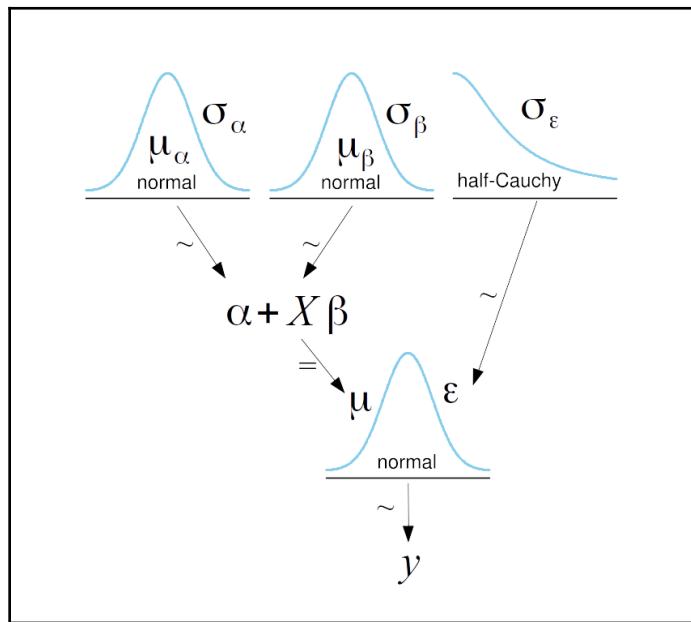


Figure 3.1

Now that we have defined the model, we need the data to feed the model. Once again, we are going to rely on a synthetic dataset to build intuition on the model. One advantage of a synthetic dataset is that we know the correct values of the parameters and we can check if we are able to recover them with our models:

```
np.random.seed(1)
N = 100
alpha_real = 2.5
beta_real = 0.9
eps_real = np.random.normal(0, 0.5, size=N)

x = np.random.normal(10, 1, N)
y_real = alpha_real + beta_real * x
```

```

y = y_real + eps_real

_, ax = plt.subplots(1,2, figsize=(8, 4))
ax[0].plot(x, y, 'C0.')
ax[0].set_xlabel('x')
ax[0].set_ylabel('y', rotation=0)
ax[0].plot(x, y_real, 'k')
az.plot_kde(y, ax=ax[1])
ax[1].set_xlabel('y')
plt.tight_layout()

```

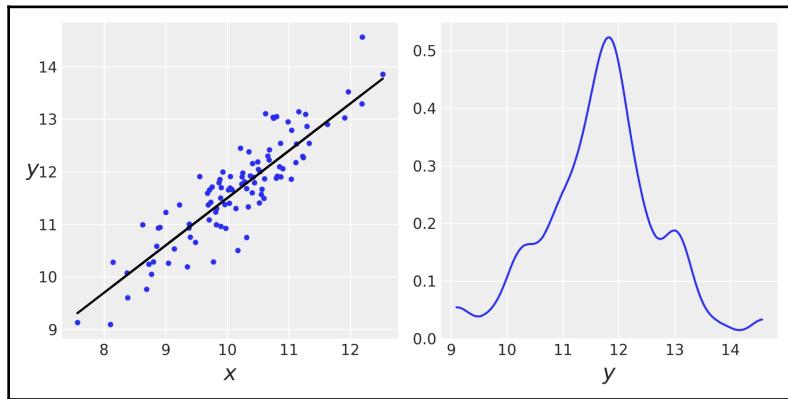


Figure 3.2

Now, we will use PyMC3 to build and fit the model—nothing we haven't seen before. Wait—in fact, there is something new! μ is expressed in the model as a deterministic variable, reflecting what we have already written in mathematical notation and in the Kruschke's diagram. If we specify a PyMC3 deterministic variable, we are telling PyMC3 to compute and save the variable in the trace:

```

with pm.Model() as model_g:
    a = pm.Normal('a', mu=0, sd=10)
    b = pm.Normal('b', mu=0, sd=1)
    epsilon = pm.HalfCauchy('epsilon', 5)

    mu = pm.Deterministic('mu', a + b * x)
    y_pred = pm.Normal('y_pred', mu=mu, sd=epsilon, observed=y)

    trace_g = pm.sample(2000, tune=1000)

```

Alternatively, instead of including a deterministic variable in the model, we can omit it. In such a case, the variable will still be computed but not saved in the trace. For example, we could have written the following:

```
y_pred = pm.Normal('y_pred', mu= α + β * x, sd=ε, observed=y)
```

To explore the results of our inference, we are going to generate a trace plot, omitting the deterministic variable μ . We can do this by passing the names of the variables we want to include in the plot as a list to the `var_names` argument. Many ArviZ functions have a `var_names` argument:

```
az.plot_trace(trace_g, var_names=['α', 'β', 'ε'])
```

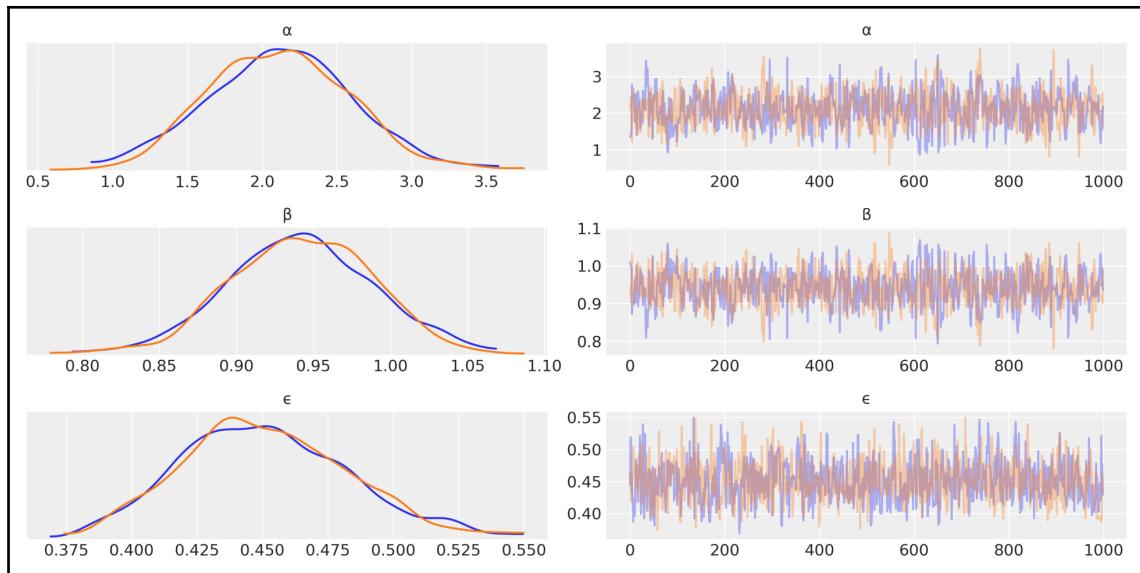


Figure 3.3

Feel free to experiment with other ArviZ plots to explore the posterior. In the next section, we are going to discuss a property of the linear model and how it can affect the sampling process and model interpretation. Then, we are going to look at a few ways to interpret and visualize the posterior.

Linear models and high autocorrelation

Linear models lead to posterior distribution where α and β are highly correlated. See the following code and *Figure 3.4* for an example:

```
az.plot_pair(trace_g, var_names=[' $\alpha$ ', ' $\beta$ '], plot_kwarg={'alpha': 0.1})
```

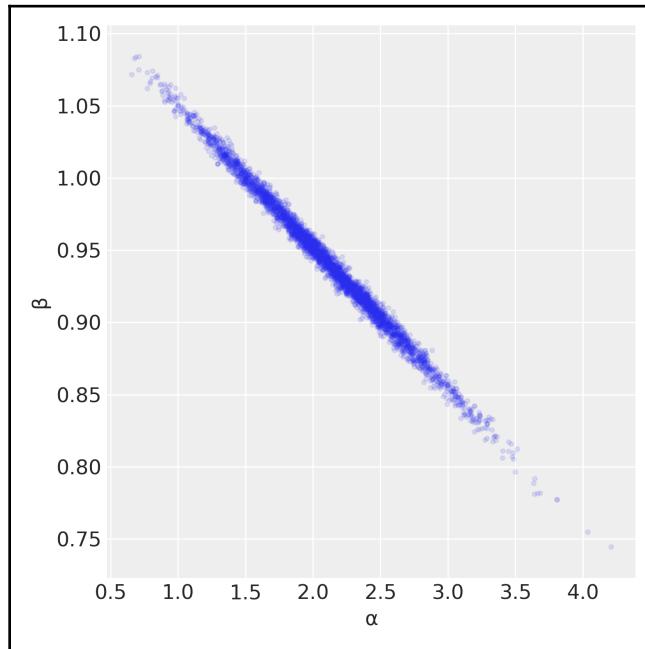


Figure 3.4

The correlation we are seeing in *Figure 3.4* is a direct consequence of our assumptions. No matter which line we fit to our data, all of them should pass for one point, that is, the mean of the x variable and the mean of the y variable. Hence, the line fitting process is somehow equivalent to spinning a straight line fixed at the center of the data, like a wheel of fortune. An increase in the slope means a decrease of the intercept and *vice versa*. Both parameters are going to be correlated by definition of the model. Hence, the shape of the posterior (excluding ϵ) is a very diagonal space. This can be problematic for samplers such as Metropolis-Hastings and to a lesser extent NUTS. For details of why this is true, see Chapter 8, *Inference Engines*.

Before continuing and for the sake of truth, let me clarify something. The fact that the line is constrained to pass through the mean of the data is only true for the least square method (and its assumptions). Using Bayesian methods, this constraint is relaxed. Later in the examples, we will see that, in general, we get lines *around* the mean values of x and y and not exactly through the exact mean. Moreover, if we use strong priors, we could end up with lines far away from the mean of x and y . Nevertheless, the idea that the autocorrelation is related to the line spinning around a more or less defined point remains true, and that is all we need to understand regarding the correlation of the α and β parameters.

Modifying the data before running

One simple way to remove the correlation of α and β is to center the x variable. For each x_i data point, we subtract the mean of the x variable (\bar{x}):

$$x' = x - \bar{x} \quad (3.4)$$

As a result, x' will be centered at 0, and hence the *pivot point* when changing the slope is exactly the intercept, and thus the plausible parameter space is now more *circular* and less correlated. Be sure to complete exercise 6 (in the *Exercises* section) to see the differences between centering and not centering the data.

Centering data is not only a computational trick; it can also help in interpreting the results. The intercept is the value of y_i when $x_i = 0$. For many problems, this interpretation has no real meaning. For example, for quantities such as the height or weight, values of zero are meaningless and hence the intercept has no value in helping to make sense of the data. Instead, when centering the variables, the intercept is always the value of y_i for the mean value of x .

For some problems, it may be useful to estimate the intercept precisely because it is not feasible to experimentally measure the value of $x_i = 0$ and so the estimated intercept can provide us with valuable information. However, extrapolations can be problematic, so be careful when doing this!

We may want to report the estimated parameters in terms of the centered data or in terms of the uncentered data, depending on our problem and audience. If we need to report the parameters as if they were determined in the original scale, we can do the following to put them back into the original scale:

$$\alpha = \alpha' - \beta' \bar{x} \quad (3.5)$$

This correction is the result of the following algebraic reasoning:

$$\begin{aligned}y &= \alpha' + \beta' x' + \epsilon \\y &= \alpha' + \beta' (x - \bar{x}) + \epsilon \\y &= \alpha' - \beta' \bar{x} + \beta' x + \epsilon\end{aligned}\tag{3.6}$$

Therefore, it follows that equation 3.5 is true and also that:

$$\beta = \beta'\tag{3.7}$$

We can even go further than centering x and transforming the data by **standardizing** it before running models. Standardizing is a common practice for linear regression models in statistics and ML since many algorithms behave better when the data is standardized. This transformation is achieved by centering the data and dividing it by the standard deviation. Mathematically we have:

$$\begin{aligned}x' &= \frac{x - \bar{x}}{x_{sd}} \\y' &= \frac{y - \bar{y}}{y_{sd}}\end{aligned}\tag{3.8}$$

One advantage of standardizing the data is that we can always use the same weakly informative priors without having to think about the scale of the data. For standardized data, the intercept will always be around 0 and the slope will be restricted to the interval [-1, 1]. Additionally, standardizing the data allow us to talk in terms of Z-scores, that is, in units of standard deviations. If someone says the value of a parameter is -1.3 in Z-score units, we automatically know that the value in question is 1.3 standard deviations below the mean, irrespective of the actual value of the mean or the actual value of the standard deviation of the data. A change in one Z-score unit is a change in one standard deviation, whatever the scale of the original data is. This can be very useful when working with several variables; having all of the variables in the same scale can simplify the interpretation of the data.

Interpreting and visualizing the posterior

As we have already seen, we can explore the posterior using ArviZ functions like `plot_trace` and `summary`, or we can use our own functions. For a linear regression, it could be useful to plot the average line that fits the data, together with the average mean values of α and β . To reflect the posterior's uncertainty, we can use semitransparent lines that have been sampled from the posterior:

```
plt.plot(x, y, 'C0.')

alpha_m = trace_g['α'].mean()
beta_m = trace_g['β'].mean()

draws = range(0, len(trace_g['α']), 10)
plt.plot(x, trace_g['α'][draws] + trace_g['β'][draws]
          * x[:, np.newaxis], c='gray', alpha=0.5)

plt.plot(x, alpha_m + beta_m * x, c='k',
         label=f'y = {alpha_m:.2f} + {beta_m:.2f} * x')

plt.xlabel('x')
plt.ylabel('y', rotation=0)
plt.legend()
```

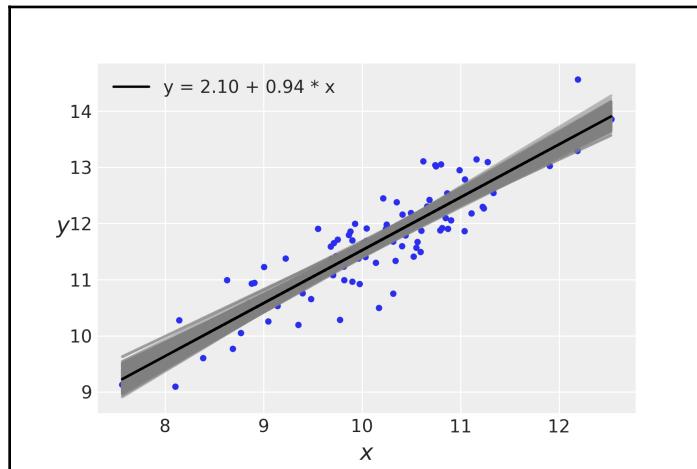


Figure 3.5

Notice that uncertainty is lower in the middle, although it is not reduced to a single point, that is, the posterior is compatible with lines not passing exactly through the mean of the data, as we have already mentioned.

Having the semitransparent lines is perfectly fine, but we may want to add a *cool-factor* to the plot and instead use a semitransparent band to illustrate the **Highest-Posterior Density (HPD)** interval of μ . In fact, this was the main reason we defined the deterministic variable μ in the model. Having this variable simplifies the following code:

```
plt.plot(x, alpha_m + beta_m * x, c='k',
          label=f'y = {alpha_m:.2f} + {beta_m:.2f} * x')
sig = az.plot_hpd(x, trace_g['μ'], credible_interval=0.98, color='k')
plt.xlabel('x')
plt.ylabel('y', rotation=0)
plt.legend()
```

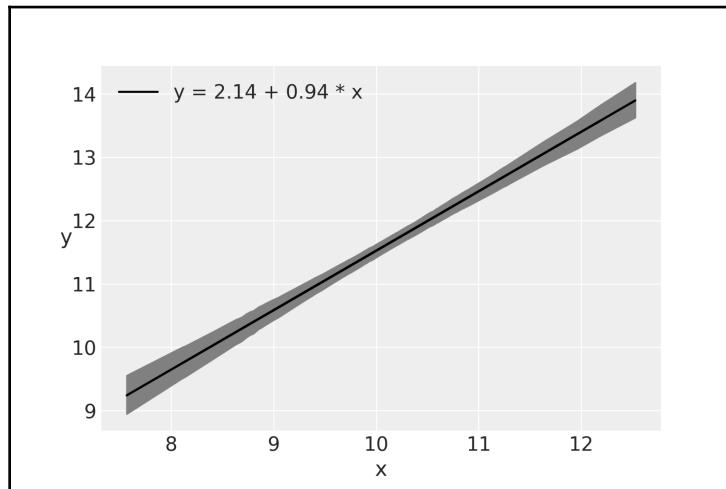


Figure 3.6

One more option is to plot the HPD (for example, 94% and 50%) of the predicted data \hat{y} , that is, where we expect to see the 94% and 50% of future data, according to our model. For *Figure 3.7*, we are going to use a darker gray for the HPD 50 and a lighter gray for the HPD 95.

Getting the posterior predictive samples is easy in PyMC3 using the `sample_posterior_predictive()` function:

```
ppc = pm.sample_posterior_predictive(trace_g,
                                      samples=2000,
                                      model=model_g)
```

Now, we can plot the results:

```
plt.plot(x, y, 'b.')
plt.plot(x, alpha_m + beta_m * x, c='k',
         label=f'y = {alpha_m:.2f} + {beta_m:.2f} * x')

az.plot_hpd(x, ppc['y_pred'], credible_interval=0.5, color='gray')
az.plot_hpd(x, ppc['y_pred'], color='gray')

plt.xlabel('x')
plt.ylabel('y', rotation=0)
```

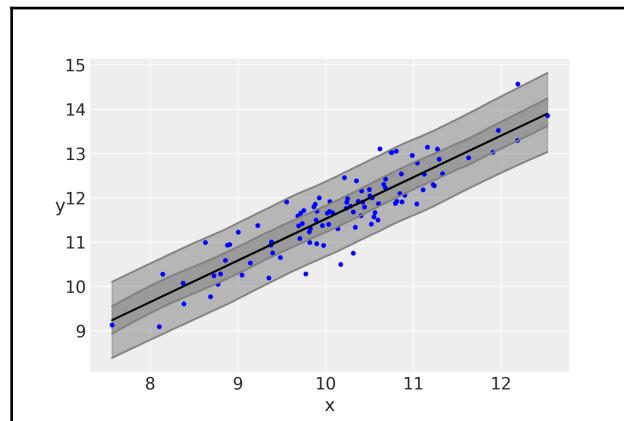


Figure 3.7

The function `az.plot_hpd` is a helper function that we can use to plot a HPD interval for linear regressions. By default this function smooth the interval. Try passing the argument `smooth=False` to see what I mean.

Pearson correlation coefficient

Sometimes, we want to measure the degree of (linear) dependence between two variables. The most common measure of the linear correlation between two variables is the **Pearson correlation coefficient**, often identified with a lowercase r . When $r = +1$, we have a perfect positive linear correlation, that is, an increase of one variable predicts an increase of the other. When we have $r = -1$, we have a perfect negative linear correlation and the increase of one variable predicts a decrease of the other. When $r = 0$, we have no linear correlation. As a general rule, we will get intermediate values. It's important to always have two aspects in mind: the Pearson correlation coefficient says nothing about non-linear correlations. We should not confuse r with the slope of a regression. The following image from Wikipedia is a good one to have at hand:

https://en.wikipedia.org/wiki/Correlation_and_dependence#/media/File:Correlation_examples2.svg.

Part of the confusion between r and the slope of a regression may be explained by the following relationship:

$$r = \beta \frac{\sigma_x}{\sigma_y} \tag{3.9}$$

That is, the slope (β) and the Pearson correlation coefficient (r) have the same value, but only when the standard deviation of x and y are equal. Notice that it is true, for example, when we standardize the data. To further clarify:

- The Pearson correlation coefficient (r) is a measure of the degree of correlation between two variables and is always restricted to the interval $[-1, 1]$, regardless of the scale of the data.
- The slope of a linear regression (β) indicates how much y changes per unit change of x , and can take any real value.

The Pearson coefficient is related to a quantity known as the **determination coefficient**, and, for a linear regression model, this is just the square of the Pearson coefficient, that is, r^2 (or sometimes R^2). This is pronounced as *r squared* and can be defined as the variance of the predicted values divided by the variance of the data. Therefore, it can be interpreted as the proportion of the variance in the dependent variable that is predicted from the independent variable. For Bayesian linear regression, the variance of the predicted values can be larger than the variance of the data and this will lead to an R^2 larger than 1, then a good solution is to define R^2 as follows:

$$R^2 = \frac{\mathbf{V}_{n=1}^N \mathbf{E}[\hat{y}^s]}{\mathbf{V}_{n=1}^N \mathbf{E}[\hat{y}^s] + \mathbf{V}_{n=1}^S (\hat{y}^s - y)} \quad (3.10)$$

In the above equation, $\mathbf{E}[\hat{y}^s]$ is the expected (or mean) value \hat{y} over S posterior samples.

This is the variance of the predicted values divided by the variance of the predicted values plus the errors (or residuals). This definition has the advantage of ensuring R^2 is restricted to the interval $[0, 1]$.

The easiest way to compute R^2 is to use the `r2_score()` function from ArviZ. We need the observed values of y and the predicted values of \hat{y} . Remember that we can get \hat{y} from `sample_posterior_predictive()`:

```
az.r2_score(y, ppc['y_pred'])
```

By default, this function will return R^2 (0.8, for this example) and the standard deviation (0.03).

Pearson coefficient from a multivariate Gaussian

Another way to compute the Pearson coefficient is by estimating the covariance matrix of a multivariate Gaussian distribution. A multivariate Gaussian distribution is the generalization of the Gaussian distribution to more than one dimension. Let's focus on the case of two dimensions because that is what we are going to use right now. Generalizing to higher dimensions is almost trivial once we understand the bivariate case. To fully describe a bivariate Gaussian distribution, we need two means (or a vector with two elements), one per each marginal Gaussian. We also need two standard deviations, right? Well, not exactly; we need a 2×2 covariance matrix, which looks like this:

$$\Sigma = \begin{bmatrix} \sigma_{x_1}^2 & \rho\sigma_{x_1}\sigma_{x_2} \\ \rho\sigma_{x_1}\sigma_{x_2} & \sigma_{x_2}^2 \end{bmatrix} \quad (3.11)$$

Here, Σ is the Greek capital sigma letter and it is common practice to use it to represent the covariance matrix. In the main diagonal, we have the variances of each variable, expressed as the square of their standard deviations σ_{x_1} and σ_{x_2} . The rest of the elements in the matrix are the covariances (the variance between variables), which are expressed in terms of the individual standard deviations and ρ , the Pearson correlation coefficient between variables. Notice that we have a single ρ because we have only two dimensions (or variables). For three variables, we would have three Pearson coefficients.

The following code generates contour plots for bivariate Gaussian distributions with both means fixed at $(0, 0)$. One of the standard deviations is fixed $\sigma_{x_1} = 1$, while the other takes the values 1 or 2 and different values for the Pearson correlation coefficient:

```

sigma_x1 = 1
sigmas_x2 = [1, 2]
rhos = [-0.90, -0.5, 0, 0.5, 0.90]

k, l = np.mgrid[-5:5:.1, -5:5:.1]
pos = np.empty(k.shape + (2,))
pos[:, :, 0] = k
pos[:, :, 1] = l

f, ax = plt.subplots(len(sigmas_x2), len(rhos),
                     sharex=True, sharey=True, figsize=(12, 6),
                     constrained_layout=True)
for i in range(2):
    for j in range(5):
        sigma_x2 = sigmas_x2[i]
        rho = rhos[j]
        cov = [[sigma_x1**2, sigma_x1*sigma_x2*rho],
               [sigma_x1*sigma_x2*rho, sigma_x2**2]]
        rv = stats.multivariate_normal([0, 0], cov)
        ax[i, j].contour(k, l, rv.pdf(pos))
        ax[i, j].set_xlim(-8, 8)
        ax[i, j].set_ylim(-8, 8)
        ax[i, j].set_yticks([-5, 0, 5])
        ax[i, j].plot(0, 0,
                      label=f'$\sigma_{x_2}$ = {sigma_x2:3.2f}\n$\rho$ = {rho:3.2f}', alpha=0)
        ax[i, j].legend()
f.text(0.5, -0.05, 'x_1', ha='center', fontsize=18)
f.text(-0.05, 0.5, 'x_2', va='center', fontsize=18, rotation=0)

```

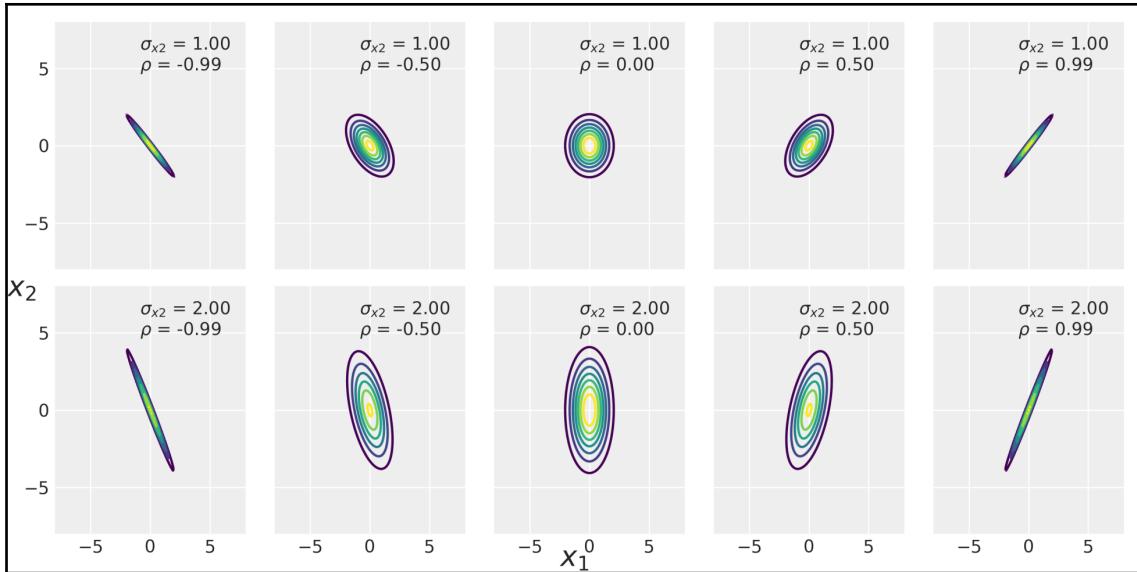


Figure 3.8

Now that we know the multivariate Gaussian distribution, we can use it to estimate the Pearson correlation coefficient. Since we do not know the values of the covariance matrix, we have to put priors over it. One solution is to use the Wishart distribution, which is the conjugate prior of the inverse covariance matrix of a multivariate normal. The Wishart distribution can be considered as the generalization to higher dimensions of the gamma distribution we saw earlier or also the generalization of the χ^2 distribution. A second option is to use the LKJ prior (see <https://docs.pymc.io/notebooks/LKJ.html> for details). This is a prior for the correlation matrix (and not the covariance matrix), which may be convenient, given that it is generally more useful to think in terms of correlations. We are going to explore a third option and we are going to put priors directly for σ_{x_1} , σ_{x_2} , and ρ , and then use those values to manually build the covariance matrix:

```
data = np.stack((x, y)).T
with pm.Model() as pearson_model:

    mu = pm.Normal('mu', mu=data.mean(0), sd=10, shape=2)

    sigma_1 = pm.HalfNormal('sigma_1', 10)
    sigma_2 = pm.HalfNormal('sigma_2', 10)
    rho = pm.Uniform('rho', -1., 1.)
    r2 = pm.Deterministic('r2', rho**2)

    cov = pm.math.stack(([sigma_1**2, sigma_1 * sigma_2 * rho],
                         [sigma_1 * sigma_2 * rho, sigma_2**2]))
```

```
[σ_1*σ_2*p, σ_2**2]))  
y_pred = pm.MvNormal('y_pred', mu=μ, cov=cov, observed=data)  
trace_p = pm.sample(1000)
```

We are going to omit all variables other than r^2 :

```
az.plot_trace(trace_p, var_names=['r2'])
```

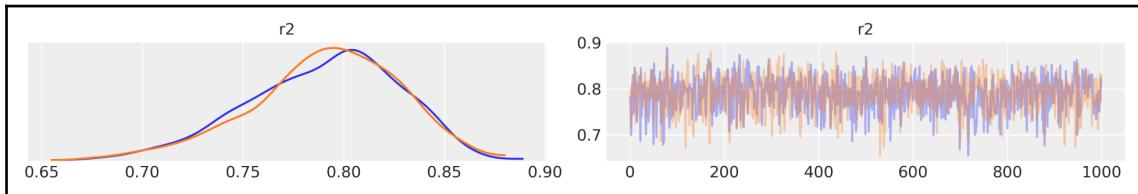


Figure 3.9

We can see that the distribution of r^2 values is around the value we got in the previous example using the `r2_score` function from ArviZ. Maybe an easier way to compare the value obtained from the multivariate Gaussian with the previous result is by using `summary`. As you can see, we got a pretty good match:

```
az.summary(trace_p, var_names=['r2'])
```

	mean	sd	mc error	hpd 3%	hpd 97%	eff_n	r_hat
r2	0.79	0.04	0.0	0.72	0.86	839.0	1.0

Robust linear regression

Assuming that the data follows a Gaussian distribution, it is perfectly reasonable in many situations. By assuming Gaussianity, we are not necessarily saying data is really Gaussian; instead, we are saying that it is a reasonable approximation for a given problem. The same applies to other distributions. As we saw in the previous chapter, sometimes, this Gaussian assumption fails, for example, in the presence of outliers. We learned that using a Student's t-distribution is a way to effectively deal with outliers and get a more robust inference. The very same idea can be applied to linear regression.

To exemplify the robustness that a Student's t-distribution brings to a linear regression, we are going to use a very simple and nice dataset: the third data group from the Anscombe quartet. If you do not know what the **Anscombe quartet** is, remember to check it later at Wikipedia (https://en.wikipedia.org/wiki/Anscombe%27s_quartet). We can upload it using pandas. We are going to center the data, just to make things easier for the sampler—even a cool sampler like NUTS needs a little help from time to time:

```
ans = pd.read_csv('../data/anscombe.csv')
x_3 = ans[ans.group == 'III']['x'].values
y_3 = ans[ans.group == 'III']['y'].values
x_3 = x_3 - x_3.mean()
```

Now, let's check what this little tiny dataset looks like:

```
_ , ax = plt.subplots(1, 2, figsize=(10, 5))
beta_c, alpha_c = stats.linregress(x_3, y_3)[:2]
ax[0].plot(x_3, (alpha_c + beta_c * x_3), 'k',
            label=f'y ={alpha_c:.2f} + {beta_c:.2f} * x')
ax[0].plot(x_3, y_3, 'C0o')
ax[0].set_xlabel('x')
ax[0].set_ylabel('y', rotation=0)
ax[0].legend(loc=0)
az.plot_kde(y_3, ax=ax[1], rug=True)
ax[1].set_xlabel('y')
ax[1].set_yticks([])
plt.tight_layout()
```

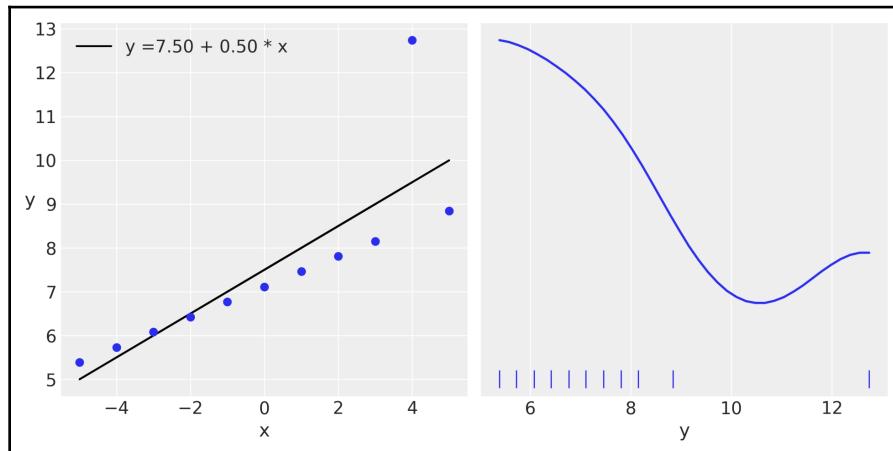


Figure 3.10

Now, we are going to rewrite the previous model (`model_g`), but this time we are going to use a Student's t-distribution instead of a Gaussian. This change also introduces the need to specify the value of ν , the normality parameter. If you do not remember the role of this parameter, check Chapter 2, *Programming Probabilistically*, before continuing.

In the following model, we are using a shifted exponential to avoid values of ν close to zero. The non-shifted exponential puts too much weight on values close to zero. In my experience, this is fine for data with no to moderate outliers, but for data with extreme outliers (or data with a few bulk points), like in the Anscombe's third dataset, it is better to avoid such low values. Take this, as well as other priors recommendations, with a pinch of salt. The defaults are good starting points, but there's no need to stick to them. Other common priors for ν are `gamma(2, 0.1)` or `gamma(mu=20, sd=15)`:

```
with pm.Model() as model_t:
    α = pm.Normal('α', mu=y_3.mean(), sd=1)
    β = pm.Normal('β', mu=0, sd=1)
    ε = pm.HalfNormal('ε', 5)
    ν_ = pm.Exponential('ν_', 1/29)
    ν = pm.Deterministic('ν', ν_ + 1)

    y_pred = pm.StudentT('y_pred', mu=α + β * x_3,
                          sd=ε, nu=ν, observed=y_3)

trace_t = pm.sample(2000)
```

In Figure 3.11, we can see the robust fit, according `model_t`, and the non-robust fit, according to SciPy's `linregress` (this function is doing least-squares regression). As a bonus exercise, you may try adding the best line that's obtained using `model_g`:

```
beta_c, alpha_c = stats.linregress(x_3, y_3)[:2]

plt.plot(x_3, (alpha_c + beta_c * x_3), 'k', label='non-robust', alpha=0.5)
plt.plot(x_3, y_3, 'C0o')
alpha_m = trace_t['α'].mean()
beta_m = trace_t['β'].mean()
plt.plot(x_3, alpha_m + beta_m * x_3, c='k', label='robust')

plt.xlabel('x')
plt.ylabel('y', rotation=0)
plt.legend(loc=2)
plt.tight_layout()
```

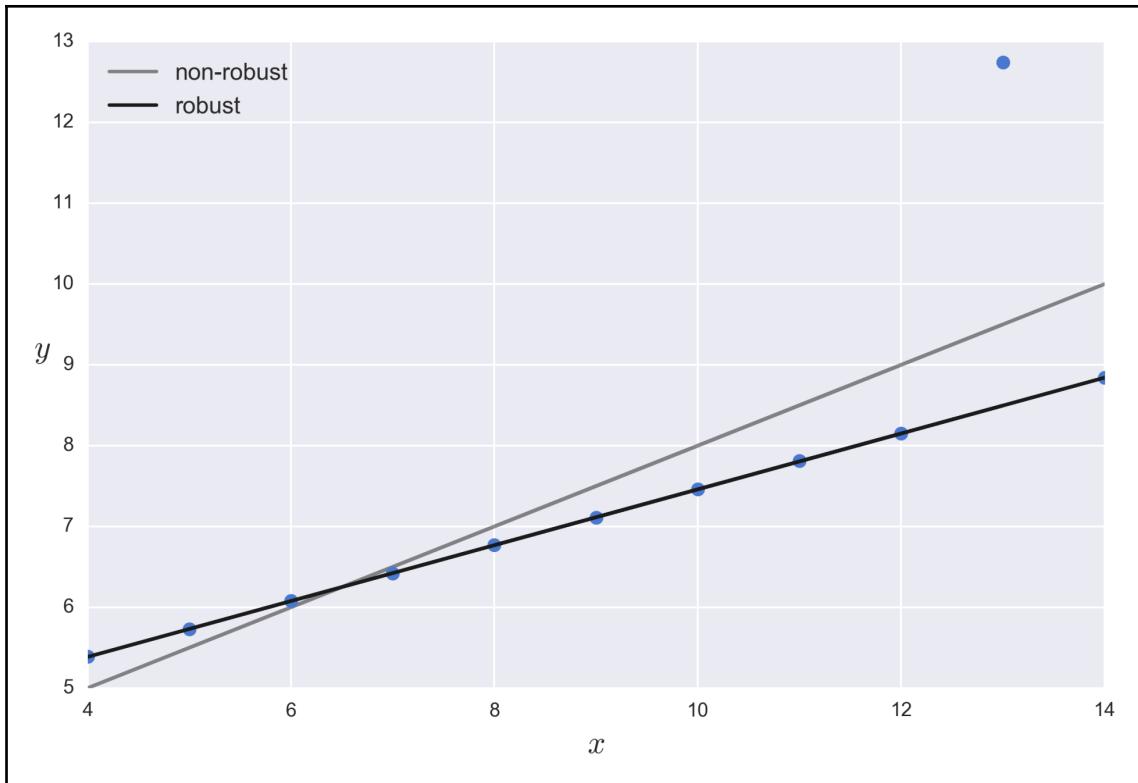


Figure 3.11

While the *non-robust* fit tries to compromise and include all points, the *robust* Bayesian model, `model_t`, automatically *discards* one point and fits a line that passes exactly through all the remaining points. I know this is a very peculiar dataset, but the message remains for *more real* and complex ones. A Student's t-distribution, due to its heavier tails, is able to give *less importance* to points that are far away from the bulk of the data.

Before moving on, take a moment to contemplate the values of the parameters (I am omitting the intermediate ν -parameter as it is not of direct interest):

```
az.summary(trace_t, var_names=varnames)
```

	mean	sd	mc error	hpd 3%	hpd 97%	eff_n	r_hat
α	7.11	0.00	0.0	7.11	7.12	2216.0	1.0
β	0.35	0.00	0.0	0.34	0.35	2156.0	1.0
ϵ	0.00	0.00	0.0	0.00	0.01	1257.0	1.0
ν	1.21	0.21	0.0	1.00	1.58	3138.0	1.0

As you can see, the values of α , β , and ϵ are very narrowly defined and even more so with ϵ , which is basically 0. This is totally reasonable, given that we are fitting a line to a perfectly aligned set of points (if we ignore the *outlier* point).

Let's run a posterior predictive check to explore how well our model captures the data. We can let PyMC3 do the hard work of sampling from the posterior for us:

```
ppc = pm.sample_posterior_predictive(trace_t, samples=200, model=model_t,
random_seed=2)

data_ppc = az.from_pymc3(trace=trace_t, posterior_predictive=ppc)
ax = az.plot_ppc(data_ppc, figsize=(12, 6), mean=True)
plt.xlim(0, 12)
```

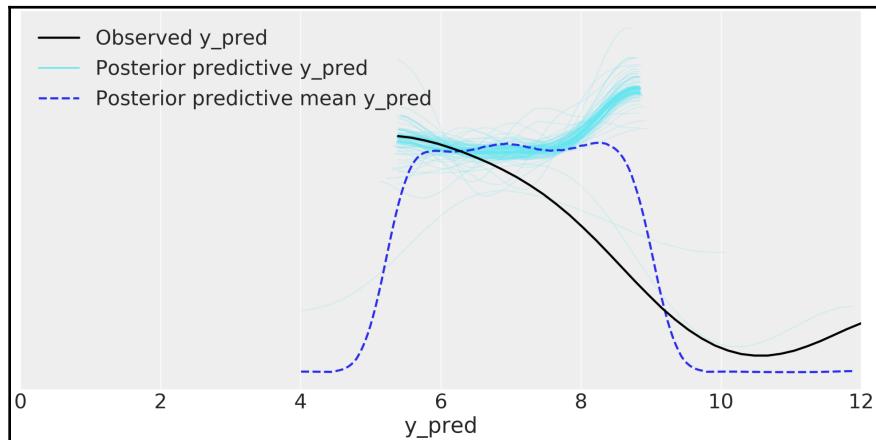


Figure 3.12

For the bulk of the data, we get a very good match. Also notice that our model predicts values away from the bulk to both sides and not just above the bulk. For our current purposes, this model is performing just fine and it does not need further changes. Nevertheless, notice that for some problems, we may want to avoid this. In such a case, we should probably go back and change the model to restrict the possible values of `y_pred` to positive values.

Hierarchical linear regression

In the previous chapter, we learned about the rudiments of hierarchical models. We can apply this concept to linear regression as well. This allows models to deal with inferences at the group level and estimations above the group level. As we already saw, this is done by including **hyperpriors**.

We are going to create eight related data groups, including one group with a single data point:

```
N = 20
M = 8
idx = np.repeat(range(M-1), N)
idx = np.append(idx, 7)
np.random.seed(314)

alpha_real = np.random.normal(2.5, 0.5, size=M)
beta_real = np.random.beta(6, 1, size=M)
eps_real = np.random.normal(0, 0.5, size=len(idx))

y_m = np.zeros(len(idx))
x_m = np.random.normal(10, 1, len(idx))
y_m = alpha_real[idx] + beta_real[idx] * x_m + eps_real

_, ax = plt.subplots(2, 4, figsize=(10, 5), sharex=True, sharey=True)
ax = np.ravel(ax)
j, k = 0, N
for i in range(M):
    ax[i].scatter(x_m[j:k], y_m[j:k])
    ax[i].set_xlabel(f'x_{i}')
    ax[i].set_ylabel(f'y_{i}', rotation=0, labelpad=15)
    ax[i].set_xlim(6, 15)
    ax[i].set_ylim(7, 17)
```

```
j += N
k += N
plt.tight_layout()
```

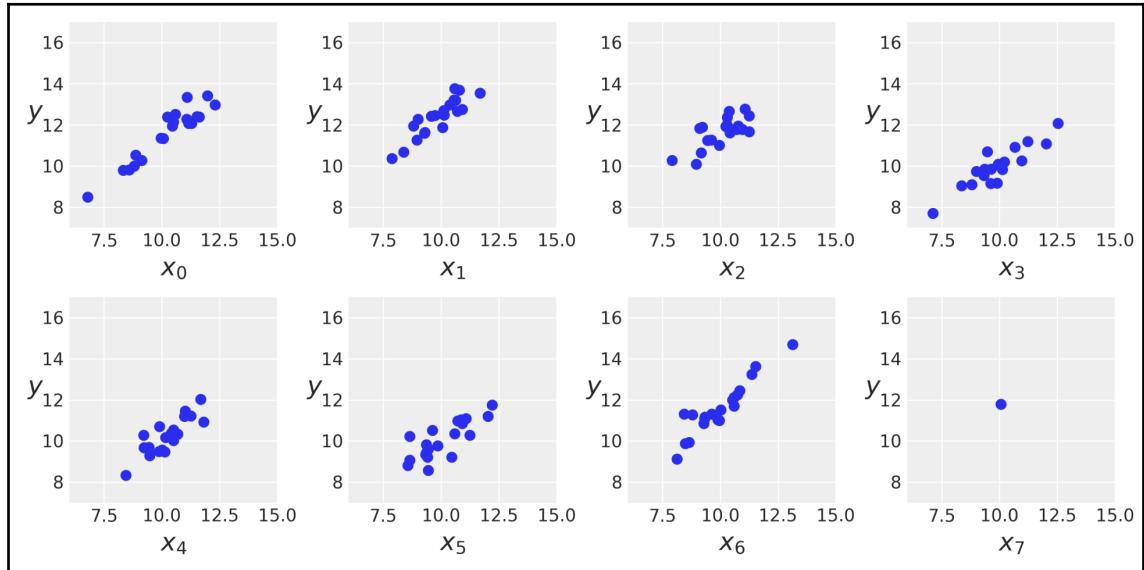


Figure 3.13

Now, we are going to center the data before feeding it to the model:

```
x_centered = x_m - x_m.mean()
```

First we are going to fit a non-hierarchical model, just as we have already seen. The only difference is that we are now going to include code to re-scale α to the original scale:

```
with pm.Model() as unpooled_model:
    a_tmp = pm.Normal('a_tmp', mu=0, sd=10, shape=M)
    β = pm.Normal('β', mu=0, sd=10, shape=M)
    ε = pm.HalfCauchy('ε', 5)
    ν = pm.Exponential('ν', 1/30)

    y_pred = pm.StudentT('y_pred', mu=a_tmp[idx] + β[idx] * x_centered,
                         sd=ε, nu=ν, observed=y_m)

    α = pm.Deterministic('α', a_tmp - β * x_m.mean())

    trace_up = pm.sample(2000)
```

As we can see in *Figure 3.14*, the estimations for the α_7 and β_7 parameters are very very wide compared to the rest of the $\alpha_{0:6}$ and $\beta_{0:6}$ parameters:

```
az.plot_forest(trace_up, var_names=[' $\alpha$ ', ' $\beta$ '], combined=True)
```

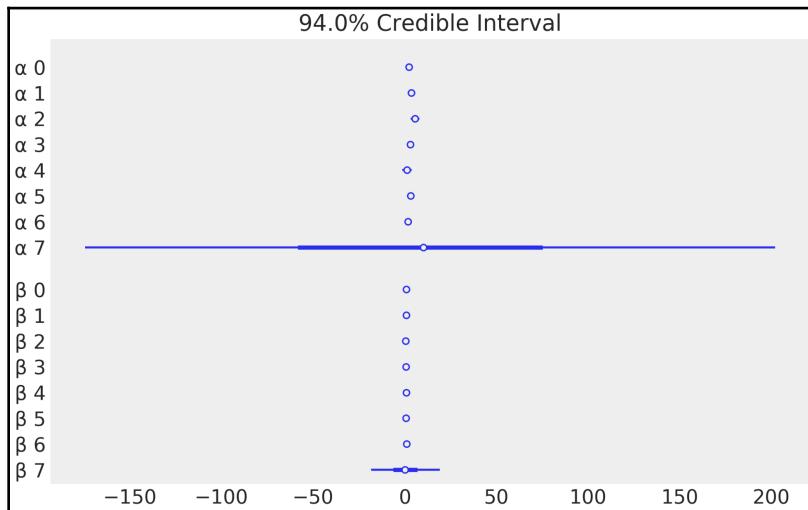


Figure 3.14

You may have already guessed what is going on—it does not make sense to try to fit a line through a single point. We need at least two points, otherwise the parameters α and β are unbounded. That is totally true unless we provide some more information; we can do this by using priors. Putting a strong prior for α can lead to a well-defined set of lines, even for one data point. Another way to convey information is by defining hierarchical models, since hierarchical models allow information to be shared between groups, shrinking the plausible values of the estimated parameters. This becomes very useful in cases where we have groups with sparse data. In this example, we have taken that sparsity of the data to the extreme—a group with a single data point!

Now, we are going to implement a hierarchical model that is the same as a regular linear regression model but with hyperpriors, as you can see in the following Kruschke diagram:

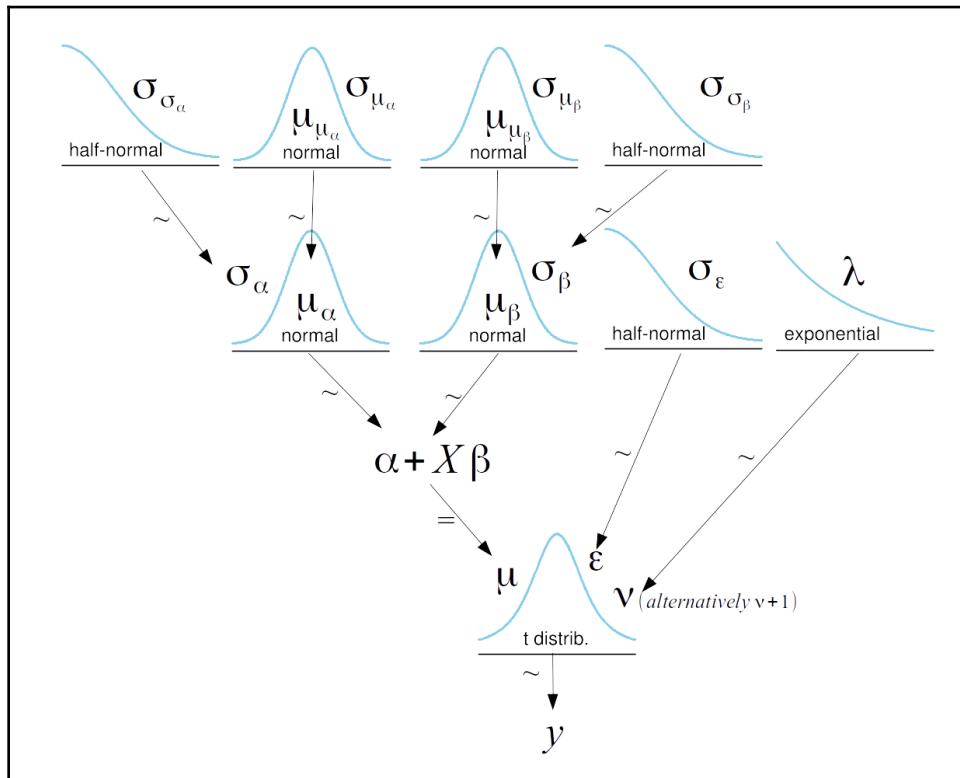


Figure 3.15

In the following PyMC3 model, the main differences compared to previous models are:

- We add hyperpriors.
- We also add a few lines to transform the parameters back to the original uncentered scale. Remember that this is not mandatory; we can keep the parameters in the transformed scale as long as we keep that in mind when interpreting the results:

```
with pm.Model() as hierarchical_model:
    # hyper-priors
    alpha_mu_tmp = pm.Normal('alpha_mu_tmp', mu=0, sd=10)
    alpha_sigma_tmp = pm.HalfNormal('alpha_sigma_tmp', 10)
    beta_mu = pm.Normal('beta_mu', mu=0, sd=10)
    beta_sigma = pm.HalfNormal('beta_sigma', sd=10)
```

```

# priors
α_tmp = pm.Normal('α_tmp', mu=α_μ_tmp, sd=α_σ_tmp, shape=M)
β = pm.Normal('β', mu=β_μ, sd=β_σ, shape=M)
ε = pm.HalfCauchy('ε', 5)
ν = pm.Exponential('ν', 1/30)

y_pred = pm.StudentT('y_pred',
                      mu=α_tmp[idx] + β[idx] * x_centered,
                      sd=ε, nu=ν, observed=y_m)

α = pm.Deterministic('α', α_tmp - β * x_m.mean())
α_μ = pm.Deterministic('α_μ', α_μ_tmp - β_μ * x_m.mean())
α_σ = pm.Deterministic('α_sd', α_σ_tmp - β_μ * x_m.mean())

trace_hm = pm.sample(1000)

```

To compare the results of `unpooled_model` with `hierarchical_model`, we are going to do one more forest plot:

```
az.plot_forest(trace_hm, var_names=['α', 'β'], combined=True)
```

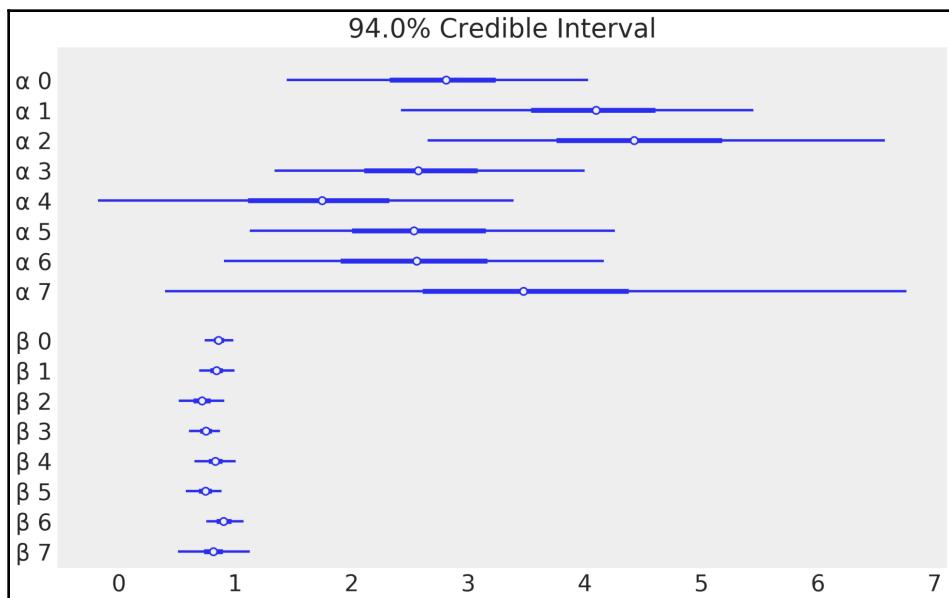


Figure 3.16

A good way to compare models using `az.plot_forest()` is to show the parameters of both models (`unpooled_model`, `hierarchical_model`) simultaneously in the same plot. To do this, you just need to pass a list of traces.

To better understand what the model is capturing about the data, let's plot the fitted lines for each of the eight groups:

```
_ , ax = plt.subplots(2, 4, figsize=(10, 5), sharex=True, sharey=True,
                     constrained_layout=True)
ax = np.ravel(ax)
j, k = 0, N
x_range = np.linspace(x_m.min(), x_m.max(), 10)
for i in range(M):
    ax[i].scatter(x_m[j:k], y_m[j:k])
    ax[i].set_xlabel(f'x_{i}')
    ax[i].set_ylabel(f'y_{i}', labelpad=17, rotation=0)
    alpha_m = trace_hm['α'][:, i].mean()
    beta_m = trace_hm['β'][:, i].mean()
    ax[i].plot(x_range, alpha_m + beta_m * x_range, c='k',
                label=f'y = {alpha_m:.2f} + {beta_m:.2f} * x')
plt.xlim(x_m.min()-1, x_m.max()+1)
plt.ylim(y_m.min()-1, y_m.max()+1)
j += N
k += N
```

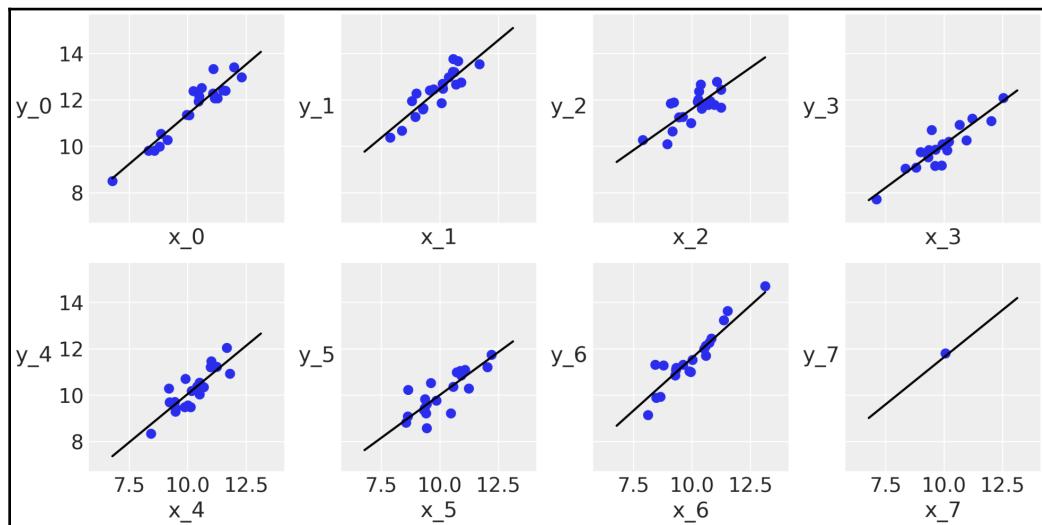


Figure 3.17

Using a hierarchical model, we were able to fit a line to a single data point, as you can see in *Figure 3.17*. At first, this may sound weird or even fishy, but this is just a consequence of the structure of the hierarchical model. Each line is informed by the lines of the other groups, thus we are not truly adjusting a line to a single point. Instead, we are adjusting a line to a single point that's been informed by the points in the other groups.

Correlation, causation, and the messiness of life

Suppose we want to predict how much we are going to pay for gas to heat our home during winter and suppose we know the amount of sun radiation in the area we live. In this example, the sun radiation is going to be the independent variable, x , and the bill is the dependent variable, y . It is very important to note that there is nothing forbidding us to invert the question and ask about the amount of sun radiation, given the bill. If we establish a linear relationship (or any other relation, for that matter), we can go from x to y or vice versa. We call a variable independent because its value cannot be predicted by the model; instead, it is an input of the model and the dependent variable is the output. We say that the value of one variable depends on the value of the other because we build a model specifying such a dependency. We are not establishing a causal relationship between variables and we are not saying x causes y . Always remember the following mantra, *correlation does not imply causation*. Let's develop this idea a little bit more. Even if we are able to predict the gas bill of a home from the sun radiation and the sun radiation from the gas bill, we can agree that it isn't true that we can control the amount of radiation emitted by the sun by changing the thermostat of our house! However, it is true that higher sun radiation can be related to a lower gas bill.

It is therefore important to remember that the statistical model we are building is one thing and the physical mechanism relating the variables is another. To establish that a correlation can be interpreted in terms of causation, we need to add a plausible physical mechanism to the description of the problem; a correlation is simply not enough. A very nice and amusing page showing clear examples of correlated variables with no causal relationship can be found at <http://www.tylervigen.com/spurious-correlations>.

Is a correlation useless when it comes to establishing a causal link? Not at all—a correlation can, in fact, support a causal link if we perform a carefully designed experiment. For example, we know that global warming is highly correlated to the increasing levels of atmospheric CO₂. From this observation alone, we cannot conclude whether higher temperatures are causing an increase in the levels of CO₂ or if the higher levels of the carbonic gas are increasing the temperature.

Even more, it could happen that there is a third variable that we are not taking into account, and this variable is producing both higher temperatures and higher levels of CO₂. However—and pay attention to this—we can do an experiment to gain insight into this problem. One possible experiment could be the following; we build a set of glass tanks filled with different quantities of CO₂. We can have one with regular air (that contains ~0.04% of CO₂) and the others with an increasing amount of CO₂. We then let these tanks receive sun light for, let's say, three hours. If we do this, we will verify that tanks with higher levels of CO₂ have higher final temperatures. Hence, we will conclude that indeed CO₂ is a greenhouse effect gas. Using the same experiment, we can also measure the concentration of CO₂ at the end of the experiment to check that temperature does not cause the CO₂ level to increase, at least not from air. It is this experimental setting together with statistical models that gives evidence in favor of CO₂ emissions contributing to global warming.

Another important aspect we will discuss following this example is that, even when the sun radiation and the gas bill are connected and maybe the sun radiation can be used to predict the gas bill, the relationship is more complicated, and other variables are involved. In fact, higher temperature can contribute to higher levels of CO₂ because oceans are a reservoir of CO₂, and CO₂ is less soluble in water when temperatures increase. Also, a higher sun radiation means that more energy is delivered to a home. Part of that energy is reflected and part is turned into heat, part of the heat is absorbed by the house, and part is lost to the environment. The amount of heat lost depends upon several factors, such as the outside temperature and the speed of the wind. Then, we have the fact that the gas bill could also be affected by other factors, such as the international price of oil and gas, the costs/profits for the company (and its level of *greediness*), and also how tightly the government regulates the company.

In summary, life is messy, problems are not generally simple to understand, and context is always important. Statistical models can help us achieve better interpretations, reduce the risk of making nonsensical statements, and get better predictions, but none of this is automatic.

Polynomial regression

I hope you are excited about the skills you have learned about so far in this chapter. Now, we are going to learn how to fit curves using linear regression. One way to fit curves using a linear regression model is by building a polynomial, like this:

$$\mu = \beta_0 x^0 + \beta_1 x^1 + \dots + \beta_m x^m \quad (3.12)$$

If we pay attention, we can see that the simple linear model is hidden in this polynomial. To uncover it, all we need to do is to make all the β_n coefficients higher than one exactly zero. Then, we will get:

$$\mu = \beta_0 + \beta_1 x^1 \quad (3.13)$$

Polynomial regression is still linear regression; the linearity in the model is related to how the parameters enter the model, not the variables. Let's try building a polynomial regression of degree 2:

$$\mu = \beta_0 + \beta_1 x^1 + \beta_2 x^2 \quad (3.14)$$

The third term controls the curvature of the relationship.

As a dataset, we are going to use the second group of the Anscombe quartet:

```
x_2 = ans[ans.group == 'II']['x'].values
y_2 = ans[ans.group == 'II']['y'].values
x_2 = x_2 - x_2.mean()

plt.scatter(x_2, y_2)
plt.xlabel('x')
plt.ylabel('y', rotation=0)
```

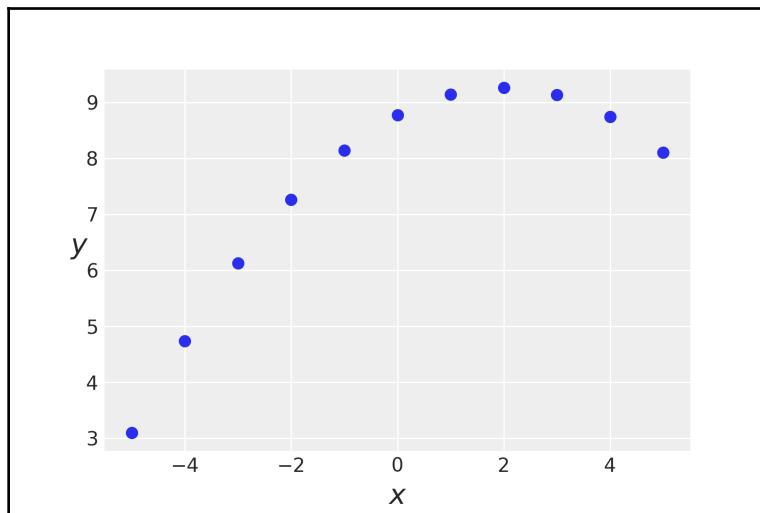


Figure 3.18

And now we build a PyMC3 model:

```
with pm.Model() as model_poly:
    a = pm.Normal('a', mu=y_2.mean(), sd=1)
    β1 = pm.Normal('β1', mu=0, sd=1)
    β2 = pm.Normal('β2', mu=0, sd=1)
    ε = pm.HalfCauchy('ε', 5)

    mu = a + β1 * x_2 + β2 * x_2**2

    y_pred = pm.Normal('y_pred', mu=mu, sd=ε, observed=y_2)

    trace_poly = pm.sample(2000)
```

Once again, we are going to omit some checks and summaries and just plot the results, which will be a nice curved line fitting the data almost with no errors. Take into account the minimalist nature of the dataset:

```
x_p = np.linspace(-6, 6)
y_p = trace_poly['a'].mean() + trace_poly['β1'].mean() * \
      x_p + trace_poly['β2'].mean() * x_p**2
plt.scatter(x_2, y_2)
plt.xlabel('x')
plt.ylabel('y', rotation=0)
plt.plot(x_p, y_p, c='C1')
```

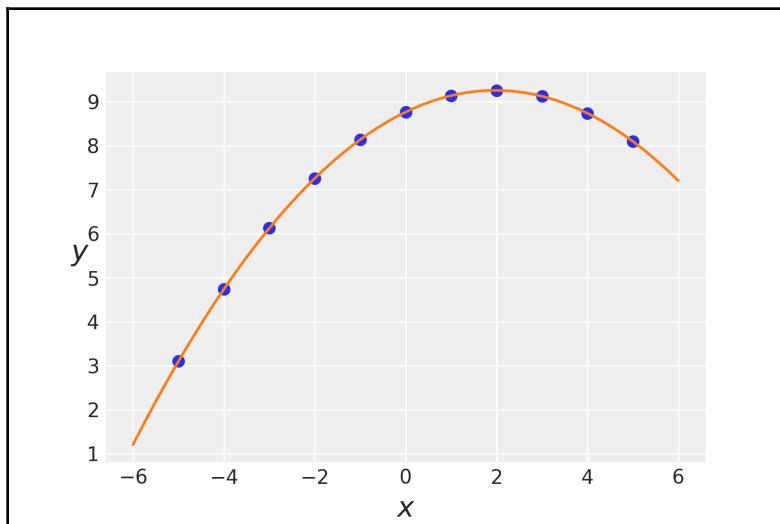


Figure 3.19

Interpreting the parameters of a polynomial regression

One of the problems of polynomial regression is the interpretation of the parameters. If we want to know how y changes per unit change of x ; we cannot just check the value of β_1 , because β_2 and higher coefficients, if present, have an effect on such a quantity. Therefore, the β coefficients are no longer slopes, they are something else. In the previous example, β_1 is positive and hence the curve begins with a positive slope, but β_2 is negative and hence, after a while, the line begins to curve downwards. So, it is like we have two forces at play, one pushing the line up and the other down. The interplay depends on the value of x . When $x \lesssim 11$ (on the original scale or 2 on the centered scale), the dominating contribution comes from β_1 , and when $x \gtrsim 11\beta_2$ dominates.

The problem of interpreting the parameters is not just a mathematical problem. If this were the case, we could solve it by careful inspection and understanding of the model. The problem is that, in many cases, the parameters are not translated to meaningful quantities in our domain knowledge. We cannot relate them with the metabolic rate of a cell or the energy emitted by a distant galaxy or the number of bedrooms in a house. They are just *knobs* we can tweak to improve the fit but without a clear physical meaning. In practice, most people will agree that polynomials of order higher than 2 or 3 are not generally very useful models and alternatives are preferred, such as Gaussian processes, which is the main subject of Chapter 7, *Gaussian Processes*.

Polynomial regression – the ultimate model?

As we saw, we can think of the line as a sub-model of the parabola when β_2 is equal to zero, and a line is also a sub-model of a cubic model when β_2 and β_3 are equal to zero. Of course, the parabola is a sub-model of the cubic one when β_3 . OK, I will stop here, but I think you already notice the pattern. This suggests that we can, in principle, use polynomial regression to fit an arbitrary complex model. We just built a polynomial with the *right* order. We could do this by increasing the order one by one until we do not observe any improvement on the fit, or we could build an infinite order polynomial and then somehow make all the *irrelevant* coefficients zero until we get a perfect fit of our data. To test this idea, we can start with a very simple example. Let's use the quadratic model to fit the third group of the Anscombe dataset. I will wait here while you do that... I am still waiting, don't worry...

OK, if you really did the exercise, you will have observed by yourself that it is possible to use a quadratic model to fit a line. While it may seem that the previous simple experiment validates the idea of building an infinite order polynomial to the fit data, we should curb our enthusiasm. In general, using a polynomial to fit data is not the best idea. Why? Because it does not matter which data we have. In principle, it is always possible to find a polynomial to fit the data perfectly! In fact, it is pretty easy to compute the exact order the polynomial should have. Why is fitting data perfectly problematic? Well that's the subject of Chapter 5, *Model Comparison*, but spoiler alert! A model that fits your current data perfectly will, in general, do a very poor job at fitting/describing unobserved data. The reason for this is that any real dataset will contain noise and sometimes (hopefully) an interesting pattern. An arbitrary over-complex model will fit the noise, leading to poor predictions. This is known as **overfitting**, and is a pervasive phenomena in statistics and ML. Polynomial regression makes for a convenient straw man when it comes to overfitting because it is easy to see the problem. This generates intuition that we can translate to more complex models, which can lead to overfitting without us really noticing. Part of the job, when analyzing data, is to be sure that models are not overfitting it. We will discuss this topic in detail in Chapter 5, *Model Comparison*.

Multiple linear regression

So far, we have been working with one dependent variable and one independent variable. Nevertheless, it is not unusual to have several independent variables that we want to include in our model. Some examples could be:

- Perceived quality of wine (dependent) and acidity, density, alcohol level, residual sugar, and sulphates content (independent variables)
- A student's average grades (dependent) and family income, distance from home to school, and mother's education (categorical variable)

We can easily extend the simple linear regression model to deal with more than one independent variable. We call this model multiple linear regression or less often multivariable linear regression (not to be confused with multivariate linear regression, the case where we have multiple dependent variables).

In a multiple linear regression model, we model the mean of the dependent variable as follows:

$$\mu = \alpha + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_m x_m \quad (3.15)$$

Notice that this looks similar to a polynomial regression, but is not exactly the same. For multiple linear regression, we have different variables instead of successive powers of the same variable. From the point of view of multiple linear regression, we can say that a polynomial regression is like a multiple linear regression but with made-up variables.

Using linear algebra notation, we can write a shorter version:

$$\mu = \alpha + X\beta \quad (3.16)$$

Here, β is a vector of coefficients of length m , that is, the number of independent variables. The variable x is a matrix of size $m \times n$ if n is the number of observations and m is the number of independent variables. If you are a little rusty with your linear algebra, you may want to check the Wikipedia article about the dot product between two vectors and its generalization to matrix multiplication. Basically what you need to know is that we are just using a shorter and more convenient way to write our model:

$$X\beta = \sum_{i=1}^n \beta_i x_i = \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_m x_m \quad (3.17)$$

Using the simple linear regression model, we find a straight line that (hopefully) explains our data. Under the multiple linear regression model we find, instead, a hyperplane of dimension m . Thus, the multiple linear regression model is essentially the same as the simple linear regression model, the only difference being that now β is a vector and X is a matrix.

Let's define our data:

```
np.random.seed(314)
N = 100
alpha_real = 2.5
beta_real = [0.9, 1.5]
eps_real = np.random.normal(0, 0.5, size=N)

X = np.array([np.random.normal(i, j, N) for i, j in zip([10, 2], [1, 1.5])]).T
X_mean = X.mean(axis=0, keepdims=True)
X_centered = X - X_mean
y = alpha_real + np.dot(X, beta_real) + eps_real
```

Now, we are going to define a convenient function to plot three scatter plots, two between each independent variable, and the dependent variable, and the last one between both dependent variables. This is nothing fancy at all, just a function we will use a couple of times during the rest of this chapter:

```
def scatter_plot(x, y):
    plt.figure(figsize=(10, 10))
    for idx, x_i in enumerate(x.T):
        plt.subplot(2, 2, idx+1)
        plt.scatter(x_i, y)
        plt.xlabel(f'x_{idx+1}')
        plt.ylabel(f'y', rotation=0)

    plt.subplot(2, 2, idx+2)
    plt.scatter(x[:, 0], x[:, 1])
    plt.xlabel(f'x_{idx}')
    plt.ylabel(f'x_{idx+1}', rotation=0)
```

Using the `scatter_plot` function we just defined, we can visualize our synthetic data:

```
scatter_plot(X_centered, y)
```

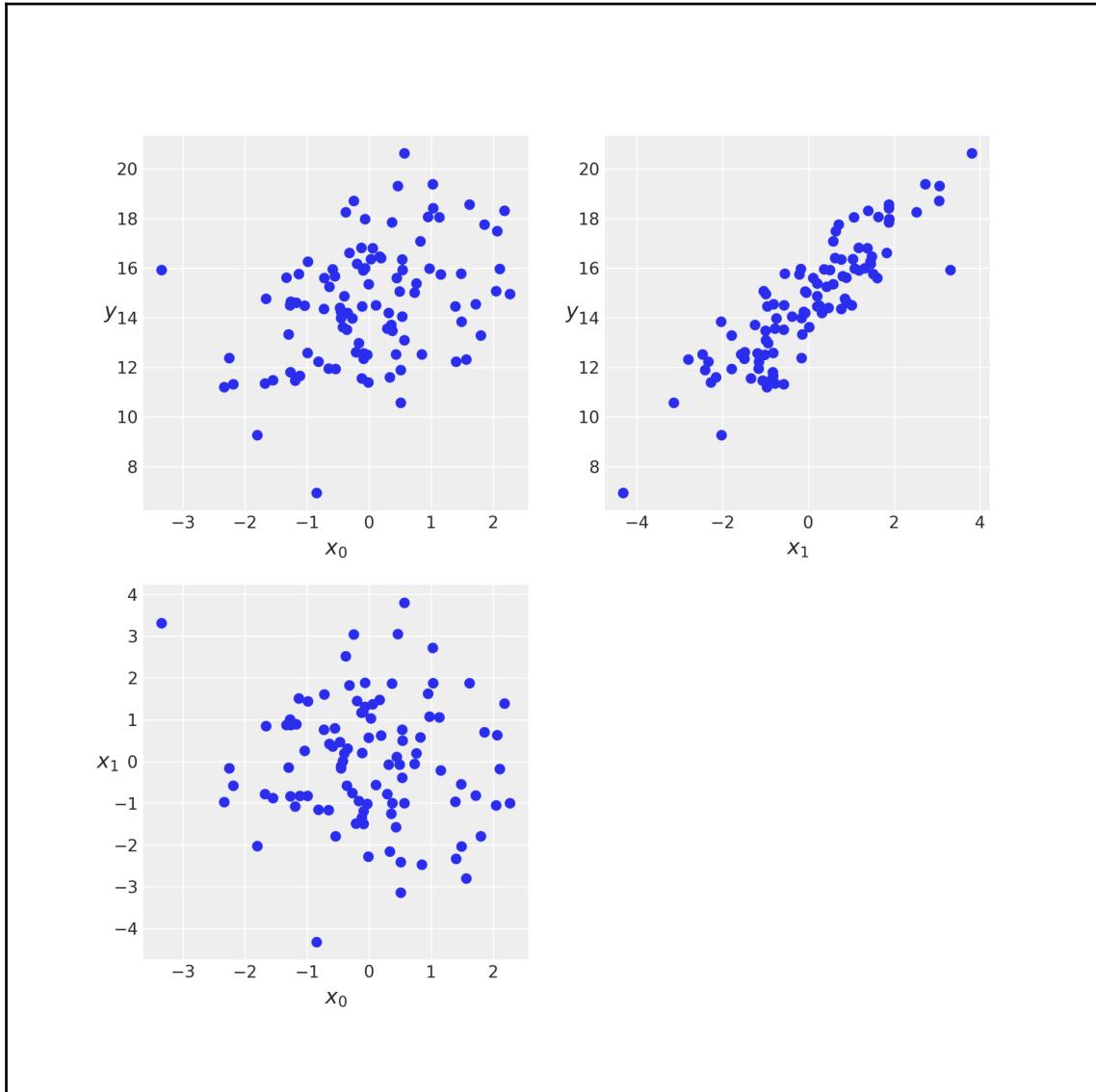


Figure 3.20

Now, let's use PyMC3 to define a model that's suitable for multiple linear regression. As expected, the code looks pretty similar to what we used for simple linear regression. The main differences are:

- The variable beta is a Gaussian with shape=2, one slope per each independent variable
- We define the variable μ using the dot product function `pm.math.dot()`

If you are familiar with NumPy, you probably know that NumPy includes a dot function, and from Python 3.5 (and from NumPy 1.10), a new matrix operator, `@`, is included.

Nevertheless, here, we use the dot function from PyMC3, which is just an alias for a Theano matrix multiplication operator. We are doing so because the variable β is a Theano tensor and not a NumPy array:

```
with pm.Model() as model_mlr:
    α_tmp = pm.Normal('α_tmp', mu=0, sd=10)
    β = pm.Normal('β', mu=0, sd=1, shape=2)
    ε = pm.HalfCauchy('ε', 5)

    μ = α_tmp + pm.math.dot(X_centered, β)

    α = pm.Deterministic('α', α_tmp - pm.math.dot(X_mean, β))

    y_pred = pm.Normal('y_pred', mu=μ, sd=ε, observed=y)

trace_mlr = pm.sample(2000)
```

Let's summarize the inferred parameters values for easier analysis of the results. How well did the model do?

```
varnames = ['α', 'β', 'ε']
az.summary(trace_mlr, var_names=varnames)
```

	mean	sd	mc error	hpd 3%	hpd 97%	eff_n	r_hat
α[0]	1.86	0.46	0.0	0.95	2.69	5251.0	1.0
β[0]	0.97	0.04	0.0	0.89	1.05	5467.0	1.0
β[1]	1.47	0.03	0.0	1.40	1.53	5464.0	1.0
ε	0.47	0.03	0.0	0.41	0.54	4159.0	1.0

As we can see, our model is capable of recovering the correct values (check the values used to generate the synthetic data).

In the following sections, we are going to focus on some precautions we should take when analyzing the results of a multiple regression model, especially the interpretation of the slopes. One important message to take home is that, in multiple linear regression, each parameter only makes sense in the context of the other parameters.

Confounding variables and redundant variables

Imagine the following situation. We have a variable z correlated with the predictor variable x and, at the same time, with the predicted variable y . Suppose that the variable z is the one responsible for causing x and y . For example, z could be the industrial revolution (a really complex variable!), x the number of pirates, and y the concentration of CO₂. This example should be very familiar to the Pastafarian reader. If we omit z from our analysis, we might end up with a nice linear relation between x and y , and we may even be able to predict y from x . However, if our interest lies in understating global warming, we could totally miss what is really going on with the actual mechanism relating these variables.

We already discussed that correlation does not imply causation. One reason this is not necessarily true is that we may be omitting the variable z from our analysis. When this happens, z is named as a **confounding variable** or confounding factor. In many real scenarios, z is easy to miss. Maybe we did not measure it or it was not present in the dataset that was sent to us, or we did not even think it could possibly be related to our problem. Not taking into account confounding variables in an analysis could lead us to establish spurious correlations. This is always a problem when we try to explain something and could also be problematic when we try to predict something without caring about understanding the underlying mechanism. Understanding the mechanism helps us translate what we have learned to new situations; blind predictions do not always have good transferability. For example, the number of sneakers produced in one country could be used as an easy-to-measure indicator of the strength of its economy, but this could be a terrible predictor for other countries with a different production matrix or cultural background.

We are going to use synthetic data to explore the concept of a confounding variable. The following code simulates a confounding variable as x_1 . Notice how this variable has influences on x_2 and y :

```
np.random.seed(42)
N = 100
x_1 = np.random.normal(size=N)
x_2 = x_1 + np.random.normal(size=N, scale=1)
#x_2 = x_1 + np.random.normal(size=N, scale=0.01)
y = x_1 + np.random.normal(size=N)
X = np.vstack((x_1, x_2)).T
```

Notice that by virtue of the way we create these variables, they are already centered, as we can easily check it with the convenient `scatter_plot` function we wrote earlier. Therefore, we do not need to center the data to speed up the inference process:

```
scatter_plot(x, y)
```

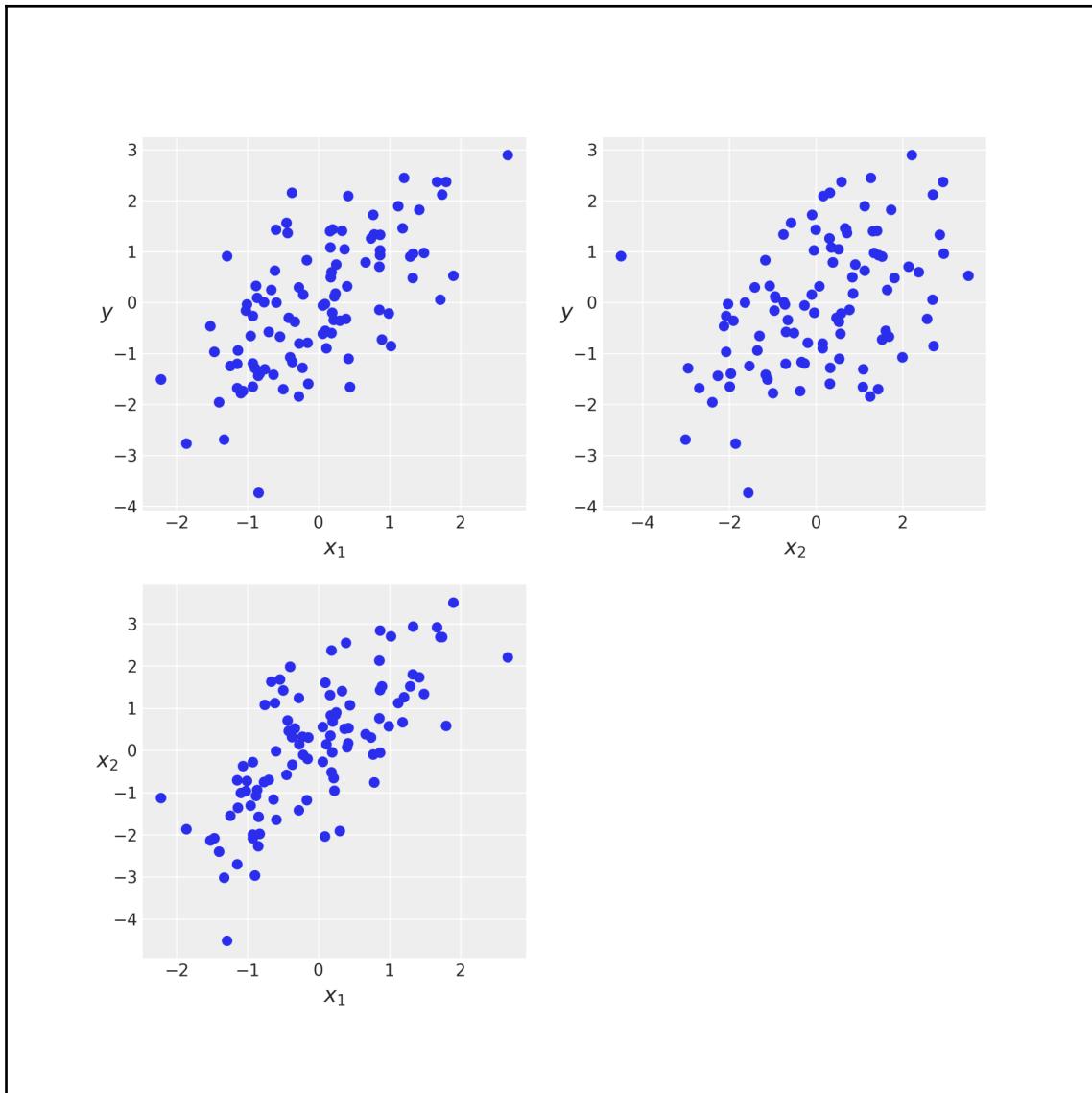


Figure 3.21

Now, we are going to build three related models, the first one being `m_x1x2`, which is a linear regression model with two independent variables, x_1 and x_2 (stacked together in the variable `X`). The second model, `m_x1`, is a simple linear regression for x_1 and the third one, `m_x2`, is a simple linear regression for x_2 :

```
with pm.Model() as m_x1x2:  
    α = pm.Normal('α', mu=0, sd=10)  
    β1 = pm.Normal('β1', mu=0, sd=10)  
    β2 = pm.Normal('β2', mu=0, sd=10)  
    ε = pm.HalfCauchy('ε', 5)  
  
    μ = α + β1 * X[:, 0] + β2 * X[:, 1]  
  
    y_pred = pm.Normal('y_pred', mu=μ, sd=ε, observed=y)  
  
    trace_x1x2 = pm.sample(2000)  
  
with pm.Model() as m_x1:  
    α = pm.Normal('α', mu=0, sd=10)  
    β1 = pm.Normal('β1', mu=0, sd=10)  
    ε = pm.HalfCauchy('ε', 5)  
  
    μ = α + β1 * X[:, 0]  
  
    y_pred = pm.Normal('y_pred', mu=μ, sd=ε, observed=y)  
  
    trace_x1 = pm.sample(2000)  
  
with pm.Model() as m_x2:  
    α = pm.Normal('α', mu=0, sd=10)  
    β2 = pm.Normal('β2', mu=0, sd=10)  
    ε = pm.HalfCauchy('ε', 5)  
  
    μ = α + β2 * X[:, 1]  
  
    y_pred = pm.Normal('y_pred', mu=μ, sd=ε, observed=y)  
  
    trace_x2 = pm.sample(2000)
```

Take a look at the β parameters for these models. Using a forest plot, we can compare them in a single plot:

```
az.plot_forest([trace_x1x2, trace_x1, trace_x2],
              model_names=['m_x1x2', 'm_x1', 'm_x2'],
              var_names=[' $\beta_1$ ', ' $\beta_2$ '],
              combined=False, colors='cycle', figsize=(8, 3))
```

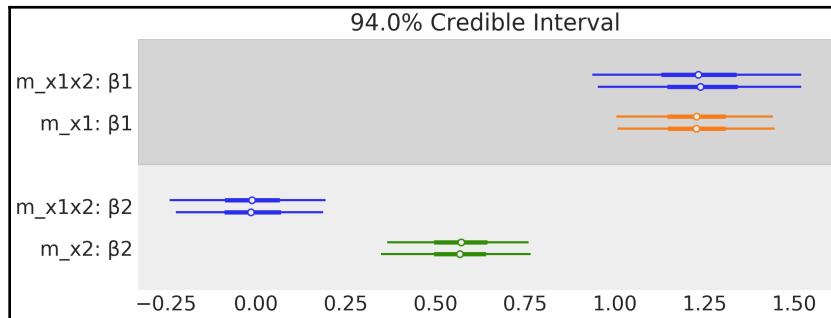


Figure 3.22

As we can see, β_2 for model m_{x1x2} is around zero, indicating an almost null contribution of the x_2 variable to explain y . This is interesting because we already know (check the synthetic data) that the really important variable is x_1 . Also notice—and this is really important—that β_2 for model m_{x2} is around 0.55. This is larger than for model m_{x1x2} . The power of x_2 to predict y is reduced when we take into account x_1 ; the information in x_2 is redundant given x_1 .

Multicollinearity or when the correlation is too high

In the previous example, we saw how a multiple linear regression model reacts to redundant variables, and we saw the importance of considering possible confounding variables. Now, we will take the previous example to an extreme and see what happens when two variables are highly correlated. To study this problem and its consequences for inference, we will use the same synthetic data and model as before, but now we will increase the degree of correlation between x_1 and x_2 by reducing the amount of Gaussian noise we add to x_1 to obtain x_2 :

```
np.random.seed(42)
N = 100
x_1 = np.random.normal(size=N)
```

```
x_2 = x_1 + np.random.normal(size=N, scale=0.01)
y = x_1 + np.random.normal(size=N)
X = np.vstack((x_1, x_2)).T
```

This change in the data-generating code is practically equivalent to summing zero to x_1 , and hence both variables are, for all practical purposes, equal. You can then try varying the values of the scale and using less extreme values, but for now we want to make things crystal clear. After generating the new data, check what the scatter plot looks like:

```
scatter_plot(X, y)
```

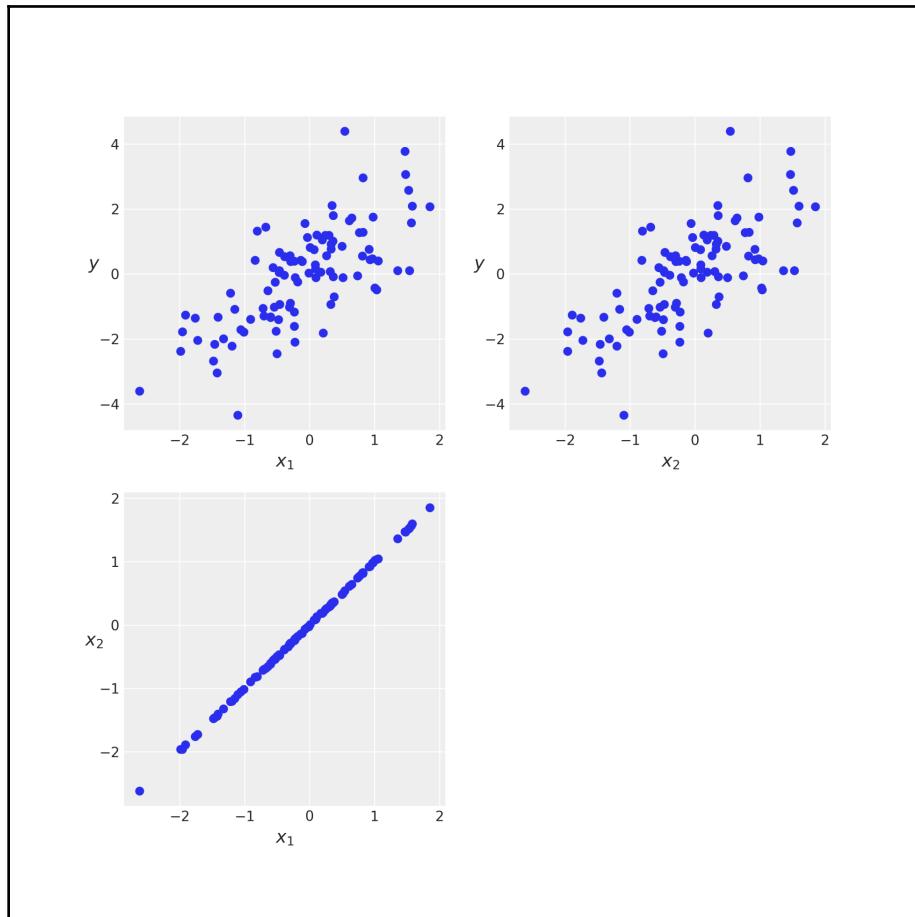


Figure 3.23

You should see something like *Figure 3.23*, the scatter plot for x_1 and x_2 is virtually a straight line with a slope around 1.

We then run a multiple linear regression:

```
with pm.Model() as model_red:
    α = pm.Normal('α', mu=0, sd=10)
    β = pm.Normal('β', mu=0, sd=10, shape=2)
    ε = pm.HalfCauchy('ε', 5)

    μ = α + pm.math.dot(X, β)

    y_pred = pm.Normal('y_pred', mu=μ, sd=ε, observed=y)

    trace_red = pm.sample(2000)
```

Then, we check the results for the β parameters with a forest plot:

```
az.plot_forest(trace_red, var_names=['β'], combined=True, figsize=(8, 2))
```

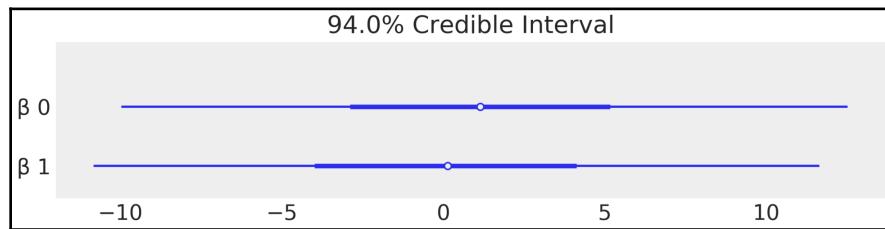


Figure 3.24

The HPD for β coefficients is suspiciously wide. We can get a clue to what is going on with a scatter plot of the β coefficients:

```
az.plot_pair(trace_red, var_names=['β'])
```

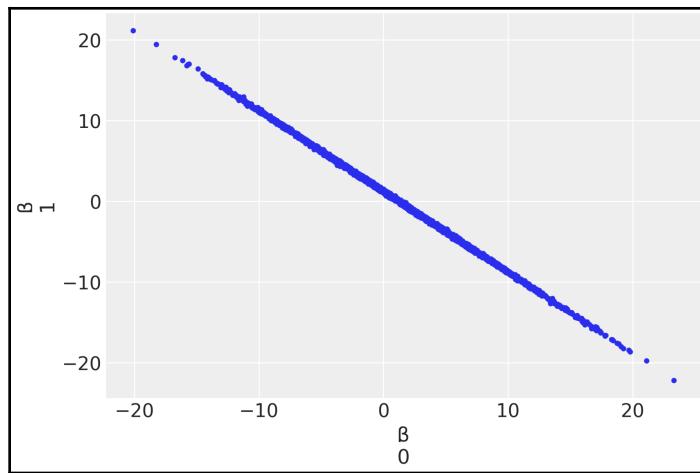


Figure 3.25

Wow! The marginal posterior for β is a really narrow diagonal. When one β coefficient goes up, the other must go down. Both are effectively correlated. This is just a consequence of the model and the data. According to our model, the mean μ is:

$$\mu = \alpha + \beta_1 x_1 + \beta_2 x_2 \quad (3.19)$$

If we assume x_1 and x_2 are not just practically equivalent, but mathematically identical, we can rewrite the model as:

$$\mu = \alpha + (\beta_1 + \beta_2)x \quad (3.20)$$

It turns out that it is the sum of β_1 and β_2 , and not their separated values, that affects μ . We can make β_1 smaller and smaller as long as we get β_2 . We practically do not have two x variables and thus practically we do not have two β parameters. We say that the model is **indeterminate** (or equivalently the data is unable to restrict the parameters in the model). In our example, there are two reasons why β cannot freely move over the $[-\infty, \infty]$ interval. First, both variables are almost the same, but they are not exactly equal, and second, and the most important point, is that we have a prior restricting the plausible values that β can take.

There are a couple of things to notice from this example. First of all, the posterior is just the logical consequence of our data and model, and hence there is nothing wrong with obtaining such wide distributions for β ; C'est la vie. Second, we can rely on this model to make predictions. Try, for example, making posterior predictive checks; the values predicted by the model are in agreement with the data; the model is capturing the data very well. Third, this may be not a very good model to understand our problem. It may be more clever just to remove one of the variables from the model. We will end up having a model that predicted the data as well as before, but with an easier (and simpler) interpretation.

In any real dataset, correlations are going to exist to some degree. How strong should two or more variables be correlated to become a problem? Well, 0.9845. No, just kidding.

Unfortunately, statistics is a discipline with very few magic numbers. It is always possible to do a correlation matrix before running any Bayesian model and check for variables with a high correlation of, let's say, above 0.9 or so. Nevertheless, the problem with this approach is that what really matters is not the pairwise correlations we can observe in a correlation matrix, but the correlation of the variables inside a model, and as we already saw, variables behave differently in isolation than when they are put together in a model. Two or more variables can increase or decrease their correlation when put in the context of other variables in a multiple regression model. As always, careful inspection of the posterior together with an iterative critical approach to model building are highly recommended and can help us to spot problems and understand the data and models.

Just as a quick guide, what should we do if we find highly correlated variables?

- If the correlation is really high, we can eliminate one of the variables from the analysis; given that both variables have similar information, which one we eliminate is often irrelevant. We can eliminate variables based on pure convenience, such as removing the least known variable in our discipline or one that is harder to interpret or measure.
- Another possibility is to create a new variable averaging the redundant variables. A more sophisticated version is to use a variable reduction algorithm such as **Principal Component Analysis (PCA)**. The problem with PCA is that the resulting variables are linear combinations of the original ones obfuscating, in general, the interpretability of the results.
- Yet another solution is to put stronger priors to restrict the plausible values the coefficient can take. In Chapter 6, *Model Comparison* we briefly discuss some choices for such priors, known as **regularizing priors**.

Masking effect variables

One tricky example of how variables contribute to an outcome is the case of **masking effect variables**. Let's create a toy dataset to exemplify this phenomena. Basically, we are creating two independent variables (x_1 and x_2). They are positively correlated to each other and they are correlated to y , but in opposite directions; x_1 is positively correlated and x_2 negatively correlated:

```
np.random.seed(42)
N = 126
r = 0.8
x_1 = np.random.normal(size=N)
x_2 = np.random.normal(x_1, scale=(1 - r ** 2) ** 0.5)
y = np.random.normal(x_1 - x_2)
X = np.vstack((x_1, x_2)).T
scatter_plot(X, y)
```

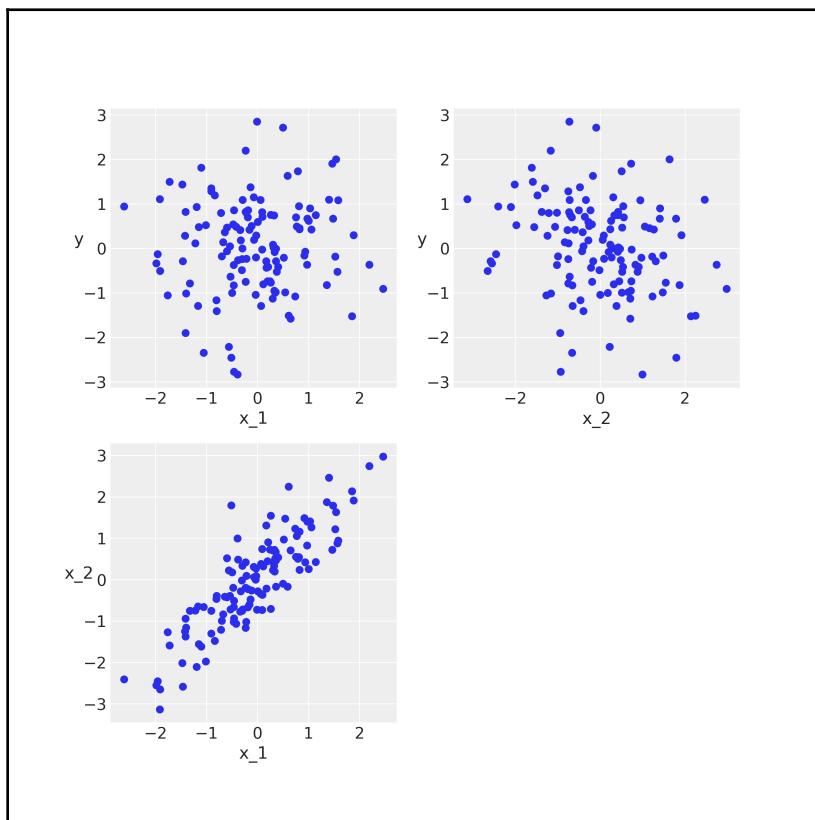


Figure 3.26

As we did before, we are going to build three related models. The first one, m_{x1x2} , is a linear regression model with two independent variables, x_1 and x_2 (stacked together in the variable x). The second model, m_{x1} , is a simple linear regression for x_1 and the third one, m_{x2} , is a simple linear regression for x_2 . After sampling from these models, take a look at the β parameters using a forest plot to compare them in a single plot:

```
az.plot_forest([trace_x1x2, trace_x1, trace_x2],
              model_names=['m_x1x2', 'm_x1', 'm_x2'],
              var_names=['β1', 'β2'],
              combined=True, colors='cycle', figsize=(8, 3))
```

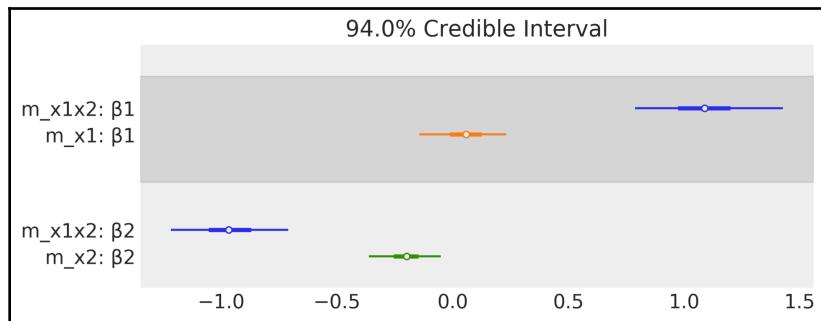


Figure 3.27

According to the posterior, the values of β for m_{x1x2} are close to 1 and -1 (as expected, according to the way we generate the data). For the simple linear regression model, that is when we study each variable on its own, we can see that the values for β are instead closer to zero, indicating a weaker effect.

Notice x_1 is correlated to x_2 . In fact, when x_1 increases, x_2 also increases. Also notice that when y increases, x_1 also increases, but x_2 decreases. As a result of this particular arrangement, we get a partial cancellation of effects unless we include both variables in the same linear regression. The linear regression model is able to untangle these effects because the model is learning for each data point what is the contribution of x_1 to y given a value of x_2 and the other way around for x_2 .

Adding interactions

So far in the definition of the multiple regression model, it is declared (implicitly) that a change in x_i results in a constant change in y , while keeping fixed the values for the rest of the predictor variables. But of course, this is not necessarily true. It could happen that changes in x_i affects y , which is modulated by changes in x_j . A classic example of this behavior is the interaction between drugs. For example, increasing the dose of drug A results in a positive effect on a patient. This is true in the absence of drug B (or for a low dose of B) while the effect of A is negative (even lethal) for increasing doses of B.

In all of the examples we have seen so far, the independent variables contribute additively to the predicted variable. We just add variables (each one multiplied by a coefficient). If we wish to capture effects, like in the drug example, we need to include terms in our model that are not additive. One common option is to multiply variables, for example:

$$\mu = \alpha + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 \quad (3.21)$$

Notice that the β_3 coefficient is multiplying a third variable that is the product of x_1 and x_2 . This non-additive term is an example of what is known in statistics as **interaction**. There are other ways to introduce interactions, but we are going to restrict the discussion to the multiplicative case, as it is the most common expression for interactions.

Interpreting linear models with interactions is not as easy as interpreting linear models without them. Let's rewrite the expression 3.21:

$$\begin{aligned} \mu &= \alpha + \underbrace{(\beta_1 + \beta_3 x_2)x_1}_{\text{slope of } x_1} + \beta_2 x_2 \\ \mu &= \alpha + \beta_1 x_1 + \underbrace{(\beta_2 + \beta_3 x_1)x_2}_{\text{slope of } x_2} \end{aligned} \quad (3.22)$$

This shows us the following:

- The interaction term can be understood as a linear model. Thus, the expression for the mean, μ , is a linear model with a linear model inside of it!
- The interaction is symmetric; we can think of it as the slope of x_1 as a function of x_2 and at the same time as the slope of x_2 as a function of x_1 .

- In a multiple regression model without interactions, we get a hyperplane, that is, a flat hypersurface. An interaction term introduces a curvature in such a hypersurface. This is because slopes are not constant anymore but functions of another variable.
- The coefficient β_1 describes the influence of predictor x_1 only at $x_2 = 0$. This is true because for that value $\beta_3 x_2 = 0$, and then the slope of x_1 reduces to $\beta_1 x_1$. By symmetry, the same reasoning can be applied to β_2 .

Variable variance

We have been using the *linear motif* to model the mean of a distribution and, in the previous section, we used it to model interactions. We can also use it to model the variance (or standard deviation) when the assumptions of constant variance do not make sense. For those cases, we may want to consider the variance as a (linear) function of the independent variable.

The **World Health Organization (WHO)** and other health institutions around the world collect data for newborns and toddlers and design growth charts standards. These charts are an essential component of the paediatric toolkit and also a measure of the general well-being of populations in order to formulate health-related policies, plan interventions, and monitor their effectiveness (<http://www.who.int/childgrowth/en/>).

An example of such data is the lengths (heights) of newborn/toddlers girls as a function of the age (in months):

```
data = pd.read_csv('../data/babies.csv')
data.plot.scatter('Month', 'Length')
```

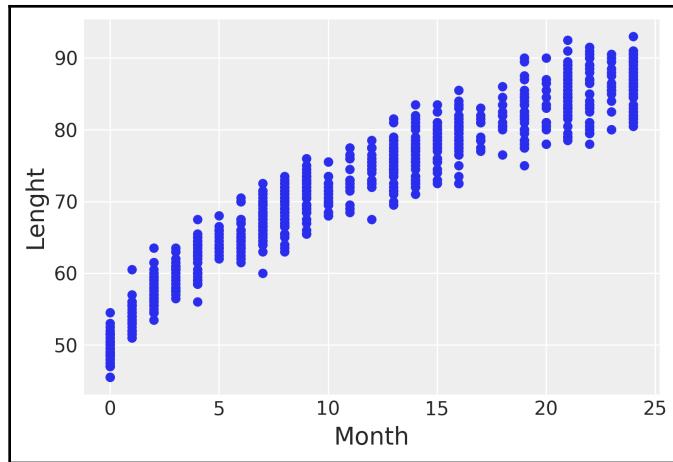


Figure 3.28

To model this data, we are going to introduce three new elements compared to the previous models:

- ϵ is now a linear function of x . To do this, we add two new parameters, γ and δ . These are direct analogs of α and β .
- The linear model for the mean is a function of \sqrt{x} . This is just a simple trick to fit a linear model to a curve.
- We define a shared variable, x_{shared} . We will use it to change the values of the x variable (Month, in this example), after model fitting without the need to refit the model. Why we want to do this will become clear soon:

```
with pm.Model() as model_vv:
    α = pm.Normal('α', sd=10)
    β = pm.Normal('β', sd=10)
    γ = pm.HalfNormal('γ', sd=10)
    δ = pm.HalfNormal('δ', sd=10)

    x_shared = shared(data.Month.values * 1.)

    μ = pm.Deterministic('μ', α + β * x_shared**0.5)
    ε = pm.Deterministic('ε', γ + δ * x_shared)

    y_pred = pm.Normal('y_pred', mu=μ, sd=ε, observed=data.Length)
    trace_vv = pm.sample(1000, tune=1000)
```

Figure 3.29 shows the result of our model. The mean of μ is represented with a black curve, and two semitransparent orange bands represent 1 and 2 standard deviations, respectively:

```
plt.plot(data.Month, data.Length, 'C0.', alpha=0.1)

μ_m = trace_vv['μ'].mean(0)
ε_m = trace_vv['ε'].mean(0)

plt.plot(data.Month, μ_m, c='k')
plt.fill_between(data.Month, μ_m + 1 * ε_m, μ_m -
                  1 * ε_m, alpha=0.6, color='C1')
plt.fill_between(data.Month, μ_m + 2 * ε_m, μ_m -
                  2 * ε_m, alpha=0.4, color='C1')

plt.xlabel('x')
plt.ylabel('y', rotation=0)
```

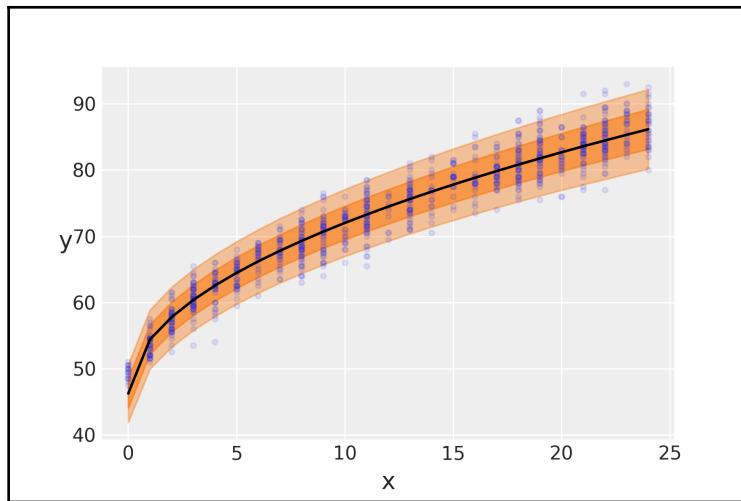


Figure 3.29

At the time of writing this book, my daughter is two weeks old (≈ 0.5 months), and thus I wonder how her length compares to the growth chart we have just created. One way to answer this question is to ask the model for the distribution of the variable length for babies of 0.5 months. Using PyMC3, we can ask this question with the `sample_posterior_predictive` function.

The output of this function will be samples of y conditioned on the observed data and the estimated distribution of parameters—that includes uncertainties. The only *problem* is that, by definition, this function will return predictions for y for the observed values of x , and 0.5 months (the value I care about) has not been observed; all measures are reported for integer months. The easier way to get predictions for non-observed values of x is to define a shared variable (as part of the model) and then update the value of the shared variable right before sampling from the posterior predictive distribution:

```
x_shared.set_value([0.5])
ppc = pm.sample_posterior_predictive(trace_vv, 2000, model=model_vv)
y_ppc = ppc['y_pred'][:, 0]
```

Now, we can plot the expected distribution of lengths for two weeks old babies and compute additional quantities, for example, the percentile of a child, given her length. Check the following block of code and *Figure 3.30* for such an example:

```
ref = 47.5
density, l, u = az._fast_kde(y_ppc)
x_ = np.linspace(l, u, 200)
plt.plot(x_, density)
percentile = int(sum(y_ppc <= ref) / len(y_ppc) * 100)
plt.fill_between(x_[x_ < ref], density[x_ < ref],
                 label='percentile = {:.2d}'.format(percentile))
plt.xlabel('length')
plt.yticks([])
plt.legend()
```

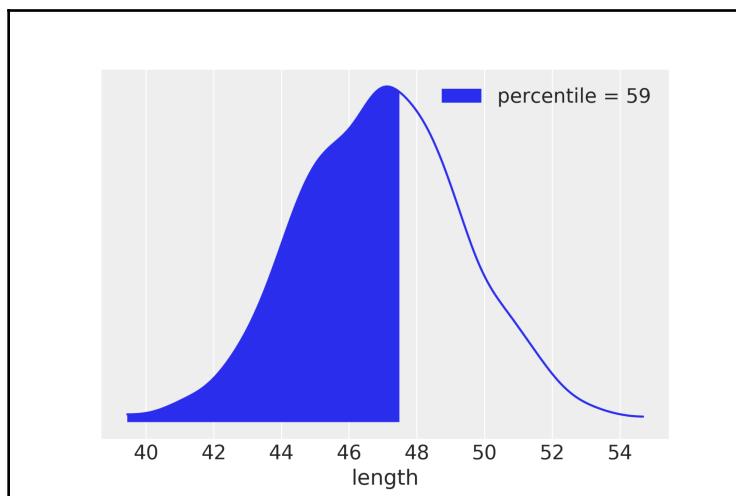


Figure 3.30

Summary

A simple linear regression is a model that can be used to predict and/or explain one variable from another one. Using machine learning language, this is a case of supervised learning. From a probabilistic perspective, a linear regression model is an extension of the Gaussian model where the mean is not directly estimated but rather computed as a linear function of a predictor variable and some additional parameters. While the Gaussian distribution is the most common choice for the dependent variable, we are free to choose other distributions. One alternative, which is especially useful when dealing with potential outliers, is the Student's t-distribution. In the next chapter, we will explore other alternatives.

In this chapter, we also discussed the Pearson correlation coefficient, the most common measure of linear correlation between two variables, and we learned to compute a Bayesian version of it from the data and posterior predictive samples using a multivariate Gaussian distribution. One useful way to expand a linear regression model is by doing a hierarchical version of it, which provides the benefits of shrinkage. This is very simple to achieve with PyMC3. We also briefly discussed the importance of not interpreting correlation as causation, at least in the absence of a mechanistic model. As surprising as it may sound, we can use linear models to fit curves. We showed this with two examples; a polynomial regression, and by taking the square root of the independent variable. Another extension for simple linear regression is to deal with more than one independent variable with what is usually called multiple linear regression. Some precautions are necessary to avoid errors and problems interpreting these type of models, and we used a few examples to demonstrate this. Other ways to use the linear motif is to model interactions, and yet another one is to deal with a non-constant variance for the dependent variable.

Exercises

1. Check the following definition of a probabilistic model. Identify the likelihood, the prior, and the posterior:

$$\begin{aligned}y_i &\sim \text{Normal}(\mu, \sigma) \\ \mu &\sim \text{Normal}(0, 10) \\ \sigma &\sim |\text{Normal}(0, 25)|\end{aligned}$$

2. For the model in exercise 1, how many parameters have the posterior? In other words, how many dimensions does it have?
3. Write down Bayes' theorem for the model in exercise 1.

4. Check the following model. Identify the linear model and identify the likelihood. How many parameters does the posterior have?

$$\begin{aligned}y &\sim \text{Normal}(\mu, \epsilon) \\ \mu &= \alpha + \beta x \\ \alpha &\sim \text{Normal}(0, 10) \\ \beta &\sim \text{Normal}(0, 1) \\ \epsilon &\sim |\text{Normal}(0, 25)|\end{aligned}$$

5. For the model in exercise 1, assume that you have a dataset with 57 data points coming from a Gaussian with a mean of 4 and a standard deviation of 0.5. Using PyMC3, compute:
- The posterior distribution
 - The prior distribution
 - The posterior predictive distribution
 - The prior predictive distribution

Tip: Besides `pm.sample()`, PyMC3 has other functions to compute samples.

6. Execute `model_g` using NUTS (the default sampler) and then using Metropolis. Compare the results using ArviZ functions like `plot_trace` and `plot_pair`. Center the variable `x` and repeat the exercise. What conclusion can you draw from this?
7. Using the `howell` dataset (available at <https://github.com/aloctavodia/BAP>) to create a linear model of the weight (`x`) against the height (`y`). Exclude subjects that are younger than 18. Explain the results.
8. For four subjects, we get the weights (45.73, 65.8, 54.2, 32.59), but not their heights. Using the model from the previous exercise, predict the height for each subject, together with their HPDs of 50% and 94%.

Tip1: Check coal mining disaster example in the PyMC3 documentation.

Tip2: Use shared variables

9. Repeat exercise 7, this time including those below 18 years old. Explain the results.

10. It is known for many species that the weight does not scale with the height, but with the logarithm of the weight. Use this information to fit the `howell` data (including subjects from all ages). Do one more model, this time without using the logarithm but instead a second order polynomial. Compare and explain both results.
11. Think about a model that's able to fit the three first datasets from the Anscombe quartet. Also think about a model to fit the fourth dataset.
12. See in the accompanying code the `model_t2` (and the data associated with it). Experiment with priors for ν , like the non-shifted exponential and gamma priors (they are commented in the code). Plot the prior distribution to ensure that you understand them. An easy way to do this is to just comment the likelihood in the model and check the trace plot. A more efficient way is to use the `pm.sample_prior_predictive()` function instead of `pm.sample()`.
13. For the `unpooled_model`, change the value of `sd` of the β prior; try values of 1 and 100. Explore how the estimated slopes change for each group. Which group is more affected by this change?
14. Using model `hierarchical_model` repeat *Figure 3.17*, the one with the eight groups and the eight lines, but this time add the uncertainty to the linear fit.
15. Re-run the `model_mlr` example, this time without centering the data. Compare the uncertainty in the α parameter for one case and the other. Can you explain these results?

Tip: Remember the definition of the α parameter (also known as *the intercept*).

16. Read and run the following notebook from PyMC3's documentation: <https://pymc-devs.github.io/pymc3/notebooks/LKJ.html>.
17. Choose a dataset that you find interesting and use it with the simple linear regression model. Be sure to explore the results using ArviZ functions and compute the Pearson correlation coefficient. If you do not have an interesting dataset, try searching online, for example, at <http://data.worldbank.org/> or <http://www.stat.ufl.edu/~winner/datasets.html>.

4

Generalizing Linear Models

We think in generalities, but we live in detail.

- Alfred North Whitehead

In the last chapter, we used a linear combination of input variables to predict the mean of an output variable. We assumed the latter to be distributed as a Gaussian. Using a Gaussian works in many situations, but for many other it could be wiser to choose a different distribution; we already saw an example of this when we replaced the Gaussian distribution with a Student's t-distribution. In this chapter, we will see more examples where it is wise to use distributions other than Gaussian. As we will learn, there is a general motif, or pattern, that can be used to generalize the linear model to many problems.

In this chapter, we will explore:

- Generalized linear models
- Logistic regression and inverse link functions
- Simple logistic regression
- Multiple logistic regression
- The softmax function and the multinomial logistic regression
- Poisson regression
- Zero-inflated Poisson regression

Generalized linear models

One of the core ideas of this chapter is rather simple: in order to predict the mean of an output variable, we can apply an arbitrary function to a linear combination of input variables.

$$\mu = f(\alpha + X\beta) \quad (4.1)$$

Where f is a function, we will call **inverse link function**. There are many inverse link functions we can choose; probably the simplest one is the identity function. This is a function that returns the same value used as its argument. All models from Chapter 3, *Modeling with Linear Regression* used the identity function, and for simplicity we just omit it. The identity function may not be very useful on its own, but it allows us to think of several different models in a more unified way.



Why do we call f , the inverse link function, instead of just the link function? Because traditionally people apply functions to the other side of equation 4.1, and unfortunately for us, they already *called dibs* on the term link function—so to avoid confusion, we are going to stick to the term inverse link function.

One situation under which we would like to use an inverse link function is when working with categorical variables, such as color names, gender, biological species, or political party/affiliation. None of these variables is well-modeled by Gaussians. Think about it, in principle, a Gaussian works well for a continuous variable taking any value on the real line, while the variables mentioned here are discrete and only take a few values (such as red, green, or blue). If we change the distribution we used to model the data, we will in general need to also change how we model the plausible values for the mean of those distributions. For example, if we use a binomial distribution, like in Chapter 1, *Thinking Probabilistically* and Chapter 2, *Programming Probabilistically*, we will need a linear model that returns a mean value in the $[0, 1]$ interval; one way to achieve this is to keep the linear model but use an inverse link function to restrict the output to the desired interval. This trick is not restricted to discrete variables; we may want to model data that can only take positive values, and thus we may want to restrict the linear model to return positive values for the mean of a distribution, such as Gamma or exponential.

Just before moving on, note that some variables can be codified as quantitative or as qualitative, and that this is a decision you have to make based on the context of your problem; for example, we can talk about the red and green categorical variables if we are talking about color names, or we can talk about the 650 nm and 510 nm continuous variables if we are working with wavelengths.

Logistic regression

Regression problems are about predicting a continuous value for an output variable given the values of one or more input variables. Instead, classification is about assigning a discrete value (representing a discrete class) to an output variable given some input variables. In both cases, the task is to get a model that properly models the mapping between output and input variables; in order to do so, we have at our disposal a sample with *correct* pairs of output-input variables. From a **machine learning** perspective, both regressions and classifications are instances of supervised learning algorithms.

My mother prepares a delicious dish called **sopa seca**, which is basically a spaghetti-based recipe and literally means *dry soup*. While it may sound like a misnomer or even an oxymoron, the name of the dish makes total sense when we learn how it is cooked. Something similar happens with logistic regression, a model that, despite its name, is generally framed as a method to solve classification problems.

The logistic regression model is a generalization of the linear regression model from Chapter 3, *Modeling with Linear Regression*, and thus its name. We achieve this generalization by replacing f in 4.1 with the logistic function as an inverse link function:

$$\text{logistic}(z) = \frac{1}{1 + e^{-z}} \tag{4.2}$$

For our purpose, the key property of the logistic function is that irrespective of the values of its argument, z , the result will always be a number in the [0-1] interval. Thus, we can see this function as a convenient way to compress the values computed from a linear model into values that we can feed into a Bernoulli distribution. This logistic function is also known as the sigmoid function, because of its characteristic S-shaped aspect, as we can see by executing the next few lines:

```
z = np.linspace(-8, 8)
plt.plot(z, 1 / (1 + np.exp(-z)))
plt.xlabel('z')
plt.ylabel('logistic(z)')
```

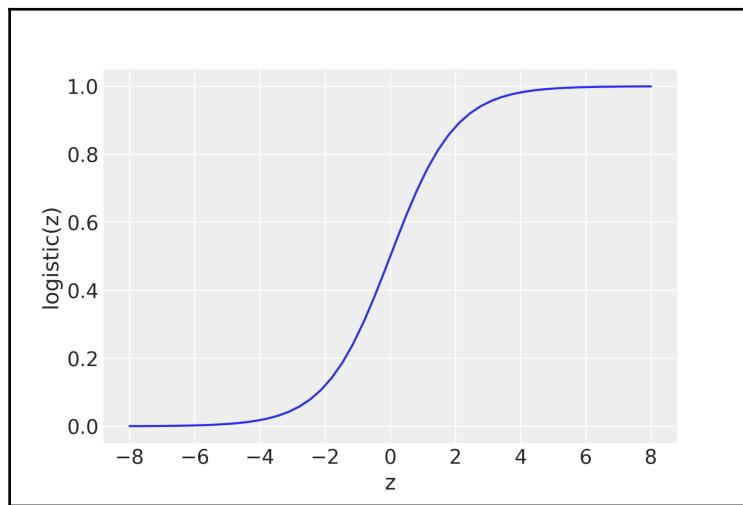


Figure 4.1

The logistic model

We have almost all the elements to turn a simple linear regression into a simple logistic regression. Let's begin with the case of only two classes or instances, for example ham/spam, safe/unsafe, cloudy/sunny, healthy/ill, or hotdog/not hotdog. First, we codify these classes by saying that the predicted variable, y , can only take two values, 0 or 1, that is, $y \in \{0, 1\}$. Stated this way, the problem sounds similar to the coin-flipping example from Chapter 2, *Programming Probabilistically* and Chapter 3, *Modeling with Linear Regression*.

We may remember we used the Bernoulli distribution as the likelihood. The difference with the coin-flipping problem is that now θ is not going to be generated from a beta distribution; instead, θ is going to be defined by a linear model with the logistic as the inverse link function. Omitting the priors we have:

$$\begin{aligned}\theta &= \text{logistic}(\alpha + x\beta) \\ y &= \text{Bern}(\theta)\end{aligned}\tag{4.3}$$



Notice that the main difference with the simple linear regression from Chapter 3, *Modeling with Linear Regression* is the use of a Bernoulli distribution instead of a Gaussian distribution, and the use of the logistic function instead of the identity function.

The Iris dataset

We are going to apply logistic regression to the iris dataset. This is a classic dataset containing information about flowers from three closely related species: setosa, virginica, and versicolor. These are going to be our dependent variables, that is, the classes we want to predict. We have 50 individual cases of each species and for each individual case, the dataset contains four variables that we are going to use as the independent variables (or features): petal length, petal width, sepal length, and sepal width. In case you are wondering, sepals are modified leaves whose function is generally related to protecting the flowers in a bud. We can load a data frame with the iris dataset by doing the following:

```
iris = pd.read_csv('~/data/iris.csv')
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Now, we will plot the three species versus `sepal_length` using the `stripplot` function from `seaborn`:

```
sns.stripplot(x="species", y="sepal_length", data=iris, jitter=True)
```

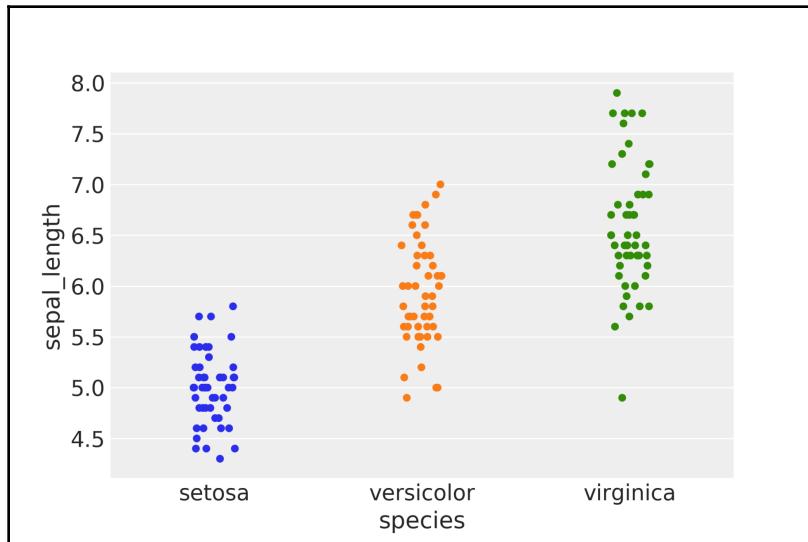


Figure 4.2

In *Figure 4.2*, the *y* axis is continuous while the *x* axis is categorical; the dispersion (or jitter) of the points along the *x* axis has no meaning at all, it is just a trick we add with the `jitter` argument to avoid having all the points collapsed onto a single line. Try setting the `jitter` argument to `False` to see what I mean. The only thing that matters when reading the *x* axis is the membership of the points to the `setosa`, `versicolor`, or `virginica` classes. You may also try other plots for this data, such as violin plots, which are also available as one-liners with `seaborn`.

Another way to inspect the data is by doing a scatter matrix with the `pairplot` function. We have scatter plots arranged in a 4×4 grid, since we have four features in the iris dataset. The grid is symmetrical, with the upper and lower triangles showing the same information. The scatter plot on the main diagonal should correspond to a variable against itself; given that such a plot is not informative at all, we have replaced those scatter plots with a kde for each feature. Inside each subplot, we have the three species (or classes) represented with a different color, which is the same as used in *Figure 4.2*:

```
sns.pairplot(iris, hue='species', diag_kind='kde')
```

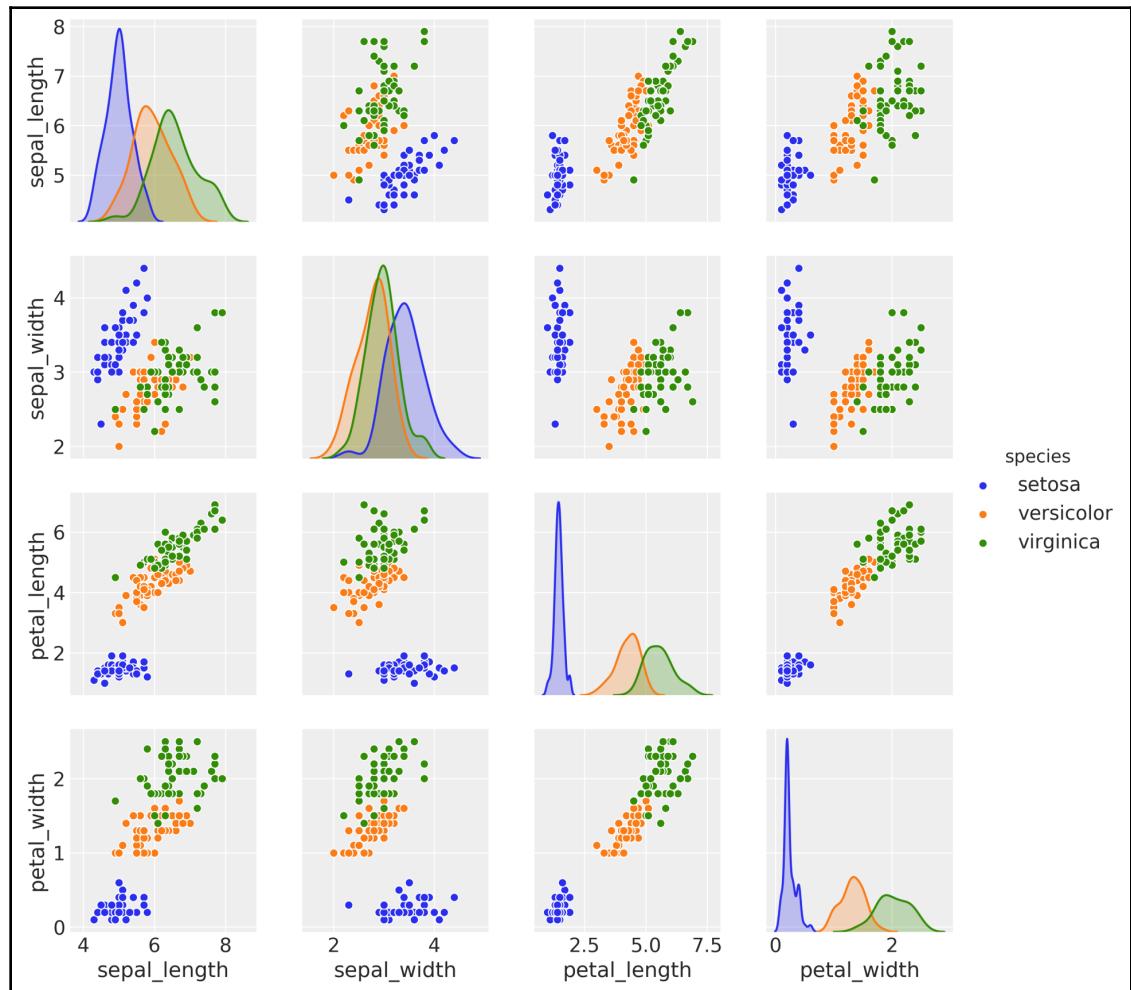


Figure 4.3

Before continuing, take some time to study *Figure 4.3* and try to get familiar with the iris dataset and how the features and classes are related.

The logistic model applied to the iris dataset

We are going to begin with the simplest possible classification problem: two classes, `setosa` and `versicolor`, and just one independent variable or feature, the `sepal_length`. As it is usually done, we are going to encode the `setosa` and `versicolor` categorical variables with the numbers 0 and 1. Using pandas, we can do the following:

```
df = iris.query("species == ('setosa', 'versicolor')")
y_0 = pd.Categorical(df['species']).codes
x_n = 'sepal_length'
x_0 = df[x_n].values
x_c = x_0 - x_0.mean()
```

As with other linear models, centering the data can help with the sampling. Now that we have the data in the proper format, we can finally build the model with PyMC3.

Notice how the first part of `model_0` resembles a linear regression model. Also pay attention to the two deterministic variables: θ and `bd`. θ is the output of the logistic function applied to the μ variable, and `bd` is the boundary decision, which is the value used to separate classes; we will discuss this later in detail. Another point worth mentioning is that instead of explicitly writing the logistic function, we are using `pm.math.sigmoid` (this is just an alias for the Theano function with the same name):

```
with pm.Model() as model_0:
    α = pm.Normal('α', mu=0, sd=10)
    β = pm.Normal('β', mu=0, sd=10)
    μ = α + pm.math.dot(x_c, β)
    θ = pm.Deterministic('θ', pm.math.sigmoid(μ))
    bd = pm.Deterministic('bd', -α/β)
    yl = pm.Bernoulli('yl', p=θ, observed=y_0)

    trace_0 = pm.sample(1000)
```

In order to save pages and avoid you getting bored with the same type of plot over and over again, we are going to omit doing a trace plot and other similar summaries, but I encourage you make your own plots and summaries to further explore the examples in the book. Instead, we are going to jump directly to generating *Figure 4.1*, a plot of the data, together with the fitted sigmoid curve and the decision boundary:

```

theta = trace_0['θ'].mean(axis=0)
idx = np.argsort(x_c)
plt.plot(x_c[idx], theta[idx], color='C2', lw=3)
plt.vlines(trace_0['bd'].mean(), 0, 1, color='k')
bd_hpd = az.hpd(trace_0['bd'])
plt.fill_betweenx([0, 1], bd_hpd[0], bd_hpd[1], color='k', alpha=0.5)

plt.scatter(x_c, np.random.normal(y_0, 0.02),
            marker='.', color=[f'C{x}' for x in y_0])
az.plot_hpd(x_c, trace_0['θ'], color='C2')

plt.xlabel(x_n)
plt.ylabel('θ', rotation=0)
# use original scale for xticks
locs, _ = plt.xticks()
plt.xticks(locs, np.round(locs + x_0.mean(), 1))

```

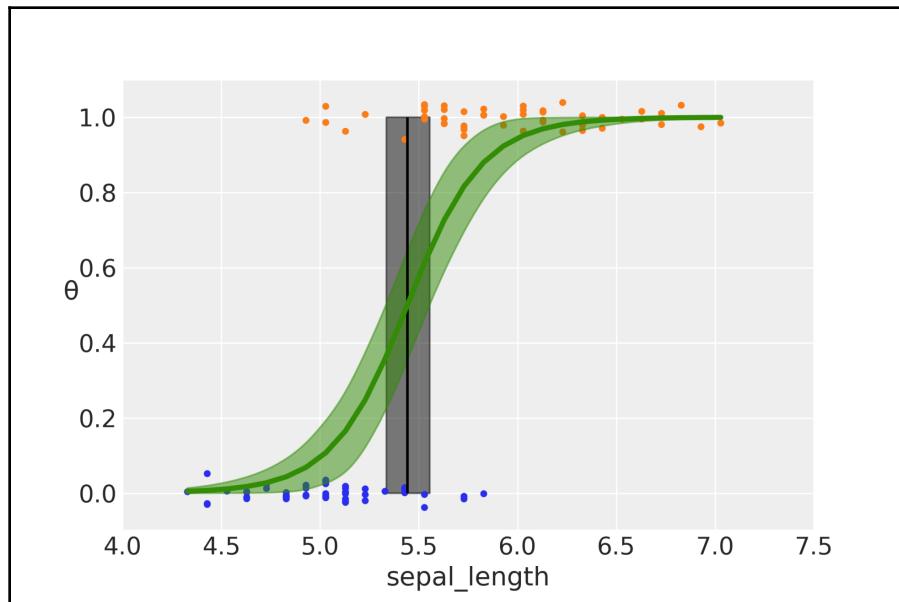


Figure 4.4

Figure 4.4 shows the sepal length versus the iris species (setosa = 0, versicolor = 1). To avoid over-plotting, the binary response variables are jittered. An S-shaped (green) line is the mean value of θ . This line can be interpreted as the probability of a flower being versicolor, given that we know the value of the sepal length. The semitransparent S-shaped (green) band is the 94% HPD interval. The boundary decision is represented as a (black) vertical line with a semi-transparent band for its 94% HPD. According to the boundary decision, the x_i values (sepal lengths, in this case) to the left correspond to class 0 (setosa), and the values to the right to class 1 (versicolor).

The decision boundary is defined as the value of x_i , for which $y = 0.5$. And it turns out to be $-\frac{\alpha}{\beta}$, which we can derive as follows:

- From the definition of the model, we have the following relationship:

$$\theta = \text{logistic}(\alpha + x_i\beta) \quad (4.4)$$

- And from the definition of the logistic function, we have $\theta = 0.5$ when the argument of the logistic regression is 0:

$$0.5 = \text{logistic}(\alpha + x_i\beta) \Leftrightarrow 0 = \alpha + x_i\beta \quad (4.5)$$

- By reordering equation 4.5, we find that the value of x_i , for which $\theta = 0.5$, corresponds to the following expression:

$$x_i = -\frac{\alpha}{\beta} \quad (4.6)$$

There are a few key points to mention:

- The value of θ is, generally speaking, $p(y = 1 | x)$. In this sense, the logistic regression is a true regression; the key detail is that we are regressing the probability that a data point belongs to class 1, given a linear combination of features.
- We are modeling the mean of a dichotomous variable, that is, a number in the [0-1] interval. Then, we introduce a rule to turn this probability into a two-class assignment. In this case, if $p(y = 1) \geq 0.5$, we assign class 1, otherwise class 0.

- There is nothing special about the value of 0.5, other than that it is the number in the middle of 0 and 1. We may argue this boundary is only reasonable if we are OK making a mistake in either one direction or the other—in other words, if it is the same for us to misclassify a setosa as a versicolor or a versicolor as a setosa. It turns out that this is not always the case, and the cost associated with the misclassification does not need to be symmetrical, as you may remember from Chapter 2, *Programming Probabilistically*, when we discussed loss functions.

Multiple logistic regression

In a similar fashion to multiple linear regression, multiple logistic regression is about using more than one independent variable. Let's try combining the sepal length and the sepal width. Remember we need to pre-process the data a little bit:

```
df = iris.query("species == ('setosa', 'versicolor')")
y_1 = pd.Categorical(df['species']).codes
x_n = ['sepal_length', 'sepal_width']
x_1 = df[x_n].values
```

The boundary decision

Feel free to skip this section and jump to the model implementation (next section) if you are not too interested in how we can derive the boundary decision.

From the model, we have the following equation:

$$\theta = \text{logistic}(\alpha + \beta_1 x_1 + \beta_2 x_2) \quad (4.7)$$

And from the definition of the logistic function, we have $\theta = 0.5$, when the argument of the logistic regression is zero:

$$0.5 = \text{logistic}(\alpha + \beta_1 x_1 + \beta_2 x_2) \Leftrightarrow 0 = \alpha + \beta_1 x_1 + \beta_2 x_2 \quad (4.8)$$

By reordering, we find the value of x_2 for which $\theta = 0.5$ corresponds to the following expression:

$$x_2 = -\frac{\alpha}{\beta_2} + \left(-\frac{\beta_1}{\beta_2} x_1 \right) \quad (4.9)$$

This expression for the boundary decision has the same mathematical form as a line equation, with the first term being the intercept and the second the slope. The parentheses are used for clarity and we can omit them if we want. The boundary being a line is totally reasonable, isn't it? If we have one feature, we have unidimensional data and we can split it into two groups using a point; if we have two features, we have a two-dimensional data space and we can separate it using a line; for three dimensions, the boundary will be a plane; and for higher dimensions, we will talk generically about hyperplanes. In fact, a hyperplane is a general concept defined roughly as the subspace of dimension $n-1$ of an n -dimensional space, so we can always talk about hyperplanes!

Implementing the model

To write the multiple logistic regression model using PyMC3, we take advantage of its vectorization capabilities, allowing us to introduce only minor modifications to the previous simple logistic model (`model_0`):

```
with pm.Model() as model_1:
    α = pm.Normal('α', mu=0, sd=10)
    β = pm.Normal('β', mu=0, sd=2, shape=len(x_n))
    μ = α + pm.math.dot(x_1, β)
    θ = pm.Deterministic('θ', 1 / (1 + pm.math.exp(-μ)))
    bd = pm.Deterministic('bd', -α/β[1] - β[0]/β[1] * x_1[:, 0])
    y1 = pm.Bernoulli('y1', p=θ, observed=y_1)

    trace_1 = pm.sample(2000)
```

As we did for a single predictor variable, we are going to plot the data and the decision boundary:

```
idx = np.argsort(x_1[:, 0])
bd = trace_1['bd'].mean(0)[idx]
plt.scatter(x_1[:, 0], x_1[:, 1], c=[f'C{x}' for x in y_0])
plt.plot(x_1[:, 0][idx], bd, color='k');

az.plot_hpd(x_1[:, 0], trace_1['bd'], color='k')

plt.xlabel(x_n[0])
plt.ylabel(x_n[1])
```

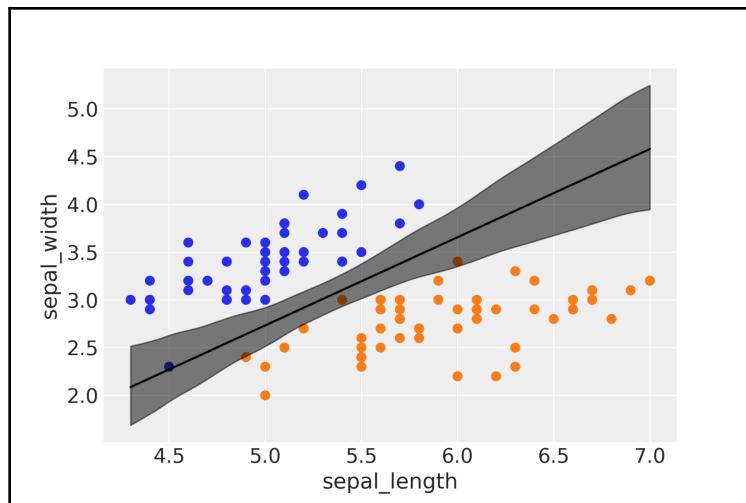


Figure 4.5

The boundary decision is a straight line, as we have already seen. Do not get confused by the curved aspect of the 94% HPD band. The apparent curvature is the result of having multiple lines *pivoting* around a central region (roughly around the mean of x and the mean of y).

Interpreting the coefficients of a logistic regression

We must be careful when interpreting the β coefficients of a logistic regression. Interpretation is not as straightforward as with linear models, which we looked at in Chapter 3, *Modeling with Linear Regression*. Using the logistic inverse link function introduces a non-linearity that we have to take into account. If β is positive, increasing x will increase $p(y = 1)$ by some amount, but the amount is not a linear function of x ; instead, it depends *non-linearly* on the value of x . We can visualize this fact in Figure 4.4; instead of a line with a constant slope, we have an S-shaped line with a slope that changes as a function of x . A little bit of algebra can give us some further insight into how much $p(y = 1)$ changes with β :

The basic logistic model is:

$$\theta = \text{logistic}(\alpha + X\beta) \quad (4.11)$$

The inverse of the logistic is the logit function, which is:

$$\text{logit}(z) = \log\left(\frac{z}{1-z}\right) \quad (4.12)$$

Thus, if we take the first equation in this section and apply the logit function to both terms, we get this equation:

$$\text{logit}(\theta) = \alpha + X\beta \quad (4.13)$$

Or equivalently:

$$\log\left(\frac{\theta}{1-\theta}\right) = \alpha + X\beta \quad (4.14)$$

Remember that θ in our model is $p(y=1)$:

$$\log\left(\frac{p(y=1)}{1-p(y=1)}\right) = \alpha + X\beta \quad (4.15)$$

The $\frac{p(y=1)}{1-p(y=1)}$ quantity is known as the **odds**.

The odds of success are defined as the ratio of the probability of success over the probability of not-success. While the probability of getting 2 by rolling a fair die is $1/6$, the odds for the same event are $\frac{1/6}{5/6} \simeq 0.2$ or one favorable event to five unfavorable events. Odds are often used by gamblers mainly because odds provide a more intuitive tool than raw probabilities when thinking about the proper way to bet.



In a logistic regression, the β coefficient encodes the increase in log-odds units by unit increase of the x variable.

The transformation from probability to odds is a monotonic transformation, meaning the odds increase as the probability increases, and the other way around. While probabilities are restricted to the $[0, 1]$ interval, odds live in the $[0, \infty)$ interval. The logarithm is another monotonic transformation and log-odds are in the $(-\infty, \infty)$ interval. *Figure 4.6* shows how probabilities are related to odds and log-odds:

```
probability = np.linspace(0.01, 1, 100)
odds = probability / (1 - probability)

_, ax1 = plt.subplots()
ax2 = ax1.twinx()
ax1.plot(probability, odds, 'C0')
ax2.plot(probability, np.log(odds), 'C1')

ax1.set_xlabel('probability')
ax1.set_ylabel('odds', color='C0')
ax2.set_ylabel('log-odds', color='C1')
ax1.grid(False)
ax2.grid(False)
```

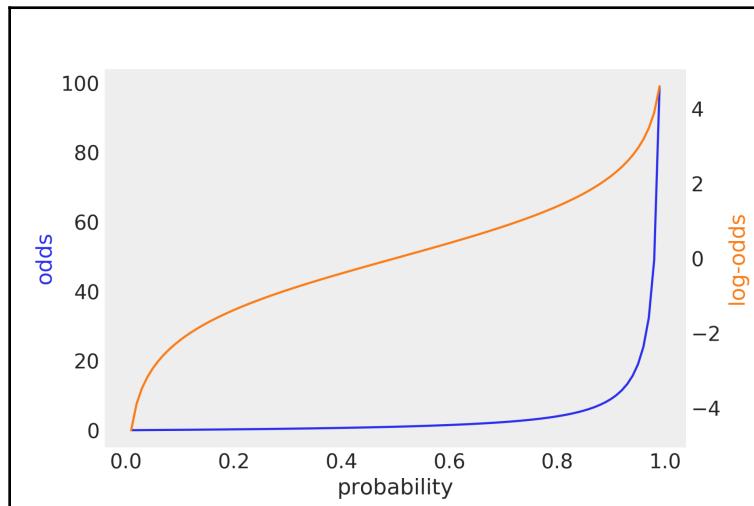


Figure 4.6

Thus, the values of the coefficients provided by `summary` are in the log-odds scale:

```
df = az.summary(trace_1, var_names=varnames)
df
```

	mean	sd	mc error	hpdi 3%	hpdi 97%	eff_n	r_hat
α	-9.12	4.61	0.15	-17.55	-0.42	1353.0	1.0
$\beta[0]$	4.65	0.87	0.03	2.96	6.15	1268.0	1.0
$\beta[1]$	-5.16	0.95	0.01	-7.05	-3.46	1606.0	1.0

One very pragmatic way of understanding models is to change parameters and see what happens. In the following block of code, we are computing the log-odds in favor of versicolor as $\text{log_odds_versicolor}_i = \alpha + \beta_1 x_1 + \beta_2 x_2$, and then the probability of versicolor with the logistic function. Then, we repeat the computation by fixing x_2 and increasing x_1 by 1:

```
x_1 = 4.5 # sepal_length
x_2 = 3   # sepal_width

log_odds_versicolor_i = (df['mean'] * [1, x_1, x_2]).sum()
probability_versicolor_i = logistic(log_odds_versicolor_i)

log_odds_versicolor_f = (df['mean'] * [1, x_1 + 1, x_2]).sum()
probability_versicolor_f = logistic(log_odds_versicolor_f)

log_odds_versicolor_f - log_odds_versicolor_i, probability_versicolor_f -
probability_versicolor_i
```

If you run the code, you will find that the increase in log-odds is ≈ 4.66 , which is exactly the value of β_0 (check the summary of `trace_1`). This is in line with our previous finding that β encodes the increase in log-odds units by unit increase of the x variable. The increase in probability is ≈ 0.70 .

Dealing with correlated variables

We know from Chapter 3, *Modeling with Linear Regression* that tricky things await us when we deal with (highly) correlated variables. Correlated variables translate into wider combinations of coefficients that are able to explain the data, or from a complementary point of view, correlated data has less power to restrict the model. A similar problem occurs when the classes become perfectly separable, that is, when there is no overlap between classes given the linear combination of variables in our model.

Using the `iris` dataset, you can try running `model_1`, but this time using the `petal_width` and `petal_length` variables. You will find that the β coefficients are broader than before, and also the 94% HPD black band in *Figure 4.5* is much wider:

```
corr = iris[iris['species'] != 'virginica'].corr()  
mask = np.tri(*corr.shape).T  
sns.heatmap(corr.abs(), mask=mask, annot=True, cmap='viridis')
```

Figure 4.7 is a heat map showing that for the `sepal_length` and `sepal_width` variables used in the first example, the correlation is not as high as the correlation between the `petal_length` and `petal_width` variables used in the second example:

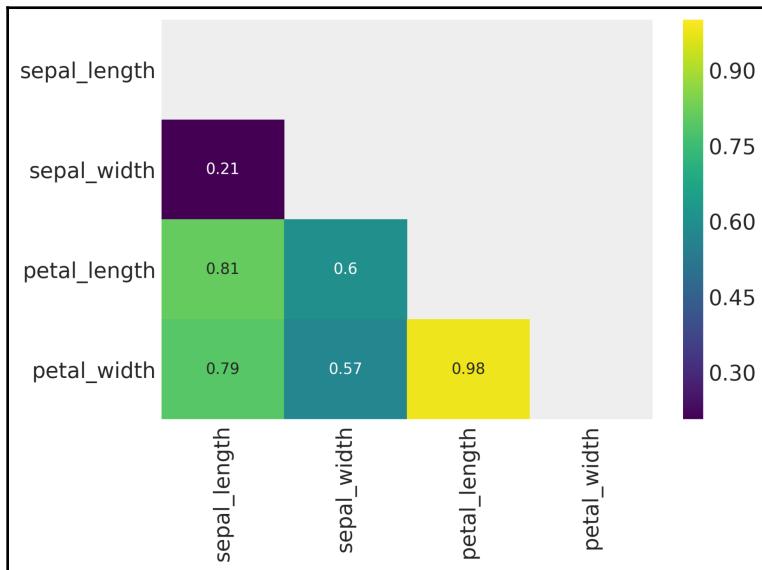


Figure 4.7

To generate *Figure 4.7*, we have used a mask to remove the upper triangle and the diagonal elements of the heat map, since these are uninformative, given the lower triangle. Also note that we have plotted the absolute value of the correlation, since at this moment we do not care about the sign of the correlation between variables, only about its strength.

One solution when dealing with (highly) correlated variables is to just remove one (or more than one) correlated variable. Another option is to put more information into the prior, this can be achieved using informative priors if we have useful information. For weakly-informative priors, Andrew Gelman and the Stan Team recommend (<https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations>) scaling all non-binary variables to have a mean of 0 and then using:

$$\beta \sim StudentT(0, \nu, sd) \quad (4.10)$$

Here, sd should be chosen in order to weakly inform us about the expected values for the scale. The normality parameter, ν , is suggested to be around 3-7. This prior is saying that, in general, we expect the coefficient to be small, but we use fat tails because occasionally we will find some larger coefficients. As we saw in [Chapter 2, Programming Probabilistically](#) and [Chapter 3, Modeling with Linear Regression](#), using a Student's t-distribution leads to a more robust model than using a Gaussian distribution.

Dealing with unbalanced classes

The iris dataset is completely balanced, in the sense that each category has exactly the same number of observations. We have 50 setosas, 50 versicolors, and 50 virginicas. This is something to thank Ronald Fisher for, unlike his dedication to popularizing the use of p-values.

On the contrary, many datasets consist of unbalanced data, that is, there are many more data points from one class than from another. When this happens, logistic regression can run into trouble, namely the boundary cannot be determined as accurately as when the dataset is more balanced.

To see an example of this behavior, we are going to use the iris dataset and we are going to arbitrarily remove some data points from the `setosa` class:

```
df = iris.query("species == ('setosa', 'versicolor')")
df = df[45:]
y_3 = pd.Categorical(df['species']).codes
x_n = ['sepal_length', 'sepal_width']
x_3 = df[x_n].values
```

And then, we are going to run a multiple logistic regression model, just as before:

```
with pm.Model() as model_3:
    α = pm.Normal('α', mu=0, sd=10)
    β = pm.Normal('β', mu=0, sd=2, shape=len(x_n))
    μ = α + pm.math.dot(x_3, β)
```

```

θ = 1 / (1 + pm.math.exp(-μ))
bd = pm.Deterministic('bd', -α/β[1] - β[0]/β[1] * x_3[:,0])
y1 = pm.Bernoulli('y1', p=θ, observed=y_3)

trace_3 = pm.sample(1000)

```

As we can see from *Figure 4.8*, the boundary decision is now shifted toward the less abundant class, and the uncertainty is larger than before. This is the typical behavior of a logistic model for unbalanced data. But wait a minute! You may be arguing that I am cheating here since the wider uncertainty could be the product of having less total data and not just fewer setosas than versicolors! That could be a valid point; try with *exercise 6* to verify that what explains this plot is the unbalanced data:

```

idx = np.argsort(x_3[:,0])
bd = trace_3['bd'].mean(0)[idx]
plt.scatter(x_3[:,0], x_3[:,1], c= [f'C{x}' for x in y_3])
plt.plot(x_3[:,0][idx], bd, color='k')

az.plot_hpd(x_3[:,0], trace_3['bd'], color='k')

plt.xlabel(x_n[0])
plt.ylabel(x_n[1])

```

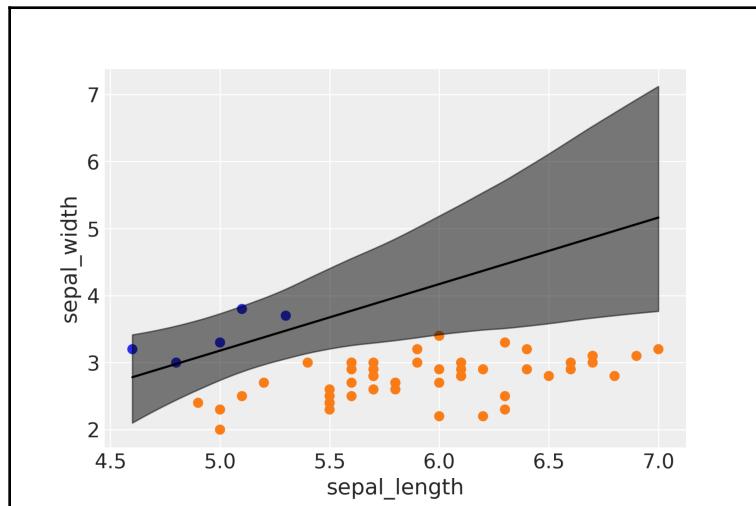


Figure 4.8

What do we do if we find unbalanced data? Well, the obvious solution is to get a dataset with roughly the same data points per class. This can be important to have in mind if you are collecting or generating the data. If you have no control over the dataset, you should be careful when interpreting the result for unbalanced data. Check the uncertainty of the model and run some posterior predictive checks to see whether the results are useful to you. Another option will be to input more prior information, if available, and/or run an alternative model, as explained later in this chapter.

Softmax regression

One way to generalize logistic regression to more than two classes is with **softmax regression**. We need to introduce two changes with respect to logistic regression; first, we replace the logistic function with the softmax function:

$$\text{softmax}_i(\mu) = \frac{\exp(\mu_i)}{\sum \exp(\mu_k)} \quad (4.16)$$

In other words, to obtain the output of the softmax function for the i -esim element of a vector, μ , we take the exponential of the i -esim value divided by the sum of all the exponentiated values in the μ vector.

Softmax guarantees we will get positive values that sum up to 1. The softmax function is reduced to the logistic function when $\kappa = 2$. As a side note, the softmax function has the same form as the **Boltzmann distribution** used in statistical mechanics, which is a very powerful branch of physics dealing with the probabilistic description of atomic and molecular systems. The Boltzmann distribution (and softmax, in some fields) has a parameter called temperature, T , that divides μ ; when $T \rightarrow \infty$, the probability distribution becomes flat and all states are equally likely, and when $T \rightarrow 0$, only the most probable state gets populated, and thus softmax behaves like a max function.

The second modification is that we replace the Bernoulli distribution with the categorical distribution. The categorical distribution is the generalization of the Bernoulli to more than two outcomes. Also, as the Bernoulli distribution (single coin flip) is a special case of the Binomial (n coin flips), the categorical (single roll of a die) is a special case of the multinomial distribution (n rolls of a die). You may try this brain teaser with your nieces and nephews!

To exemplify the softmax regression, we are going to continue working with the iris dataset, only this time we are going to use its three classes (setosa, versicolor, and virginica) and its four features (sepal length, sepal width, petal length, and petal width). We are also going to standardize the data, since this will help the sampler to run more efficiently (we could have also just centered the data):

```
iris = sns.load_dataset('iris')
y_s = pd.Categorical(iris['species']).codes
x_n = iris.columns[:-1]
x_s = iris[x_n].values
x_s = (x_s - x_s.mean(axis=0)) / x_s.std(axis=0)
```

The PyMC3 code reflects the changes between the logistic and softmax models. Notice the shapes of the α and β coefficients. We use the softmax function from Theano; we have used the import theano.tensor as tt idiom, which is the convention used by PyMC3 developers:

```
with pm.Model() as model_s:
    α = pm.Normal('α', mu=0, sd=5, shape=3)
    β = pm.Normal('β', mu=0, sd=5, shape=(4, 3))
    μ = pm.Deterministic('μ', α + pm.math.dot(x_s, β))
    θ = tt.nnet.softmax(μ)
    yl = pm.Categorical('yl', p=θ, observed=y_s)
    trace_s = pm.sample(2000)
```

How well does our model perform? Let's find out by checking how many cases we can predict correctly. In the following code, we just use the mean of the parameters to compute the probability of each data point belonging to each of the three classes, then we assign the class by using the argmax function. And we compare the result with the observed values:

```
data_pred = trace_s['μ'].mean(0)
y_pred = [np.exp(point)/np.sum(np.exp(point)), axis=0]
for point in data_pred]
f'{np.sum(y_s == np.argmax(y_pred, axis=1)) / len(y_s):.2f}'
```

The result is that we classify $\sim 98\%$ of the data points correctly, that is, we miss only three cases. This is very good. Nevertheless, a true test to evaluate the performance of our model will be to check it on data not used to fit the model. Otherwise, we may be overestimating the ability of the model to generalize to other data. We will discuss this subject in detail in Chapter 5, *Model Comparison*. For now, we will leave this just as an auto-consistency test indicating that the model runs OK.

You may have noticed that the posterior, or more properly the marginal distributions of each parameter, are very wide; in fact, they are as wide as indicated by the priors. Even when we were able to make correct predictions, this does not look OK. This is the same non-identifiability problem we already encountered for correlated data in other regression models or with perfectly separable classes. In this case, the wide posterior is due to the condition that all probabilities must sum to one. Given this condition, we are using more parameters than we need to fully specify the model. In simple terms, if you have ten numbers that sum to one, you just need to give me nine of them; the other I can compute. One solution is to fix the extra parameters to some value, for example, zero. The following code shows how to achieve this using PyMC3:

```
with pm.Model() as model_sf:
    α = pm.Normal('α', mu=0, sd=2, shape=2)
    β = pm.Normal('β', mu=0, sd=2, shape=(4, 2))
    α_f = tt.concatenate([[0], α])
    β_f = tt.concatenate([np.zeros((4, 1)), β], axis=1)
    μ = α_f + pm.math.dot(x_s, β_f)
    θ = tt.nnet.softmax(μ)
    yl = pm.Categorical('yl', p=θ, observed=y_s)
    trace_sf = pm.sample(1000)
```

Discriminative and generative models

So far, we have discussed logistic regression and a few extensions of it. In all cases, we tried to directly compute $p(y | x)$, that is, the probability of a given class knowing x , with x being feature(s) we measured to members of the classes. In other words, we try to directly model the mapping from the independent variables to the dependent ones, and then use a threshold to turn the *continuous* computed probability into a *discrete* boundary that allows us to assign classes.

This approach is not unique. One alternative is to first model $p(x | y)$, that is, the distribution of x for each class, and then assign the classes. This kind of model is called a **generative classifier**, because we are creating a model from which we can *generate* samples from each class. On the other hand, logistic regression is a type of **discriminative** classifier, since it tries to classify by *discriminating* classes but we cannot generate examples from each class from the model. We are not going to go into much detail here about generative models for classification, but we are going to see one example that illustrates the core of this type of model for classification. We are going to do it for two classes and only one feature, exactly as the first model we built in this chapter (`model_0`), and we are going to use the very same data.

The following is a PyMC3 implementation of a generative classifier. From the code, you can see that now the boundary decision is defined as the average between the estimated Gaussian means. This is the correct boundary decision when the distributions are normal and their standard deviations are equal. These are the assumptions made by a model known as **linear discriminant analysis (LDA)**. Despite its name, the LDA model is generative:

```
with pm.Model() as lda:
    mu = pm.Normal('mu', mu=0, sd=10, shape=2)
    sigma = pm.HalfNormal('sigma', 10)
    setosa = pm.Normal('setosa', mu=mu[0], sd=sigma, observed=x_0[:50])
    versicolor = pm.Normal('versicolor', mu=mu[1], sd=sigma,
                           observed=x_0[50:])
    bd = pm.Deterministic('bd', (mu[0] + mu[1]) / 2)
    trace_lda = pm.sample(1000)
```

Now, we are going to plot a figure showing the two classes ($\text{setosa} = 0$ and $\text{versicolor} = 1$) against the values for sepal length, and also the boundary decision as a red line and the 94% **Highest-Posterior Density (HPD)** interval for it as a semitransparent red band:

```
plt.axvline(trace_lda['bd'].mean(), ymax=1, color='C1')
bd_hpd = az.hpd(trace_lda['bd'])
plt.fill_betweenx([0, 1], bd_hpd[0], bd_hpd[1], color='C1', alpha=0.5)

plt.plot(x_0, np.random.normal(y_0, 0.02), '.', color='k')
plt.ylabel('theta', rotation=0)
plt.xlabel('sepal_length')
```

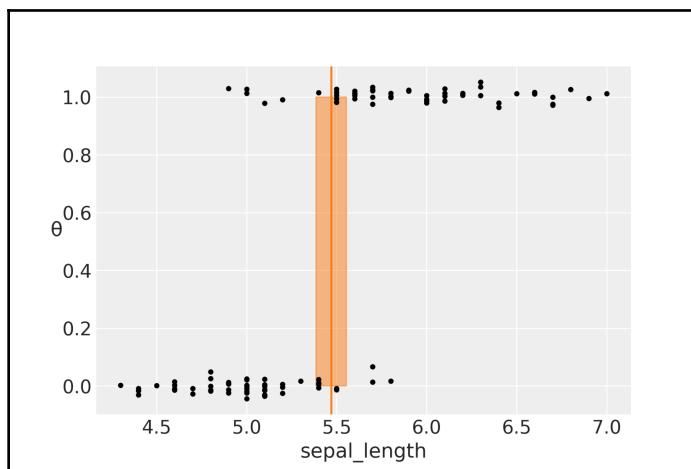


Figure 4.9

Compare *Figure 4.9* with *Figure 4.4*; they're pretty similar, right? Also, check the values of the boundary decision in the following summary:

```
az.summary(trace_lda)
```

	mean	sd	mc error	hpd 3%	hpd 97%	eff_n	r_hat
$\mu[0]$	5.01	0.06	0.0	4.89	5.13	2664.0	1.0
$\mu[1]$	5.94	0.06	0.0	5.82	6.06	2851.0	1.0
σ	0.45	0.03	0.0	0.39	0.51	2702.0	1.0
bd	5.47	0.05	0.0	5.39	5.55	2677.0	1.0

Both the LDA model and the logistic regression provide similar results. The linear-discriminant model can be extended to more than one feature by modeling the classes as multivariate Gaussians. Also, it is possible to relax the assumption of the classes sharing a common variance (or covariance). This leads to a model known as **quadratic linear discriminant (QDA)**, since now the decision boundary is not linear but quadratic.

In general, LDA or QDA models will work better than logistic regression when the features we are using are more or less Gaussian-distributed, and logistic regression will perform better in the opposite case. One advantage of the generative model for classification is that it may be easier or more natural to incorporate prior information; for example, we may have information about the mean and variance of the data to incorporate into the model.

It is important to note that the boundary decisions of LDA and QDA are known in closed form and hence they are usually computed in such a way. To use an LDA for two classes and one feature, we just need to compute the mean of each distribution and average those two values, and we get the boundary decision. In the preceding model, we just did that but with a Bayesian twist. We estimated the parameters of the two Gaussians and then we plugged those estimates into a predefined formula. Where do such formulas come from? Well, without getting into too much detail, to obtain that formula, we must assume that the data is Gaussian-distributed, and hence such a formula will only work if the data does not deviate drastically from normality. Of course, we may hit a problem if we want to relax the normality assumption, such as by using a Student's t-distribution (or a multivariate Student's t-distribution, or whatever). In such a case, we can no longer use the closed form for the LDA (or QDA); nevertheless, we can still compute a decision boundary numerically using PyMC3.

Poisson regression

Another very popular generalized linear model is the Poisson regression. This model assumes data is distributed according to the, wait for it... Poisson distribution.

One scenario where Poisson distribution is useful is when counting things, such as the decay of a radioactive nucleus, the number of children per couple, or the number of Twitter followers. What all these examples have in common is that we usually model them using discrete non-negative numbers: {0, 1, 2, 3,}. This type of variable receives the name of **count data**.

Poisson distribution

Imagine we are counting the number of red cars passing through an avenue per hour. We could use Poisson distribution to describe this data. Poisson distribution is generally used to describe the probability of a given number of events occurring on a fixed time/space interval. Thus, the Poisson distribution assumes that the events occur independently of each other and at a fixed interval of time and/or space. This discrete distribution is parametrized using only one value, μ (the rate, also commonly represented with the Greek letter λ). μ corresponds to the mean and also the variance of the distribution. The probability mass function of Poisson distribution is as follows:

$$f(x | \mu) = \frac{e^{-\mu} \mu^x}{x!} \quad (4.17)$$

The equation is described here:

- μ is the average number of events per unit of time/space
- x is a positive integer value {0, 1, 2, ...}
- $x!$ is the factorial of x , that is, $x! = x \times (x - 1) \times (x - 2) \times \dots \times 2 \times 1$.

In *Figure 4.10*, we can see some examples of the Poisson distribution family for different values of μ :

```
mu_params = [0.5, 1.5, 3, 8]
x = np.arange(0, max(mu_params) * 3)
for mu in mu_params:
    y = stats.poisson(mu).pmf(x)
    plt.plot(x, y, 'o-', label=f'\u03bc = {mu:.1f}')
plt.legend()
plt.xlabel('x')
plt.ylabel('f(x)')
```

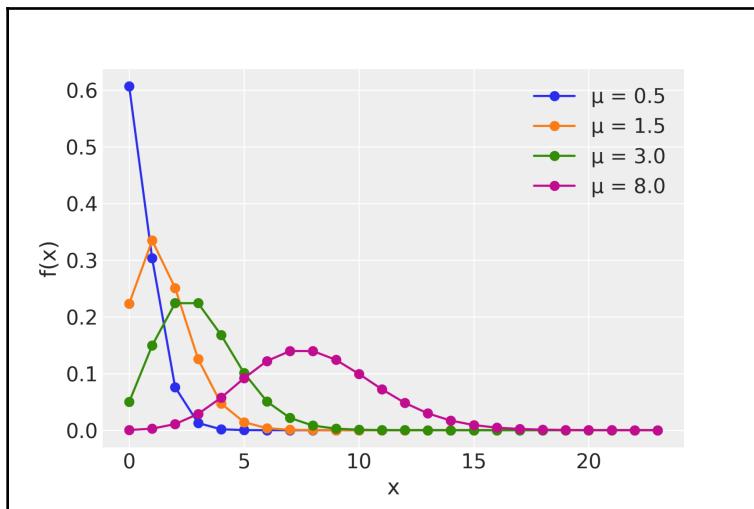


Figure 4.10

Note that μ can be a float, but the output of the distribution is always an integer. In *Figure 4.10*, the dots represent the values of the distribution, while the continuous lines are a visual aid to help us easily grasp the *shape* of the distribution. Remember, the Poisson distribution is a discrete distribution.

The Poisson distribution can be seen as a special case of the binomial distribution when the number of trials n is very large but the probability of success p is very low. Without going into too much mathematical detail, let's try to clarify this statement. Following the car example, we can affirm that we either see the red car or we do not, thus we can use a binomial distribution. In that case we have:

$$x \sim Bin(n, p) \quad (4.18)$$

Then, the mean of the binomial distribution is:

$$\mathbf{E}[x] = np \quad (4.19)$$

And the variance is given by:

$$\mathbf{V}[x] = np(1 - p) \quad (4.20)$$

But note that even if you are in a very busy avenue, the chance of seeing a red car compared to the total number of cars in a city is very small, and therefore we have:

$$n >> p \Rightarrow np \simeq np(1 - p) \quad (4.21)$$

So, we can make the following approximation:

$$\mathbf{V}[x] = np \quad (4.22)$$

Now, the mean and the variance are represented by the same number, and we can confidently state that our variable is distributed as a Poisson distribution:

$$x \sim \text{Poisson}(\mu = np) \quad (4.23)$$

The zero-inflated Poisson model

When counting things, one option is to not count a thing, that is, to get a zero. The number zero can generally occur for many reasons; we get a zero because we were counting red cars and a red car did not pass through the avenue or because we missed it (maybe we did not see that tiny red car behind that large truck). So if we use a Poisson distribution, we will notice, for example, when performing a posterior predictive check, that the model generates fewer zeros compared to the data. How do we fix that? We may try to address the exact cause of our model predicting fewer zeros than observed and include that factor in the model. But, as is often the case, it is enough and easier for our purposes just to assume that we have a mixture of two processes:

- One modeled by a Poisson distribution with probability ψ
- One giving *extra* zeros with probability $1 - \psi$

This is known as the **zero-inflated Poisson (ZIP)** model. In some texts, you will find that ψ represents the extra zeros and $1 - \psi$ the probability of the Poisson. This is not a big deal; just pay attention to which is which for a concrete example.

Basically, a ZIP distribution is:

$$p(y_j = 0) = 1 - \psi + (\psi)e^{-\mu} \quad (4.24)$$

$$p(y_j = k_i) = \psi \frac{\mu^{x_i} e^{-\mu}}{x_i!} \quad (4.25)$$

Where $1 - \psi$ is the probability of extra zeros.

To exemplify the use of the ZIP distribution, let's create a few synthetic data points:

```
n = 100
theta_real = 2.5
psi = 0.1

# Simulate some data
counts = np.array([(np.random.random() > (1-psi)) *
                   np.random.poisson(theta_real) for i in range(n)])
```

We could easily implement equations 4.24 and 4.25 into a PyMC3 model. However, we can do something even easier: we can use the built-in ZIP distribution from PyMC3:

```
with pm.Model() as ZIP:
    psi = pm.Beta('psi', 1, 1)
    theta = pm.Gamma('theta', 2, 0.1)
    y = pm.ZeroInflatedPoisson('y', psi, theta,
                                observed=counts)
    trace = pm.sample(1000)
```

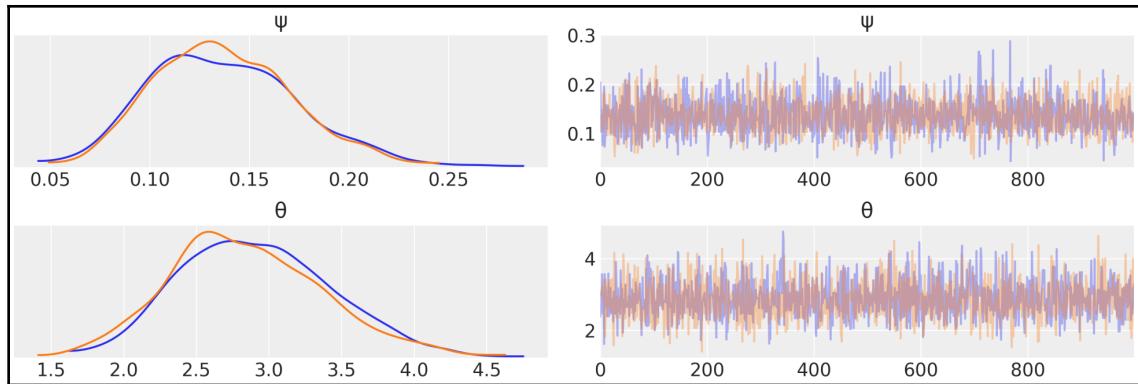


Figure 4.11

Poisson regression and ZIP regression

The ZIP model may look a little dull, but sometimes we need to estimate simple distributions, such as this one, or others such as Poisson or Gaussian distributions.

Anyway, we can also use Poisson or ZIP distributions as part of a linear model. As we saw with the logistic and softmax regressions, we can use an inverse link function to transform the result of a linear model into a variable in a range suitable to be used with other distributions than the normal. Following the same idea, we can now perform a regression analysis where the output variable is a count variable using a Poisson or a ZIP distribution. This time, we can use the exponential function, e^{\cdot} , as the inverse link function. This choice guarantees the values returned by the linear model are always positive:

$$\theta = e^{(\alpha + X\beta)} \quad (4.26)$$

To exemplify a ZIP-regression model implementation, we are going to work with a dataset taken from the *Institute for Digital Research and Education* (<http://www.ats.ucla.edu/stat/data>). We have 250 groups of visitors to a park. Here are some parts of the data per group:

- The number of fish they caught (count)
- How many children were in the group (child)
- Whether they brought a camper to the park (camper)

Using this data, we are going to build a model that predicts the number of caught fish as a function of the child and camper variables. We can use pandas to load the data:

```
fish_data = pd.read_csv('../data/fish.csv')
```

I leave it as an exercise for you to explore the dataset using plots and/or a Pandas function, such as `describe`. For now, we are going to continue by implementing the ZIP_reg PyMC3 model:

```
with pm.Model() as ZIP_reg:
    ψ = pm.Beta('ψ', 1, 1)
    α = pm.Normal('α', 0, 10)
    β = pm.Normal('β', 0, 10, shape=2)
    θ = pm.math.exp(α + β[0] * fish_data['child'] + β[1] *
    fish_data['camper'])
    yl = pm.ZeroInflatedPoisson('yl', ψ, θ, observed=fish_data['count'])
    trace_ZIP_reg = pm.sample(1000)
```

Camper is a binary variable with a value of 0 for not-camper and 1 for camper. A variable indicating the absence/presence of an attribute is usually denoted as a *dummy variable* or *indicator variable*. Note that when camper takes the value of 0, the term involving β_1 will also be 0 and the model reduces to a regression with a single independent variable.

To better understand the results of our inference, let's do a plot:

```
children = [0, 1, 2, 3, 4]
fish_count_pred_0 = []
fish_count_pred_1 = []
for n in children:
    without_camper = trace_ZIP_reg['α'] + trace_ZIP_reg['β'][:,0] * n
    with_camper = without_camper + trace_ZIP_reg['β'][:,1]
    fish_count_pred_0.append(np.exp(without_camper))
    fish_count_pred_1.append(np.exp(with_camper))
plt.plot(children, fish_count_pred_0, 'C0o', alpha=0.01)
plt.plot(children, fish_count_pred_1, 'C1o', alpha=0.01)

plt.xticks(children);
plt.xlabel('Number of children')
plt.ylabel('Fish caught')
plt.plot([], 'C0o', label='without camper')
plt.plot([], 'C1o', label='with camper')
plt.legend()
```

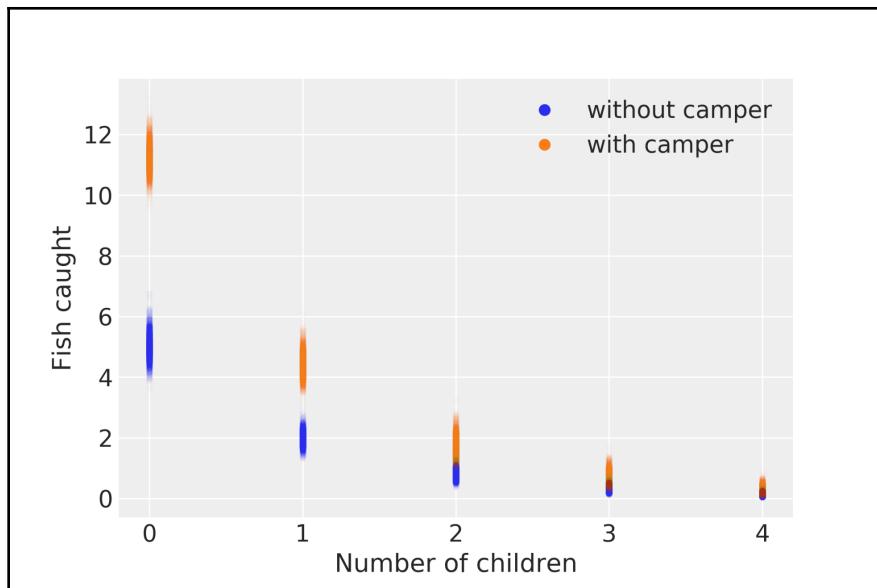


Figure 4.12

From *Figure 4.12*, we can see that the higher the number of children, the lower the number of fish caught. Also people that travel with a camper generally catch more fish. If you check the β coefficients for child and camper, you will see that we can say the following:

- For each additional child, the expected count of the fish caught decreases by ≈ 0.4
- Being in a camper increases the expected count of the fish caught by ≈ 2

We arrive at these values by taking the exponential of the β_1 and β_2 coefficients, respectively.

Robust logistic regression

We just saw how to fix an excess of zeros without directly modeling the factor that generates them. A similar approach, suggested by Kruschke, can be used to perform a more robust version of logistic regression. Remember that in logistic regression, we model the data as binomial, that is, zeros and ones. So it may happen that we find a dataset with unusual zeros and/or ones. Take, as an example, the iris dataset that we already saw, but with some added *intruders*:

```
iris = sns.load_dataset("iris")
df = iris.query("species == ('setosa', 'versicolor')")
y_0 = pd.Categorical(df['species']).codes
x_n = 'sepal_length'
x_0 = df[x_n].values
y_0 = np.concatenate((y_0, np.ones(6, dtype=int)))
x_0 = np.concatenate((x_0, [4.2, 4.5, 4.0, 4.3, 4.2, 4.4]))
x_c = x_0 - x_0.mean()
plt.plot(x_c, y_0, 'o', color='k');
```

Here, we have some versicolors (1s) with an unusually short sepal length. We can fix this with a mixture model. We are going to say that the output variable comes with the π probability by random guessing, or with the $1 - \pi$ probability from a logistic regression model. Mathematically we have:

$$p = \pi 0.5 + (1 - \pi) \text{logistic}(\alpha + X\beta) \quad (4.26)$$

Notice that when $\pi = 1$, we get $p = 0.5$, and for $\pi = 0$, we recover the expression for logistic regression. Implementing this model is a straightforward modification of the first model from this chapter:

```
with pm.Model() as model_rlg:
    α = pm.Normal('α', mu=0, sd=10)
    β = pm.Normal('β', mu=0, sd=10)
    μ = α + x_c * β
    θ = pm.Deterministic('θ', pm.math.sigmoid(μ))
    bd = pm.Deterministic('bd', -α/β)
    π = pm.Beta('π', 1., 1.)
    p = π * 0.5 + (1 - π) * θ
    y1 = pm.Bernoulli('y1', p=p, observed=y_0)

    trace_rlg = pm.sample(1000)
```

If we compare these results with those from `model_0` (the first model in this chapter), we will see that we get more or less the same boundary. As we can see by comparing *Figure 4.13* with *Figure 4.4*:

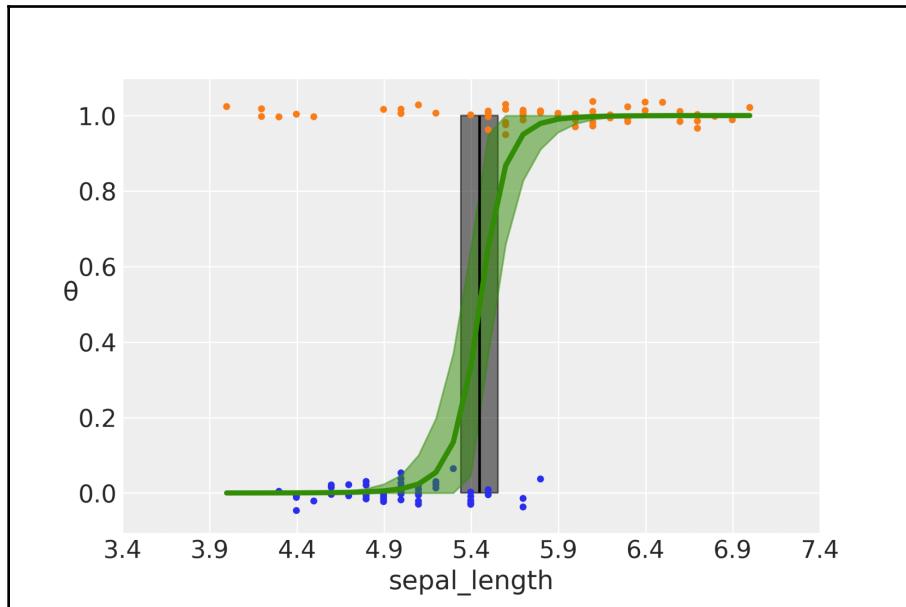


Figure 4.13

You may also want to compute the summary for `model_0` and `model_rlg` to compare the values of the boundary according to each model.

The GLM module

As we discussed at the beginning of this chapter, linear models are very useful statistical tools. Extensions such as the ones we saw in this chapter make them even more general tools. For that reason, PyMC3 includes a module to simplify the creation of linear models: the **Generalized Liner Model (GLM)** module. For example, a simple linear regression will be as follows:

```
with pm.Model() as model:  
    glm.glm('y ~ x', data)  
    trace = sample(2000)
```

The second line of the preceding code takes care of adding priors for the intercept and for the slope. By default, the intercept is assigned a flat prior, and the slopes $\mathcal{N}(0, 1 \times 10^6)$ an prior. Note that the **maximum a posteriori (MAP)** of the default model will be essentially equivalent to the one obtained using the ordinary least squared method. These is totally fine as a default linear regression; you can change it using the `priors` argument. The GLM module also adds a Gaussian likelihood by default. You can change it using the `family` argument; you can choose from the following options: `Normal` (default), `StudentT`, `Binomial`, `Poisson`, or `NegativeBinomial`.

To describe a statistical model, the GLM module uses Patsy (<https://patsy.readthedocs.io/en/latest/index.html>), which is a Python library that provides a *formula mini-language* syntax inspired by the one used in R and S. In the previous code block, `y ~ x` specifies we have an output variable, `y`, that we want to estimate as a linear function of `x`.

Summary

The main idea discussed in this chapter is a rather simple one: in order to predict the mean of an output variable, we can apply an arbitrary function to a linear combination of input variables. I know I already said this at the beginning of the chapter, but repetition is important. We call that arbitrary function the inverse link function. The only restriction we have in choosing such a function is that the output has to be adequate to be used as a parameter of the sampling distribution (generally the mean). One situation in which we would like to use an inverse link function is when working with categorical variables, another is when the data can only take positive values, and yet another is when we need a variable in the $[0, 1]$ interval. All these different variations become different models; many of those models are routinely used as statistical tools, and their application and statistical properties have been studied.

The logistic regression model is a generalization of the linear regression model from the previous chapter, which can be used for classification or in general for predicting binomial data. The distinct feature of logistic regression is the use of the logistic as the inverse link function and the use of the Bernoulli distribution as the likelihood. The use of an inverse link function introduces a non-linearity that we should take into account when interpreting the coefficients of logistic regression. The coefficient encodes the increase in log-odds units by unit increase of the x variable.

One way to generalize logistic regression to more than two classes is with softmax regression. Now the inverse link function is the softmax function, which is a generalization of the logistic function for more than two values, and the use of the categorical distribution as the likelihood.

Both logistic and softmax are examples of discriminative models; we try to model $p(y | x)$ without explicitly modeling $p(x)$. A generative model for classification will first model $p(x | y)$, that is, the distribution of x for each class, y , and then assign the classes. This kind of model is called a **generative classifier** because we are creating a model from which we can generate samples from each class. We saw one example of a generative classifier using Gaussian distributions.

We used the iris dataset to demonstrate all these models and to briefly discuss correlated variables, perfectly-separable classes, and unbalanced classes.

Another the popular generalized linear models is the Poisson regression. This model assumes data is distributed according to the Poisson distribution and the inverse link function is the exponential function. The Poisson distribution and regression are useful to model count data, that is, data taking only non-negative integers, arising from counting rather than ranking. Most distributions are connected among them; one example is the Gaussian and the Student's t-distributions. Another example is the Poisson distribution, which can be seen as a special case of the binomial distribution when the number of trials is very large but the probability of success is very low.

One useful way of extending the Poisson distribution is the ZIP distribution; we can think of the latter as a mixture of two other distributions: a Poisson distribution and a binary distribution that generates *extra zeros*. Another useful extension that can be interpreted as a mixture is the negative-binomial distribution—in this case as a mixture of Poisson distributions where the rate (μ) is a random variable distributed as a gamma distribution. The negative binomial is a useful alternative to the Poisson distribution when the data is over-dispersed, that is, it is data that has a variance greater than its mean.

Exercises

1. Rerun the first model using the petal length and then petal width variables. What are the main differences in the results? How wide or narrow is the 95% HPD interval in each case?
2. Repeat exercise 1, this time using a Student's t-distribution as a weakly informative prior. Try different values of ν .
3. Go back to the first example, the logistic regression for classifying setosa or versicolor given sepal length. Try to solve the same problem using a simple linear regression model, as we saw in Chapter 3, *Modeling with Linear Regression*. How useful is linear regression compared to logistic regression? Can the result be interpreted as a probability? Tip, check whether the values of y are restricted to the [0, 1] interval.

4. In the example from the *Interpreting the coefficients of a logistic regression* section, we changed `sepal_length` by 1 unit. Using *Figure 4.6*, corroborate that the value of `log_odds_versicolor_i` corresponds to the value of `probability_versicolor_i`. Do the same for `log_odds_versicolor_f` and `probability_versicolor_f`. Just by noting `log_odds_versicolor_i` is negative, what can you say about the probability? Use *Figure 4.6* to help you. Is this result clear to you from the definition of log-odds?
5. Use the same example from the previous exercise. For `model_1`, check how much the log-odds change when increasing `sepal_length` from 5.5 to 6.5 (spoiler: it should be 4.66). How much does the probability change? How does this increase compare to when we increase from 4.5 to 5.5?
6. In the example for dealing with unbalanced data, change `df = df[45:]` to `df = df[22:78]`. This will keep roughly the same number of data points, but now the classes will be balanced. Compare the new result with the previous ones. Which one is more similar to the example using the complete dataset?
7. Suppose instead of a softmax regression, we use a simple linear model by coding `setosa =0`, `versicolor =1`, and `virginica = 2`. Under the simple linear regression model, what will happen if we switch the coding? Will we get the same or different results?
8. Compare the likelihood of the logistic model versus the likelihood of the LDA model. Use the `sample_posterior_predictive` function to generate predicted data, and compare the types of data you get for both cases. Be sure you understand the difference between the types of data the model predicts.
9. Using the `fish` data, extend the `ZIP_reg` model to include the `persons` variable as part of a linear model. Include this variable to model the number of extra zeros. You should get a model that includes two linear models: one connecting the number of children and the presence/absence of a camper to the Poisson rate (as in the example we saw), and another connecting the number of persons to the ψ variable. For the second case, you will need a logistic inverse link!
10. Use the data for the robust logistic example to feed a non-robust logistic regression model and to check that the outliers actually affected the results. You may want to add or remove outliers to better understand the effect of the estimation on a logistic regression and the robustness on the model introduced in this chapter.
11. Read and run the following notebooks from PyMC3's documentation:
 - <https://pymc-devs.github.io/pymc3/notebooks/GLM-linear.html>
 - <https://pymc-devs.github.io/pymc3/notebooks/GLM-robust.html>
 - <https://pymc-devs.github.io/pymc3/notebooks/GLM-hierarchical.html>

5

Model Comparison

"A map is not the territory it represents, but, if correct, it has a similar structure to the territory."

-Alfred
Korzybski

Models should be designed as approximations to help us understand a particular problem, or a class of related problems. Models are not designed to be verbatim copies of the *real world*. Thus, all models are *wrong* in the same sense that maps are not the territory. Even when *a priori*, we consider every model to be *wrong*, not every model is equally wrong; some models will be better than others at describing a given problem. In the foregoing chapters, we focused our attention on the inference problem, that is, how to learn values of parameters from the data. In this chapter, we are going to focus on a complementary problem: how to compare two or more models that are used to explain the same data. As we will learn, this is not a simple problem to solve and at the same time is a central problem in data analysis.

In this chapter, we will explore the following topics:

- Posterior predictive checks
- Occam's razor—simplicity and accuracy
- Overfitting and underfitting
- Information criteria
- Bayes factors
- Regularizing priors

Posterior predictive checks

In Chapter 1, *Thinking Probabilistically*, we introduced the concept of posterior predictive checks, and, in subsequent chapters, we have used it as a way to evaluate how well models explain the same data that's used to fit the model. The purpose of posterior predictive checks is not to dictate that a model is wrong; we already know that! By performing posterior predictive checks, we hope to get a better grasp of the limitations of a model, either to properly acknowledge them, or to attempt to improve the model. Implicit in the previous statement is the fact that models will not generally reproduce all aspects of a problem equally well. This is not generally a problem given that models are built with a purpose in mind. A posterior predictive check is a way to evaluate a model in the context of that purpose; thus, if we have more than one model, we can use posterior predictive checks to compare them.

Let's upload and plot a very simple dataset:

```
dummy_data = np.loadtxt('..../data/dummy.csv')
x_1 = dummy_data[:, 0]
y_1 = dummy_data[:, 1]

order = 2
x_1p = np.vstack([x_1**i for i in range(1, order+1)])
x_1s = (x_1p - x_1p.mean(axis=1, keepdims=True)) / \
    x_1p.std(axis=1, keepdims=True)
y_1s = (y_1 - y_1.mean()) / y_1.std()
plt.scatter(x_1s[0], y_1s)
plt.xlabel('x')
plt.ylabel('y')
```

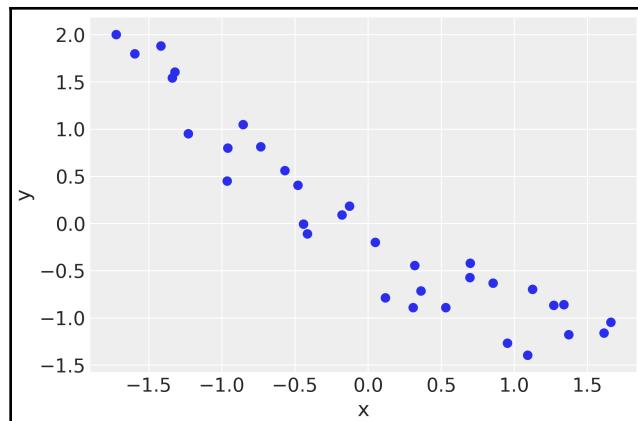


Figure 5.1

Now, we are going to fit this data with two slightly different models, a linear one and a polynomial of order 2, also known as a **parabolic** or **quadratic** model:

```

with pm.Model() as model_l:
    α = pm.Normal('α', mu=0, sd=1)
    β = pm.Normal('β', mu=0, sd=10)
    ε = pm.HalfNormal('ε', 5)

    μ = α + β * x_1s[0]

    y_pred = pm.Normal('y_pred', mu=μ, sd=ε, observed=y_1s)

    trace_l = pm.sample(2000)

with pm.Model() as model_p:
    α = pm.Normal('α', mu=0, sd=1)
    β = pm.Normal('β', mu=0, sd=10, shape=order)
    ε = pm.HalfNormal('ε', 5)

    μ = α + pm.math.dot(β, x_1s)

    y_pred = pm.Normal('y_pred', mu=μ, sd=ε, observed=y_1s)

    trace_p = pm.sample(2000)

```

Now, we will plot the mean fit for both models:

```

x_new = np.linspace(x_1s[0].min(), x_1s[0].max(), 100)

α_l_post = trace_l['α'].mean()

```

```

β_l_post = trace_l['β'].mean(axis=0)
y_l_post = α_l_post + β_l_post * x_new

plt.plot(x_new, y_l_post, 'C1', label='linear model')

α_p_post = trace_p['α'].mean()
β_p_post = trace_p['β'].mean(axis=0)
idx = np.argsort(x_1s[0])
y_p_post = α_p_post + np.dot(β_p_post, x_1s)

plt.plot(x_1s[0][idx], y_p_post[idx], 'C2', label=f'model order {order}')

α_p_post = trace_p['α'].mean()
β_p_post = trace_p['β'].mean(axis=0)
x_new_p = np.vstack([x_new**i for i in range(1, order+1)])
y_p_post = α_p_post + np.dot(β_p_post, x_new_p)

plt.scatter(x_1s[0], y_1s, c='C0', marker='.')
plt.legend()

```

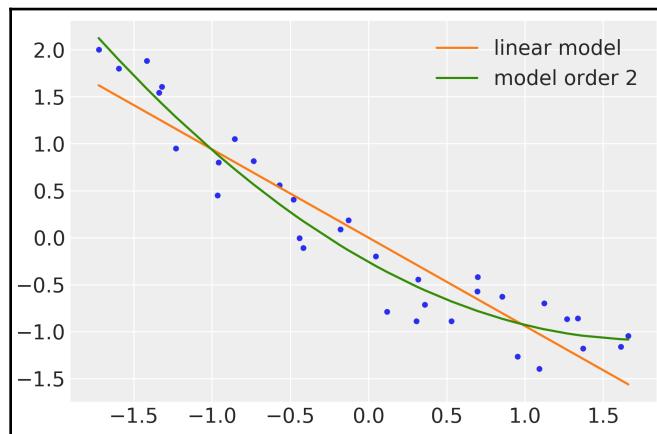


Figure 5.2

The order 2 model seems to be doing a better job, but the linear model is not that bad. Let's use PyMC3 to obtain posterior predictive samples for both models:

```

y_l = pm.sample_posterior_predictive(trace_l, 2000,
                                      model=model_l)['y_pred']

y_p = pm.sample_posterior_predictive(trace_p, 2000,
                                      model=model_p)['y_pred']

```

As we already saw, posterior predictive checks are often performed using visualizations, as in the following example:

```
plt.figure(figsize=(8, 3))
data = [y_1s, y_l, y_p]
labels = ['data', 'linear model', 'order 2']
for i, d in enumerate(data):
    mean = d.mean()
    err = np.percentile(d, [25, 75])
    plt.errorbar(mean, -i, xerr=[[-err[0]], [err[1]]], fmt='o')
    plt.text(mean, -i+0.2, labels[i], ha='center', fontsize=14)
plt.ylim([-i-0.5, 0.5])
plt.yticks([])
```

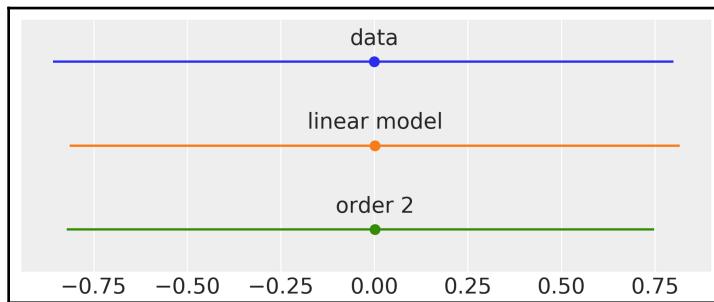


Figure 5.3

The preceding diagram shows the mean and the **interquartile range (IQR)** for the data and for the linear and quadratic models. In this diagram, we are averaging over the posterior predictive samples for each model. We can see that the mean is (on average) well reproduced for both models, and that the interquartile range is not very off, but there are some small differences that, in a real problem, could be worthy of some attention. We could do many different plots to explore a posterior predictive distribution. For example, we can plot the dispersion of both the mean and the interquartile range, as opposed to their mean values. The following diagram is an example of such a plot:

```
fig, ax = plt.subplots(1, 2, figsize=(10, 3), constrained_layout=True)

def iqr(x, a=0):
    return np.subtract(*np.percentile(x, [75, 25], axis=a))

for idx, func in enumerate([np.mean, iqr]):
    T_obs = func(y_1s)
    ax[idx].axvline(T_obs, 0, 1, color='k', ls='--')
```

```

for d_sim, c in zip([y_l, y_p], ['C1', 'C2']):
    T_sim = func(d_sim, 1)
    p_value = np.mean(T_sim >= T_obs)
    az.plot_kde(T_sim, plot_kwarg={'color': c},
                 label=f'p-value {p_value:.2f}', ax=ax[idx])
    ax[idx].set_title(func.__name__)
    ax[idx].set_yticks([])
    ax[idx].legend()

```

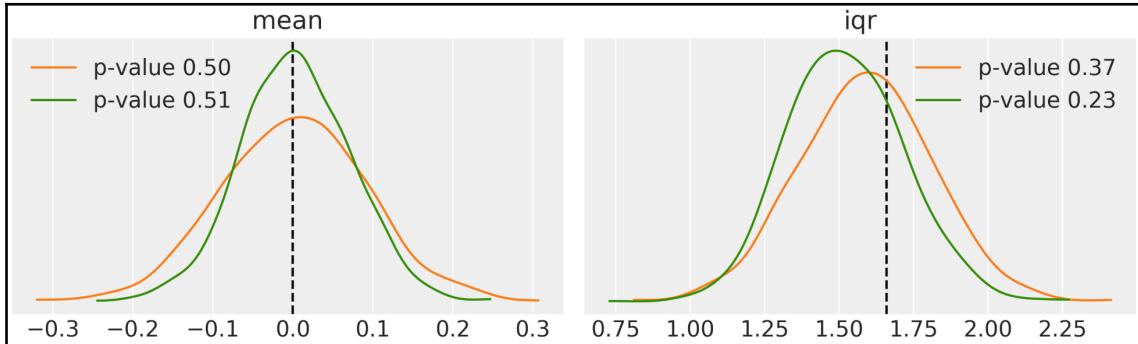


Figure 5.4

In *Figure 5.4*, the black dashed line represents the statistic computed from the data (either the mean or the IQR). Because we have a single dataset we have a single value for the statistic (and not a distribution). The curves (using the same color code from *Figure 5.3*) represent the distribution of the mean (left panel) or interquartile range (right panel) that was computed from the posterior predictive samples. You may have also noted that *Figure 5.4* also includes values labeled as *p*-values. We compute such *p*-values by comparing the simulated data to the actual data. For both sets, we compute a summary statistic (the mean or IQR, in this example), and then we count the proportion of times the summary statistics from the simulation is equal or greater than the one computed from the data. If the data and simulation agrees, we should expect a *p*-value around 0.5, otherwise we are in the presence of a biased posterior predictive distribution.



Bayesian's *p*-values are just a way to get a number measuring the fit of a posterior predictive check.

If you are familiar with *frequentists methods* and you are reading this book because you were told that the cool guys were not doing p -values anymore, then you may feel shocked or even disappointed by your formerly beloved author. But keep calm and keep reading. These Bayesian p -values are indeed p -values because they are basically defined in the same way as their frequentist cousins:

$$\text{Bayesian } p\text{-value} \triangleq p(T_{sim} \geq T_{obs}) \mid y \quad (5.1)$$

That is, we are getting the probability of getting a simulated statistic T_{sim} equal or more extreme than a statistic from the data T_{obs} . Here, T can be almost anything that provides a summary of the data. In *Figure 5.4*, T is the mean for the left panel and the standard deviation for the right panel. T should be chosen while taking into account the question that motivated the inference in the first place.

These p -values are **Bayesian** because, for the *sampling distribution*, we are using a posterior predictive distribution. Also notice that we are not conditioning on any null hypothesis; in fact, we have the entire posterior distribution of θ and we are conditioning on the observed data. Yet another difference is the fact that we are not using any predefined threshold to declare statistical significance, nor are we performing hypothesis testing—we are just trying to compute a number to assess the fit of the posterior predictive distribution to a dataset.

Posterior predictive checks, either using plots or numerical summaries such as Bayesian p -values, or even a combination of both, are very flexible ideas. This concept is general enough to let the analyst use their imagination to come up with different ways to explore the posterior predictive distribution and use whatever fits well to allow them to tell a data-driven story, including but not restricted to model comparison. In the following sections, we are going to explore other methods to compare models.

Occam's razor – simplicity and accuracy

When choosing among alternatives, there is a guiding principle known as Occam's razor that loosely states the following:

If we have two or more equivalent explanations for the same phenomenon, we should choose the simpler one.

There are many justifications for this heuristic; one of them is related to the falsifiability criterion introduced by Popper. Another takes a pragmatic perspective and states that: *Given simpler models are easier to understand than more complex models, we should keep the simpler one.* Another justification is based on Bayesian statistics, as we will see when we discuss Bayes factors. Without getting into the details of these justifications, we are going to accept this criterion as a useful rule of thumb for the moment, just something that sounds reasonable.

Another factor we generally should take into account when comparing models is their accuracy, that is, how well the model fits the data. We have already seen some measures of accuracy, such as the coefficient of determination R^2 , that we can interpret as the proportion of explained variance in a linear regression, also that the posterior predictive checks are based on the idea of accuracy of the data. If we have two models and one of them explains the data better than the other, we should prefer that model, that is, we want the model with higher accuracy, right?

Intuitively, it seems that when comparing models, we tend to like those that have high accuracy and those that are simple. So far, so good, but what should we do if the simpler model has the worst accuracy? And, more generally, how can we balance both contributions?

During the rest of this chapter, we are going to discuss this idea of balancing between these two contributions. This chapter is more theoretical than the previous chapters (even when we are just scratching the surface of this topic). However, we are going to use code, figures, and examples that will help us move from this (correct) intuition of balancing accuracy versus complexity to a more theoretical (or at least empirical) grounded justification.

We are going to begin by fitting increasingly complex polynomials to a very simple dataset. Instead of using the Bayesian machinery, we are going to use the least square approximation for fitting linear models. Remember that the latter can be interpreted from a Bayesian perspective as a model with flat priors. So, in a sense, we are still being Bayesian here, only we are taking a bit of a shortcut:

```
x = np.array([4., 5., 6., 9., 12, 14.])
y = np.array([4.2, 6., 6., 9., 10, 10.])

plt.figure(figsize=(10, 5))
order = [0, 1, 2, 5]
plt.plot(x, y, 'o')
for i in order:
    x_n = np.linspace(x.min(), x.max(), 100)
    coeffs = np.polyfit(x, y, deg=i)
    ffit = np.polyval(coeffs, x_n)
```

```
p = np.poly1d(coeffs)
yhat = p(x)
ybar = np.mean(y)
ssreg = np.sum((yhat-ybar)**2)
sstot = np.sum((y - ybar)**2)
r2 = ssreg / sstot

plt.plot(x_n, ffit, label=f'order {i}, $R^2={r2:.2f}$')

plt.legend(loc=2)
plt.xlabel('x')
plt.ylabel('y', rotation=0)
```

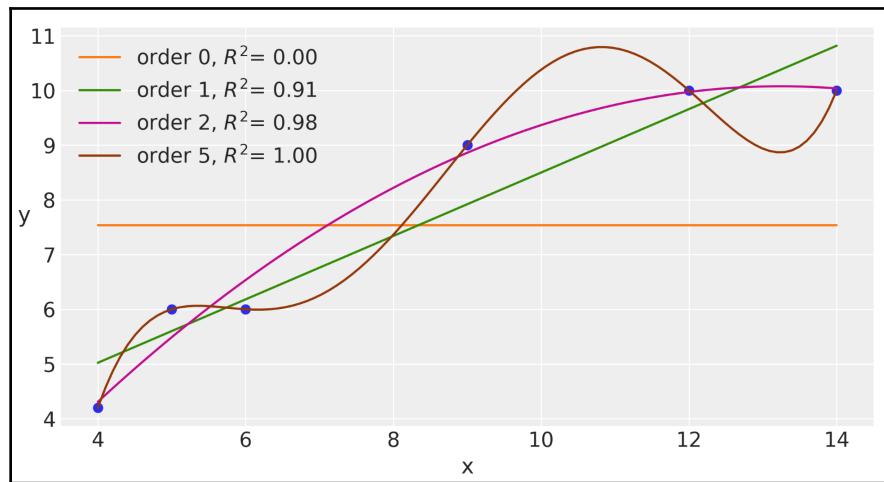


Figure 5.5

Too many parameters leads to overfitting

From *Figure 5.5*, we can see that increasing the complexity of the model is accompanied by an increasing accuracy that's reflected in the coefficient of determination R^2 ; in fact, we can see that the polynomial of order 5 fits the data perfectly! You may remember that we briefly discussed this behavior of polynomials in *Chapter 3, Modeling with Linear Regression*, and we also discussed that, in general, it is not a very good idea to use polynomials for real problems.

Why is the polynomial of order 5 able to capture the data without missing a single data point? The reason for this is that we have the same number of parameters, 6, as the number of data points, which is also 6, and hence, the model is just encoding the data in a different way. The model is not really learning something from the data, it is just memorizing stuff! From this example, we can see that a model with higher accuracy is not always what we really want.

Imagine that we get more money or time and hence we collect more data points to include in the previous dataset. For example, we collect the points $[(10, 9), (7, 7)]$ (see *Figure 5.6*).

How well does the order 5 model explain these points compared to the order 1 or 2 models? Not very well, right? The order 5 model did not learn any interesting pattern in the data; instead, it just memorized stuff (sorry for persisting with this idea) and hence the order 5 model does a very bad job at generalizing to future, unobserved, but potentially observable, data:

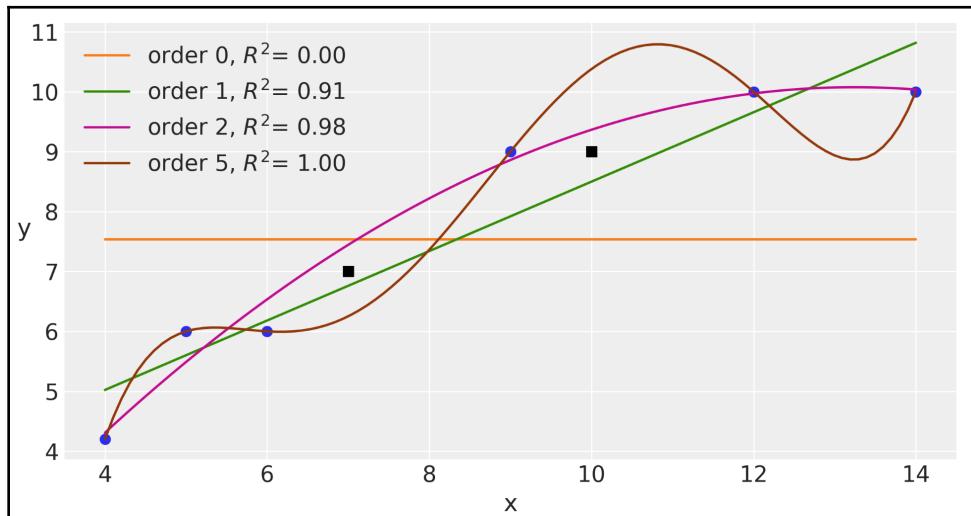


Figure 5.6

When a model fits very well with the dataset that was used to learn its parameters in the first place, but is very bad at fitting other datasets, we say that we have overfitting. This is a very general problem in statistics and machine learning. A very useful way of picturing the problem of overfitting is by thinking that datasets have two components: the signal and the noise. The signal is whatever we want to learn from the data. If we use a dataset, it is because we think there is a signal in there, otherwise it will be an exercise in futility. The noise, on the other hand, is not useful and is the product of measuring errors, limitations on the way data was generated or captured, corrupted data, and so on. A model overfits when it is so flexible that it learns the noise, effectively hiding the signal. This is a practical justification for Occam's razor. At least in principle, we can always create a model so complex that it explains everything in detail, as in the Empire, as described by Borges, where cartographers attained such a level of sophistication that they created a map of the Empire whose size was that of the Empire, and which coincided point for point with it.

Too few parameters leads to underfitting

Continuing with the same example but on the other extreme of complexity, we have the model of order 0. This model is just a Gaussian model of the variable y disguised as a linear model. Notice from *Figure 5.5* that this model is only capturing the average of the dependent variable, and is completely independent of the values of the variable x . We say that this model has underfitted.

The balance between simplicity and accuracy

Things should be as simple as possible, but not simpler is a quote often attributed to Einstein and is like the Occam's razor with a twist. Ideally, we would like to have a model that neither overfits nor underfits the data. So, in general, we will face a trade-off and somehow we have to optimize or tune our models.

This trade-off is usually discussed in terms of variance and bias:

- High bias is the result of a model with low ability to accommodate the data. High bias can cause a model to miss the relevant pattern and thus can lead to underfitting.
- High variance is the result of a model that has high sensitivity to details in the data. High variance can cause a model to capture the noise in the data and thus can lead to overfitting.

In Figure 5.5, the order 0 model is the one with the higher bias (and lower variance), because it is *biased* to return a *flat line* at the value of the mean of the variable y , irrespective of the value of x . The order 5 model is the one with the higher variance (and lower bias). The easier way to see this is to imagine different datasets of six points. You can arrange these six points in very different ways, and this model will adapt to each one of this arrangement. It will fit perfectly with most of them (except some arrangements, like circles). See exercise 6 at the end of this chapter for more details.

A model with high bias is a model with more prejudices (if you will excuse the anthropomorphization) or more inertia (if you will excuse the physicalization), while a model with high variance is a more *open-minded* model. The problem of being too biased is that you are ill-equipped to accommodate new evidences; the problem of being too *open-minded* is that you end up believing nonsensical stuff like terraplaners or anti-Vaxxers. In general, when we increase one of these terms, we decrease the other, leading us to a bias-variance trade-off. Once again, the main idea is that we want a balanced model.

Predictive accuracy measures

In the previous example, it is more or less easy to see that the order 0 model is very simple and that the order 5 model is too complex for the data, but what about the other two? To answer this question, we need a more principled way of taking into account the accuracy on one side and the simplicity on the other. To do so, we will need to introduce several new concepts. The first two are:

- **Within-sample accuracy:** The accuracy measured with the data that's used to fit a model
- **Out-of-sample accuracy:** The accuracy of the model that's measured on data that is not used for fitting the model—this is also known as predictive accuracy

For any combination of data and models, the within-sample accuracy will be, on average, smaller than the out-of-sample accuracy. Thus, using the within-sample accuracy could fool us into thinking that we have a better model than we truly have. For this reason, out-of-sample measures are preferred over within-sample measures. However, generally, there is a problem with this. We need to be able to afford leaving aside a portion of the data—not to fit the model, but to test it. This is often a luxury for most analysts. To circumvent this problem, people have spent a lot of effort devising methods to estimate the out-of-sample predictive accuracy using only within-sample data. Two such methods for this are:

- **Cross-validation:** This is an empirical strategy based on dividing the available data into subsets that are used for fitting and evaluation in an alternated way

- **Information criteria:** This is an umbrella term for several relatively simple expressions that can be considered as ways to approximate the results that we could have obtained by performing cross-validation

Cross-validation

Cross-validation is a simple and, in most cases, effective solution to evaluate a model without leaving aside data. This process is summarized in the following figure. We take our data and we partition it into K portions. We try to keep the portions more or less equal (in size and sometimes also in other features, such as, for example, an equal number of classes). Then, we use $K - 1$ on them to train the model and the remaining to test it. Then, we repeat this procedure systematically by leaving a different portion out of the training set and using that portion as the test set until we have done K rounds. The results, A_k , are then averaged along the K runs. This is known as **K-fold cross-validation**. When K is equal to the number of data points, we get what is known as **leave-one-out cross-validation (LOOCV)**. Sometimes, when doing LOOCV, the number of rounds can be less than the total number of data points if we have a prohibitive number of them:

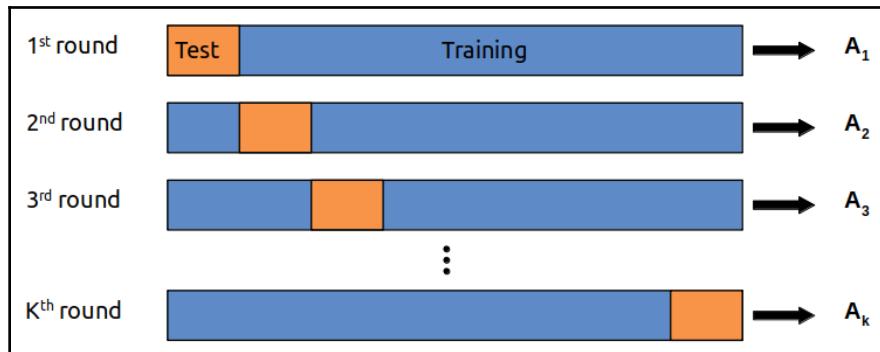


Figure 5.7

CV is the bread and butter of machine learner practitioners. There are a few details we are obviating here, but this is OK for the current discussion. For more details, you could read the book *Python Machine Learning*, by Sebastian Raschka, or *Python Data Science Handbook* by Jake Vanderplas.

Cross-validation is a very simple and useful idea, but for some models or for large amounts of data, the computational cost of cross-validation may be beyond our possibilities. Many people have tried to come up with simpler-to-compute quantities that approximate the results obtained with cross-validation and that work in scenarios where cross-validation is not that straightforward to perform. This is the subject of the next section.

Information criteria

Information criteria is a collection of different and somehow related tools that are used to compare models in terms of how well they fit the data while taking into account their complexity through a penalization term. In other words, information criteria formalizes the intuition we developed at the beginning of this chapter. We need a proper way to balance how well a model explains the data on the one hand, and how complex the model is on the other hand.

The exact way these quantities are derived has to do with a field known as **information theory**, something that is beyond the scope of this book, so we are going to limit ourselves to understand them from a practical point of view.

Log-likelihood and deviance

An intuitive way of measuring how well a model fits the data is to compute the quadratic mean error between the data and the predictions made by the model:

$$\frac{1}{n} \sum_{i=1}^n (y_i - E(y_i | \theta))^2 \quad (5.2)$$

$E(y_i | \theta)$ is just the predicted value, given the estimated parameters.

Note that this is essentially the average of the difference between the observed and predicted data. Squaring the errors ensures that the differences do not cancel out and emphasizes large errors relative to other ways of computing similar quantities, such as using the absolute value. A more general measure is to compute the log-likelihood:

$$\sum_{i=1}^n \log p(y_i | \theta) \quad (5.3)$$

When the likelihood is normal, this turns out to be proportional to the quadratic mean error.

In practice, and for historical reasons, people usually do not use log-likelihood directly; instead, they use a quantity known as deviance:

$$-2 \sum_{i=1}^n \log p(y_i | \theta) \quad (5.4)$$

The deviance is used for Bayesians and non-Bayesians alike; the difference is that under a Bayesian framework, θ is estimated from the posterior and, like any quantity derived from a posterior, it has a distribution. On the contrary, in non-Bayesian settings, θ is a point estimate. To learn how to use this deviance, we should note two key aspects of this quantity:

- The lower the deviance, the higher the log-likelihood and the higher the agreement of the model predictions and the data. Therefore, we want low deviance values.
- The deviance is measuring the within-sample accuracy of the model and hence complex models will generally have a lower deviance than simpler ones. Thus, we need to somehow include a penalization term for complex models.

In the following sections, we will learn about different information criteria. They share the fact that they use deviance and a penalization term. What makes them different is how the deviance and the penalization term are computed.

Akaike information criterion

This is a very well-known and widely used information criterion, especially for non-Bayesians, and is defined as follows:

$$AIC = -2 \sum_{i=1}^n \log p(y_i | \hat{\theta}_{mle}) + 2pAIC \quad (5.5)$$

Here, $pAIC$ is just the number of parameters and $\hat{\theta}_{mle}$ is the maximum likelihood estimation of θ . Maximum likelihood estimation is a common practice for non-Bayesians and, in general, is equivalent to the Bayesian **maximum a posteriori (MAP)** estimation when using flat priors. Notice that $\hat{\theta}_{mle}$ is a point estimation and not a distribution.

Once again, the -2 is there for historical reasons. The important observation, from a practical point of view, is that the first term takes into account how well the model fits the data and the second term penalizes complex models. Hence, if two models explain the data equally well, but one has more parameters than the other, *AIC* tells us that we should choose the one with the fewer parameters.

AIC works well for non-Bayesian approaches, but is problematic otherwise. One reason is that it does not use the posterior, and hence it is discarding information about the uncertainty in the estimation; it is also assuming flat priors and hence this measure is incompatible with informative and weakly informative priors, like those used in this book.

Widely applicable Information Criterion

This is the fully Bayesian version of *AIC*. Like with *AIC*, the **widely applicable information criterion (WAIC)** has two terms: one that measures how well the data fits the model and one penalizing complex models:

$$WAIC = -2l_{ppd} + 2p_{WAIC} \quad (5.6)$$

If you want to better understand what these two terms are please read the *WAIC in depth* section. From a practical point of view, you just need to know that we prefer lower values of *WAIC*.

Pareto smoothed importance sampling leave-one-out cross-validation

Pareto smoothed importance sampling **leave-one-out cross-validation (LOO-CV)** is a method that's used to approximate the LOO-CV results but without actually performing the K iterations. This is not an information criteria, but in practice provides results that are very similar to WAIC, and under certain general conditions, both WAIC and LOO converge asymptotically. Without going into too much detail, the main idea is that it is possible to approximate LOO-CV by re-weighting the likelihoods appropriately. This can be done using a very well-known and useful technique in statistics known as importance sampling. The problem is that the results are unstable. To fix this instability issue, a new method was introduced. This method uses a technique known as **Pareto smoothed importance sampling (PSIS)**, which can be used to compute more reliable estimates of LOO. The interpretation is similar to AIC and WAIC; the lower the value, the higher the estimated predictive accuracy of the model. Thus, we prefer models with lower values.

Other Information Criteria

Another common Information Criterion is **Deviance Information Criterion (DIC)**. If we use the *bayes-o-meter™*, this is somehow in the middle between AIC and WAIC. While still popular, WAIC has been proved more useful both theoretically and empirically than DIC, and thus using WAIC is recommended over DIC.

Yet another information criteria is the **Bayesian Information Criterion (BIC)**, which is similar to logistic regression and my mother's *sopa seca*. This name can be misleading. BIC was proposed as a way to correct some of the problems with AIC, and the author proposed a Bayesian justification for it. But BIC is not really Bayesian, and in fact is pretty similar to AIC. It also assumes flat priors and uses a maximum likelihood estimate. Even more importantly, BIC is different to AIC and WAIC and is more related to the concept of Bayes factors, something we will discuss later in this chapter.

Model comparison with PyMC3

Model comparison with ArviZ is more easily *done than said!*

```
waic_1 = az.waic(trace_1)
waic_1
```

	waic	waic_se	p_waic	warning
0	28.750381	5.303983	2.443984	0

If you want to compute LOO instead of WAIC, you must use `az.loo`.

For WAIC and LOO PyMC3 reports four values:

- A point estimate
- The standard error of the point estimate (this is computed by assuming normality and hence it may not be very reliable when the sample size is low)
- The effective number of parameters
- A warning (read the *A note on the reliability of WAIC and LOO computations* section for more details)

Since the values of WAIC/LOO are always interpreted in a relative fashion, that is, by comparing them across models, ArviZ provides two auxiliary functions to ease the comparison. The first one is `az.compare`:

```
cmp_df = az.compare({'model_l':trace_l, 'model_p':trace_p},
                    method='BB-pseudo-BMA')
cmp_df
```

	waic	pwaic	dwaic	weight	se	dse	warning
1	9.07	2.59	0	1	5.11	0	0
2	28.75	2.44	19.68	0	4.51	5.32	0

We have many columns, so let's check the meaning of them, one by one:

1. The first column clearly contains the values of WAIC. The DataFrame is always sorted from the lowest to the highest WAIC. The index reflects the order in which the models are passed to this function.
2. The second column is the estimated effective number of parameters. In general, models with more parameters will be more flexible to fit data and at the same time could also lead to overfitting. Thus, we can interpret pWAIC as a penalization term. Intuitively, we can also interpret it as a measure of how flexible each model is in fitting the data.
3. The third column is the relative difference between the value of WAIC for the top-ranked model and the value of WAIC for each model. For this reason, we will always get a value of 0 for the first model.
4. Sometimes, when comparing models, we do not want to select the *best* model. Instead, we want to perform predictions by averaging along all the models (or at least several models). Ideally, we would like to perform a weighted average, giving more weight to the model that seems to explain/predict the data better. There are many approaches to perform this task. One of them is to use Akaike weights based on the values of WAIC for each model. These weights can be loosely interpreted as the probability of each model (among the compared models), given the data. One caveat of this approach is that the weights are based on point estimates of WAIC (that is, the uncertainty is ignored).
5. The fifth column records the standard error for the WAIC computations. The standard error can be useful for assessing the uncertainty of the WAIC estimates.

6. In the same way that we can compute the standard error for each value of WAIC, we can compute the standard error of the differences between two values of WAIC. Notice that both quantities are not necessarily the same. The reason for this is that the uncertainty about WAIC is correlated between models. This quantity is always 0 for the top-ranked model.
7. Finally, we have the last column, named *warning*. A value of 1 indicates that the computation of WAIC may not be reliable. Read the *A note on the reliability of WAIC and LOO computations* section for further details.

We can also get similar information as a visualization by using the `az.plot_compare` function. This second convenience function takes the output of `az.compare` and produces a summary plot in the style of the one used in the book *Statistical Rethinking* by Richard McElreath:

```
az.plot_compare(cmp_df)
```

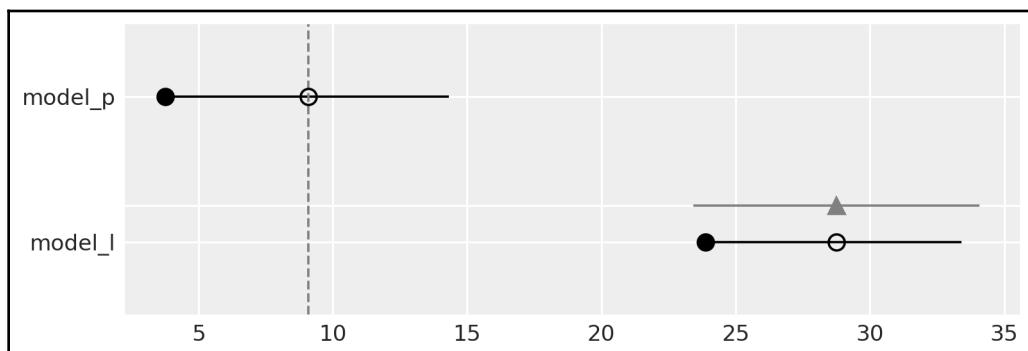


Figure 5.8

Let me describe the *Figure 5.8* in detail:

- The empty circle represents the values of WAIC and the black error bars associated with them are the values of the standard deviation of WAIC.
- The value of the lowest WAIC is also indicated with a vertical dashed grey line to ease comparison with other WAIC values.
- The filled in black dots are the in-sample deviance of each model, which for WAIC is $2 p\text{WAIC}$ from the corresponding WAIC value.
- For all models except the top-ranked one, we also get a triangle, indicating the value of the difference of WAIC between that model and the top model, and a grey error bar indicating the standard error of the differences between the top-ranked WAIC and WAIC for each model.

The simpler way to use information criteria is to perform model selection. Simply choose the model with the lower Information Criterion value and forget about any other model. If we follow this interpretation, this is a very easy choice—the quadratic model is the best. Notice that the standard errors do not overlap, giving us confidence about making this choice. If instead the standard errors were overlapping, we should provide a more nuanced answer.

A note on the reliability of WAIC and LOO computations

When computing WAIC or LOO, you may get a warning message, indicating that the result of either computation could be unreliable. This warning is raised based on a cut-off value that was determined empirically (see the *Keep reading* section for a reference). While it is not necessarily problematic, it could be an indication of a problem with the computation of these measures. WAIC and LOO are relative newcomers and we probably still need to develop better ways to access their reliability. Anyway, if this happens to you, first, make sure that you have enough samples and that you have a well-mixed, reliable sample (see Chapter 8, *Inference Engines*). If you still get those messages, the authors of the LOO method recommend using a more robust model, such as using a Student's t-distribution instead of a Gaussian one. If none of these recommendations work, then you may need to think about using another method, such as directly performing K-fold cross-validation.

On a more general note, WAIC and LOO can only help you choose among a given set of models, but they cannot help to decide if a model is really a good solution to our particular problem. For this reason, WAIC and LOO should be complemented with posterior predictive checks, along with any other information and tests that help us to put models and data in the light of the domain knowledge relevant to the particular problem we are trying to solve.

Model averaging

Model selection is appealing for its simplicity, but we are discarding information about the uncertainty in our models. This is somehow similar to computing the full posterior and then only keeping the mean of the posterior; we may become overconfident about what we really know. One alternative is to perform model selection but report and discuss the different models, along with the computed information criteria values, their standard error values, and perhaps also the posterior predictive checks. It is important to put all of these numbers and tests in the context of our problem so that we and our audience can have a better feel of the possible limitations and shortcomings of the models. If you are in the academic world, you can use this approach to add elements to the discussion section of a paper, presentation, thesis, and so on.

Yet another approach is to fully embrace the uncertainty in the model comparison and perform model averaging. The idea now is to generate a meta-model (and meta-predictions) using a weighted average of each model. One way to compute these weights is to apply the following formula:

$$w_i = \frac{e^{\frac{1}{2}dE_i}}{\sum_j^M e^{-\frac{1}{2}dE_j}} \quad (5.7)$$

Here, dE_i is the difference between the value of WAIC for the i -esim model, and the model with the lowest WAIC. Instead of WAIC, you can use any other Information Criterion you want, like AIC or other measures, like LOO. This formula is a heuristic way to compute the relative probability of each model (given a fixed set of models) from WAIC values (or other similar measures). Look at how the denominator is just a normalization term to ensure that the weights sum up to one. You may remember this expression from Chapter 4, *Generalizing Linear Models*, because it is just the softmax function. Using the weights from the preceding formula to average models is known as **pseudo Bayesian Modeling**

Averaging. The **true Bayesian Modeling Averaging** will be to use the marginal likelihoods instead of WAIC or LOO. Nevertheless, even when using the marginal likelihoods sounds theoretically appealing, there are theoretical and empirical reasons to prefer WAIC or LOO over the marginal likelihood for both model comparison and model averaging. You will find more details about this in the *Bayes factors* section.

Using PyMC3, you can compute the weights that are expressed in the preceding formula by passing the `method='pseudo-BMA'` (pseudo-Bayesian Modeling Averaging) argument to the `az.compare` function. One of the caveats of this formula is that it does not take into account the uncertainty in the computation of the values of E_i . Assuming a Gaussian approximation, we can compute the standard error for each E_i . These are the errors returned by the functions `az.waic`, `az.loo` and also by the function `az.compare` when the `method='pseudo-BMA'` argument is passed. We can also estimate uncertainty by using Bayesian bootstrapping. This is a more robust method than assuming normality. PyMC3 can compute this for you if you pass `method='BB-pseudo-BMA'` to the `az.compare` function.

A different approach to compute weights for averaging models is known as the stacking of *predictive distributions* or just **stacking**. This is implemented in PyMC3 by passing `method='stacking'` to `az.compare`. The basic idea is to combine several models in a metamodel by minimizing the divergence between the meta-model and the true generating model. When using a logarithmic scoring rule, this is equivalent to the following:

$$\max_n \frac{1}{n} \sum_{i=1}^n \log \sum_{k=1}^K w_k p(y_i | y_{-i}, M_k) \quad (5.8)$$

Here, n is the number of data points and K the number of models. To enforce a solution, we constrain w to be $w_k \geq 0$ and $\sum w_k = 1$. The quantity $p(y_i | y_{-i}, M_k)$ is the leave-one-out predictive distribution for the M_k model. As we already discussed, computing it requires fitting each model n times, each time leaving out one data point. Fortunately, we can approximate the exact leave-one-out predictive distribution using WAIC or LOO, and that is what PyMC3 does.

There are other ways to average models such as, for example, explicitly building a metamodel that includes all the models of interest as submodels. We can build such a model in such a way that we perform inference for the parameter of each submodel and at the same time, we compute the relative probability of each model (see the *Bayes Factor* section for an example of this).

Besides averaging discrete models, we can sometimes think of continuous versions of them. A toy example is to imagine that we have a coin-flipping problem and that we have two different models: one with a prior bias toward heads and the other toward tails. A continuous version of this will be a hierarchical model where the prior distribution is directly estimated from the data. This hierarchical model includes the discrete models as special cases.

Which approach is better? This depends on our concrete problem. Do we really have a good reason to think about discrete models, or is our problem better represented as a continuous model? Is it important for our problem to single out a model, because we are thinking in terms of competing explanations, or would averaging be a better idea because we are more interested in predictions, or can we truly think of the process generating process as an average of subprocesses? All of these questions are not answered by statistics, and are only informed by statistics in the context of domain knowledge.

The following is just a dummy example of how to get a weighed posterior predictive sample from PyMC3. Here, we are using the `pm.sample_posterior_predictive_w` function (notice the `w` at the end of the function's name). The difference between `pm.sample_posterior_predictive` and `pm.sample_posterior_predictive_w` is that the latter accepts more than one trace and model, as well as a list of weights (by default, the weights are the same for all models). You can get these weights from `az.compare` or any other source you want:

```
w = 0.5
y_lp = pm.sample_posterior_predictive_w([trace_l, trace_p],
                                         samples=1000,
                                         models=[model_l, model_p],
                                         weights=[w, 1-w])

_, ax = plt.subplots(figsize=(10, 6))
az.plot_kde(y_l, plot_kwarg={'color': 'C1'},
             label='linear model', ax=ax)
az.plot_kde(y_p, plot_kwarg={'color': 'C2'},
             label='order 2 model', ax=ax)
az.plot_kde(y_lp['y_pred'], plot_kwarg={'color': 'C3'},
             label='weighted model', ax=ax)

plt.plot(y_1s, np.zeros_like(y_1s), '|', label='observed data')
plt.yticks([])
plt.legend()
```

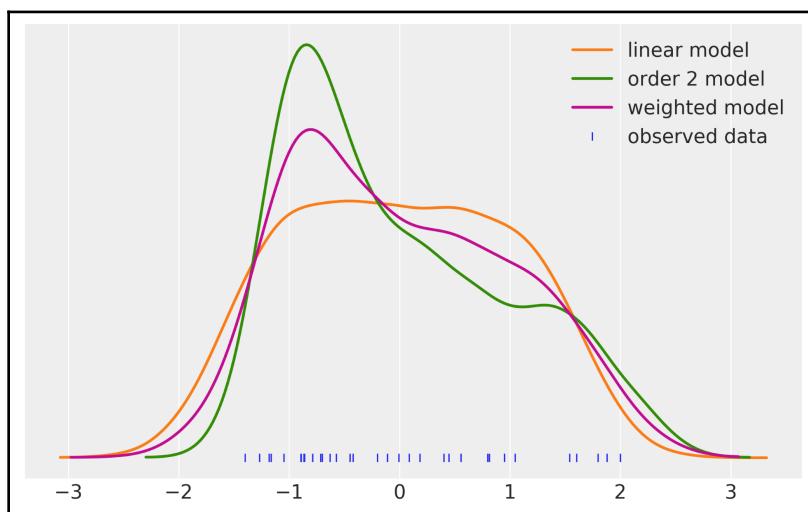


Figure 5.9

I said this is a dummy example because the quadratic model has such a lower value of WAIC compared to the linear model that the weight is basically 1 for the first model and 0 for the latter and to generate *Figure 5.9*, I have assumed that both models have the same weight.

Bayes factors

A common alternative to evaluate and compare models in the Bayesian world (at least in some of its countries) are Bayes factors. To understand what Bayes factors are, let's write Bayes' theorem one more time (we have not done so for a while!):

$$p(\theta | y) = \frac{p(y | \theta)p(\theta)}{p(y)} \quad (5.9)$$

Here, y represents the data. We can make the dependency of the inference on a given M model explicit and write:

$$p(\theta | y, M_k) = \frac{p(y | \theta, M_k)p(\theta | M_k)}{p(y | M_k)} \quad (5.10)$$

The term in the denominator is known as **marginal likelihood** (or **evidence**), as you may remember from the first chapter. When doing inference, we do not need to compute this normalizing constant, so in practice, we often compute the posterior up to a constant factor. However, for model comparison and model averaging, the marginal likelihood is an important quantity. If our main objective is to choose only one model, the best one, from a set of k models, we can just choose the one with the largest $p(y | M_k)$. As a general rule, $p(y | M_k)$ are tiny numbers and do not tell us too much on their own; like with information criteria, what matters are the relative values. So, in practice, people often compute the ratio of two marginal likelihoods, and this is called a **Bayes factor**:

$$BF = \frac{p(y | M_0)}{p(y | M_1)} \quad (5.11)$$

When $BF > 1$, model 0 explains data better than model 1.

Some authors have proposed tables with ranges to discretize and ease *BF* interpretation. For example the following bullet-list indicates the strength of the evidence, favoring model 0 against model 1:

- 1-3: Anecdotal
- 3-10: Moderate
- 10-30: Strong
- 30-100: Very strong
- > 100: Extreme

Remember, these rules are just conventions, simple guides at best. However, the results should always be put into context and should be accompanied with enough details that others could potentially check if they agree with our conclusions. The evidence that's necessary to make a claim is not the same in particle physics, or a court, or in a plan to evacuate a town to prevent hundreds of deaths.

Using $p(y | M_k)$ to compare models is totally fine if all of the models are assumed to have the same prior probability. Otherwise, we have to compute the *posterior odds*:

$$\underbrace{\frac{p(M_0 | y)}{p(M_1 | y)}}_{\text{posterior odds}} = \underbrace{\frac{p(y | M_0)}{p(y | M_1)}}_{\text{Bayes factors}} \underbrace{\frac{p(M_0)}{p(M_1)}}_{\text{prior odds}} \quad (5.12)$$

Some remarks

Now, we will briefly discuss some key facts about the marginal likelihood. By carefully inspecting the definition of marginal likelihood, we can understand their properties and consequences for their practical use:

$$p(y | M_k) = \int_{\theta_k} p(y | \theta_k, M_k) p(\theta_k, M_k) d\theta_k \quad (5.13)$$

- **The good:** Models with more parameters have a larger penalization than models with fewer parameters. Bayes factor has a built-in Occam's Razor! The intuitive reason for this is that the larger the number of parameters, the more spread the prior with respect to the likelihood. Thus, when computing the integral in the preceding formula, you will get a smaller value with a more concentrated prior.

- **The bad:** Computing the marginal likelihood is, generally, a hard task since the preceding formula is an integral of a highly variable function over a high dimensional parameter space. In general, this integral needs to be solved numerically using more or less sophisticated methods.
- **The ugly:** The marginal likelihood depends *sensitively* on the values of the priors.

Using the marginal likelihood to compare models is a good idea because a penalization for complex models is already included (thus preventing us from overfitting). At the same time, a change in the prior will affect the computations of the marginal likelihood. At first, this sounds a little bit silly—we already know that priors affect computations (otherwise, we could simply avoid them), but the point here is the word *sensitively*. We are talking about changes in the prior that will keep the inference of θ more or less the same, but could have a big impact on the value of the marginal likelihood. You may have noticed in the previous examples that, in general, having a normal prior with a standard deviation of 100 is the same as having one with a standard deviation of 1,000. Instead, Bayes factors will be affected by these kind of changes.

Another source of criticism regarding Bayes factors is that they can be used as a Bayesian way of doing hypothesis testing. There is nothing wrong with this *per se*, but many authors have pointed out that an inference approach, similar to the one used in this book and other books like *Statistical Rethinking* by Richard McElreath, is better suited to most problems than the hypothesis testing approach (whether Bayesian or not).

Having said all this, let's look at how we can compute Bayes factors.

Computing Bayes factors

The computation of Bayes factors can be framed as a hierarchical model, where the high-level parameter is an index that's assigned to each model and sampled from a categorical distribution. In other words, we perform inference of the two (or more) competing models at the same time and we use a discrete variable that jumps between models. How much time we spend sampling each model is proportional to $p(M_k | y)$. Then, we apply equation 5.10 to get Bayes factors.

To exemplify the computation of the Bayes factors, we are going to flip coins one more time:

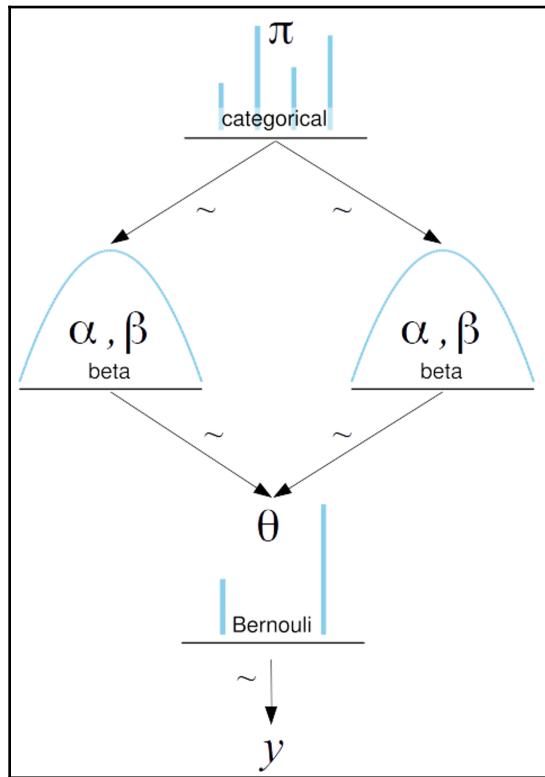


Figure 5.10

Notice that while we are computing Bayes factors between models that differ only on the prior, the models could differ on the likelihood, or even both. This idea is the same.

Let's create some data so that we can use it in an example:

```

coins = 30
heads = 9
y_d = np.repeat([0, 1], [coins-heads, heads])
  
```

Now, let's look at the PyMC3 model. To switch between priors, we are using the `pm.math.switch` function. If the first argument of this function evaluates to true, then the second argument is returned, otherwise the third argument is returned. Notice that we are also using the `pm.math.eq` function to check whether the `model_index` variable is equal to 0:

```
with pm.Model() as model_BF:
    p = np.array([0.5, 0.5])
    model_index = pm.Categorical('model_index', p=p)

    m_0 = (4, 8)
    m_1 = (8, 4)
    m = pm.math.switch(pm.math.eq(model_index, 0), m_0, m_1)

    # a priori
    θ = pm.Beta('θ', m[0], m[1])
    # likelihood
    y = pm.Bernoulli('y', θ, observed=y_d)

    trace_BF = pm.sample(5000)
az.plot_trace(trace_BF)
```

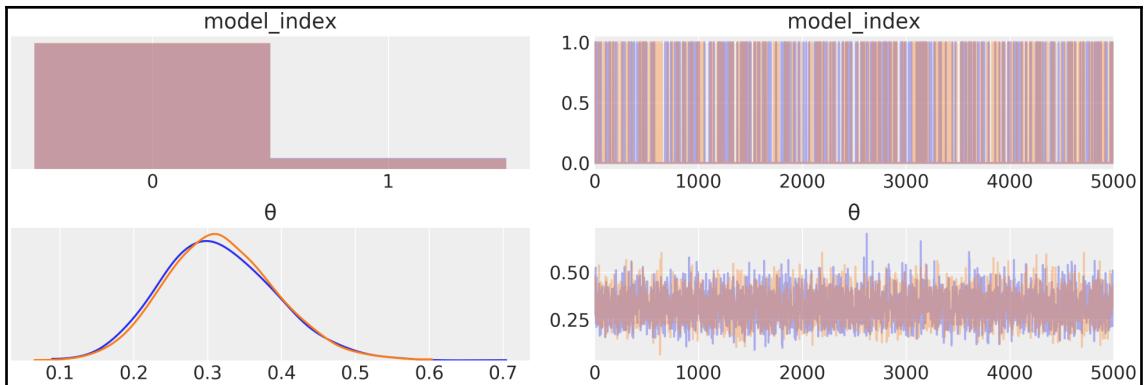


Figure 5.11

Now, we need to compute the Bayes factor by counting the `model_index` variable. Notice that we have included the values of the priors for each model:

```
pM1 = trace_BF['model_index'].mean()
pM0 = 1 - pM1
BF = (pM0 / pM1) * (p[1] / p[0])
```

As a result, we get a value of ≈ 11 , meaning that model 0 is favored over model 1 by one order of magnitude. This makes total sense since the data has fewer values of heads than expected for $\theta = 0.5$ and the only difference between both models is that the prior of model 0 is more compatible with $\theta < 0.5$ (more tails than heads) and that model 1 is more compatible with $\theta > 0.5$ (more heads than tails).

Common problems when computing Bayes factors

Some common problems when computing Bayes factors the way we did is that if one model is better than the other, by definition, we will spend more time sampling from it than from the other model. This could be problematic because we can undersample one of the models. Another problem is that the values of the parameters get updated, even when the parameters are not used to fit that model. That is, when model 0 is chosen, the parameters in model 1 are updated, but since they are not used to explain the data, they only get restricted by the prior. If the prior is too vague, it is possible that when we choose model 1, the parameter values are too far away from the previous accepted values and hence the step is rejected. Therefore, we end up having a problem with sampling.

In case we encounter these problems, we can perform two modifications to our model to improve sampling:

- Ideally, we can get a better sampling of both models if they are visited equally, so we can adjust the prior for each model (the p variable in the previous model) in such a way to favor the less favorable model and disfavor the most favorable model. This will not affect the computation of the Bayes factor because we are including the priors in the computation.
- Use pseudo priors, as suggested by Kruschke and others. The idea is simple: if the problem is that the parameters drift away unrestricted, when the model they belong to is not selected, then one solution is to try to restrict them artificially, but only when not used! You can find an example of using pseudo priors in a model used by Kruschke in his book and ported by me to Python/PyMC3 at https://github.com/aloctavodia/Doing_bayesian_data_analysis.

Using Sequential Monte Carlo to compute Bayes factors

Another route to compute Bayes factors is by using a sampling method known as **Sequential Monte Carlo (SMC)**. We are going to learn the details of this method in Chapter 8, *Inference Engines*. For now, we just need to know that this sampler computes an estimation of the marginal likelihood as a by-product that we can directly use to compute Bayes factors. To use SMC in PyMC3, we just need to pass `pm.SMC()` to the `step` argument of `sample`:

```
with pm.Model() as model_BF_0:  
    θ = pm.Beta('θ', 4, 8)  
    y = pm.Bernoulli('y', θ, observed=y_d)  
    trace_BF_0 = pm.sample(2500, step=pm.SMC())  
  
with pm.Model() as model_BF_1:  
    θ = pm.Beta('θ', 8, 4)  
    y = pm.Bernoulli('y', θ, observed=y_d)  
    trace_BF_1 = pm.sample(2500, step=pm.SMC())  
  
model_BF_0.marginal_likelihood / model_BF_1.marginal_likelihood
```

According to the SMC method, the Bayes factor is also around 11. If you want to compute Bayes factors with PyMC3, I strongly recommend using the SMC method. The other method presented in this book is more computationally cumbersome and requires more manual tweaking, mostly because the *jumping* between models requires more tuning by trial and error by the user. SMC, on the other hand, is a more automated method.

Bayes factors and Information Criteria

Notice that if we take the logarithm of Bayes factors, we can turn the ratio of the marginal likelihood into a difference. Comparing differences in marginal likelihoods is similar to comparing differences in information criteria. Moreover, we can interpret the Bayes factors, or to be more precise, the marginal likelihoods, as having a fitting term and a penalizing term. The term indicating how well the model fits the data is the likelihood part, and the penalization part comes from averaging over the prior. The larger the number of parameters, the larger the prior volume compared to the likelihood volume, and hence we will end up taking averages from zones where the likelihood has very low values. The more parameters, the more diluted or diffused the prior and hence the greater the penalty when computing the evidence. This is the reason people say Bayes' theorem leads to a natural penalization of complex models, that is, Bayes theorem comes with a built-in Occam's razor.

We have already said that Bayes factors are more sensitive to priors than many people like (or even realize). It is like having differences that are practically irrelevant when performing inference but that turn out to be important when computing Bayes factors. If there is an infinite multiverse, I am almost sure that there is a Geraldo talk show with Bayesians fighting and cursing each other about Bayes factors. In that universe (well... also in this one) I will be cheering for the anti-BF side. Nevertheless, now, we are going to look at an example that will help clarify what Bayes factors are doing and what information criteria are doing and how, while similar, they focus on two different aspects. Go back to the definition of the data for the coin flip example and now set 300 coins and 90 heads; this is the same proportion as before, but we have 10 times more data. Then, run each model separately:

```
traces = []
waics = []
for coins, heads in [(30, 9), (300, 90)]:
    y_d = np.repeat([0, 1], [coins-heads, heads])
    for priors in [(4, 8), (8, 4)]:
        with pm.Model() as model:
            θ = pm.Beta('θ', *priors)
            y = pm.Bernoulli('y', θ, observed=y_d)
            trace = pm.sample(2000)
            traces.append(trace)
            waics.append(az.waic(trace))
```

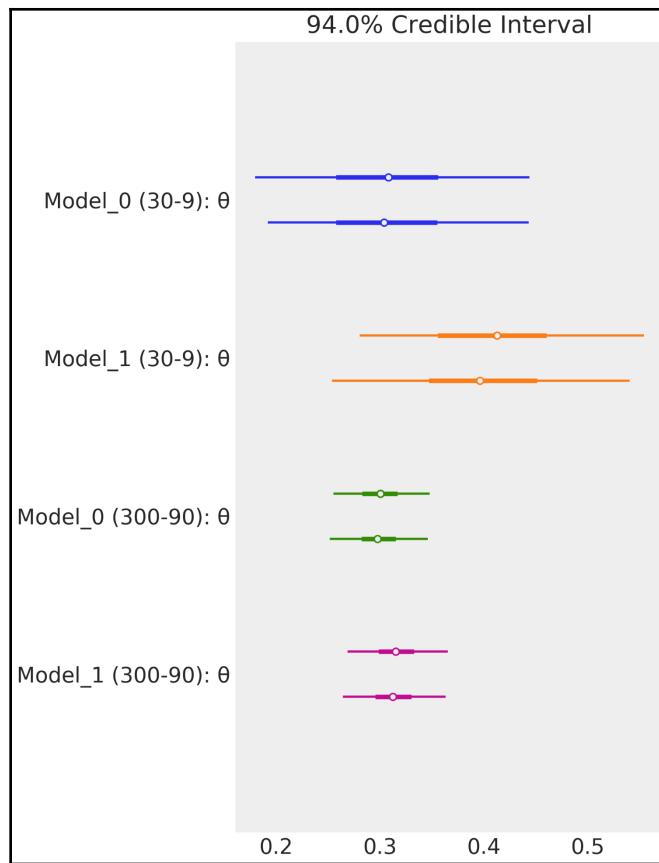


Figure 5.12

By adding more data, we have almost entirely overcome the prior and now both models make similar predictions. Using 30 coins and 9 heads as our data, we saw a BF of 11. If we repeat the computation (feel free to do it by yourself) with the data of 300 coins and 90 heads, we will see a BF of ≈ 25 . The Bayes factor is saying that model 0 is favored over model 1 even more than before. When we increase the data, the decision between models becomes clearer. This makes total sense because now we are more certain that model 1 has a prior that does not agree with the data.

Also notice that as we increase the amount of data, both models will tend to agree on the value of θ ; in fact, we get ≈ 0.3 with both models. Thus, if we decide to use θ to predict new outcomes, it will barely make any difference from which model we compute the distribution of θ .

Now, let's compare what WAIC is telling us (see *Figure 5.13*). WAIC is ≈ 368.4 for model 0 and ≈ 368.6 for model 1. This intuitively sounds like a small difference. What's more important than the actual difference is that if you compute the information criteria again for the data, that is, 30 coins and 9 heads, you will get something like ≈ 38.1 for model 0 and ≈ 39.4 for model 1 with WAIC. That is, the relative difference when increasing the data becomes smaller—the more similar the estimation of θ , are the more similar the values for the predictive accuracy estimated by the information criteria. You will observe essentially the same if you use LOO instead of WAIC:

```
fig, ax = plt.subplots(1, 2, sharey=True)

labels = model_names
indices = [0, 0, 1, 1]
for i, (ind, d) in enumerate(zip(indices, waics)):
    mean = d.waic
    ax[ind].errorbar(mean, -i, xerr=d.waic_se, fmt='o')
    ax[ind].text(mean, -i+0.2, labels[i], ha='center')

ax[0].set_xlim(30, 50)
ax[1].set_xlim(330, 400)
plt.ylim([-i-0.5, 0.5])
plt.yticks([])
plt.subplots_adjust(wspace=0.05)
fig.text(0.5, 0, 'Deviance', ha='center', fontsize=14)
```

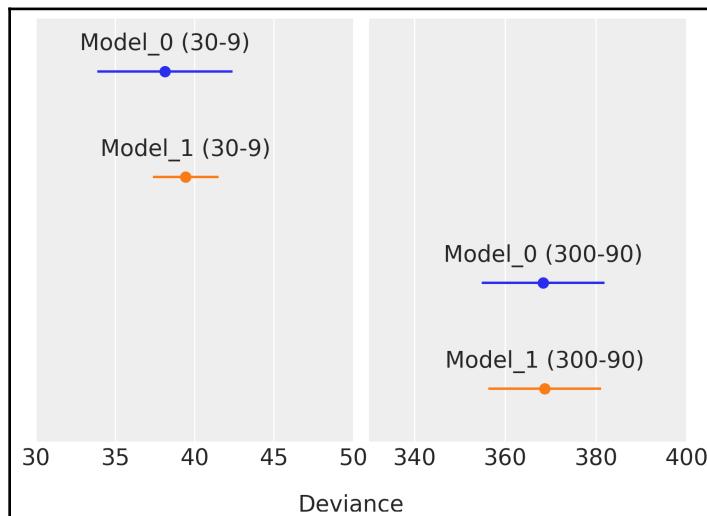


Figure 5.13

Bayes factors are focused on which model is better while WAIC (and LOO) is focused on which model will give the better predictions. You can see these differences if you go back and check equations 5.6 and 5.11. WAIC, like other information criteria, uses log-likelihood in one way or another, and the priors are not directly part of the computations. Priors only participate indirectly by helping us estimate the value of θ . Instead, Bayes factors use priors directly as we need to average the likelihood over the entire range of prior values.

Regularizing priors

Using informative and weakly informative priors is a way of introducing bias in a model and, if done properly, this can be a really good because bias prevents overfitting and thus contributes to models being able to make predictions that generalize well. This idea of adding a bias to reduce a generalization error without affecting the ability of the model to adequately model the data that's used to fit is known as **regularization**. This regularization often takes the form of penalizing larger values for the parameters in a model. This is a way of reducing the information that a model is able to represent and thus reduces the chances that a model captures the noise instead of the signal.

The regularization idea is so powerful and useful that it has been discovered several times, including outside the Bayesian framework. In some fields, this idea is known as **Tikhonov regularization**. In non-Bayesian statistics, this regularization idea takes the form of two modifications on the least square method, known as **ridge regression** and **Lasso regression**. From the Bayesian point of view, a ridge regression can be interpreted as using normal distributions for the beta coefficients (of a linear model), with a small standard deviation that pushes the coefficients toward zero. In this sense, we have been doing something similar to ridge regression for every single linear model in this book (except the examples in this chapter that uses SciPy!). On the other hand, Lasso regression can be interpreted from a Bayesian point of view as the MAP of the posterior computed from a model with Laplace priors for the beta coefficients. The Laplace distribution *looks similar* to the Gaussian distribution, but its first derivative is undefined at zero because it has a very sharp peak at zero (see *Figure 5.14*). The Laplace distribution concentrates its probability mass much closer to zero compared to the normal distribution. The idea of using such a prior is to provide both regularization and **variable selection**. The idea is that since we have this peak at zero, we expect the prior to induce sparsity, that is, we create a model with a lot of parameters and the prior will automatically makes most of them zero, keeping only the *relevant* variables contributing to the output of the model. Unfortunately, the Bayesian Lasso does not really work like this, basically because in order to have many parameters, the Laplace prior is forcing the non-zero parameters to be small. Fortunately, not everything is lost—there are Bayesian models that can be used for inducing sparsity and performing variable selection, like the horseshoe and the Finnish horseshoe.

It is important to notice that the classical versions of ridge and Lasso regressions correspond to single point estimates, while the Bayesian versions gave a full posterior distribution as a result:

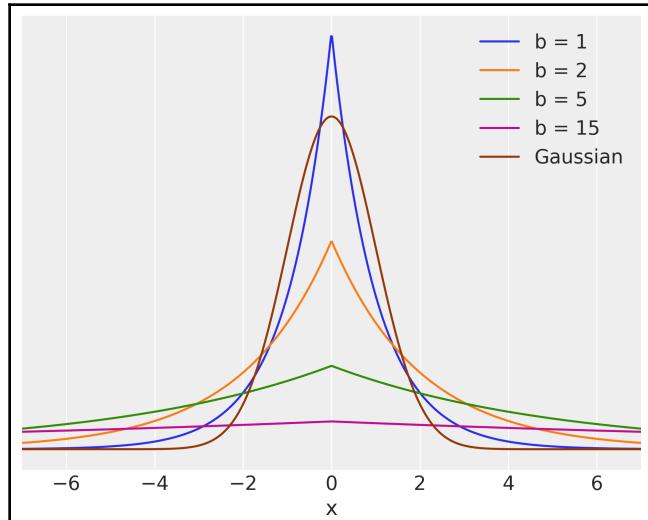


Figure 5.14

WAIC in depth

If we expand equation 5.6, we get the following:

$$WAIC = -2 \sum_i^n \log \left(\frac{1}{S} \sum_{s=1}^S p(y_i | \theta^s) \right) + 2 \sum_i^n \left(\sum_{s=1}^S V(\log p(y_i | \theta^s)) \right) \quad (5.14)$$

Both terms in this expression look very similar. The first one, the lppd (log point-wise predictive density), is computing the mean likelihood over the S posterior samples. We do this for each data point and then we take the logarithm and sum up over all data points. Please compare this term with equations 5.3 and 5.4. This is just what we call *deviance*, but computed, taking into account the posterior. Thus, if we accept that computing the log-likelihood is a good way to measure the appropriateness of the fit of a model, then computing it from the posterior is a logic path for a Bayesian approach. As we already said, the lddp of observed data y is an overestimate of the lppd for future data. Thus, we introduce a second term to correct the overestimation. The second term computes the variance of the log-likelihood over the S posterior samples. We do this for each data point and then we sum up over all data points. Why does the variance give a penalization term? Well, the intuition is similar to that of the Bayes factor's built-in Occam's Razor. The larger the number of effective parameters, the greater the spread of the posterior will be. When we add structure to a model such as with informative/regularizing priors or hierarchical dependencies, we are restricting the posterior and thus decreasing the effective number of parameters in comparison with a similar unregularized or less structured model.

Entropy

Now, I would like to briefly talk about the concept of **entropy**. Mathematically, we can define it as follows:

$$H(p) = - \sum_i p_i \log(p_i) \quad (5.15)$$

Intuitively, the more *spread* a distribution is, the larger its entropy. We can see this by running the following block of code and inspecting *Figure 5.15*:

```
np.random.seed(912)
x = range(0, 10)
q = stats.binom(10, 0.75)
r = stats.randint(0, 10)

true_distribution = [list(q.rvs(200)).count(i) / 200 for i in x]

q_pmf = q.pmf(x)
r_pmf = r.pmf(x)

_, ax = plt.subplots(1, 3, figsize=(12, 4), sharey=True,
                    constrained_layout=True)

for idx, (dist, label) in enumerate(zip([true_distribution, q_pmf, r_pmf],
                                         ['true_distribution', 'q', 'r'])):
```

```

ax[idx].vlines(x, 0, dist, label=f'entropy = {stats.entropy(dist):.2f}')
ax[idx].set_title(label)
ax[idx].set_xticks(x)
ax[idx].legend(loc=2, handlelength=0)

```

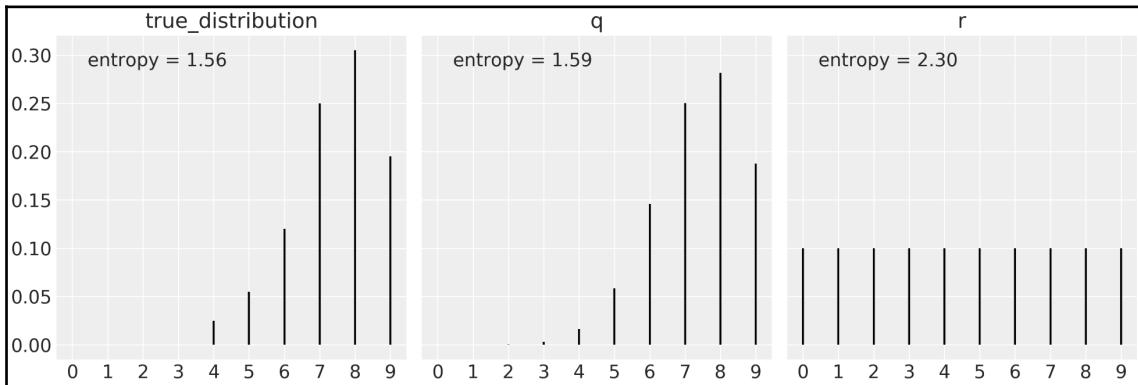


Figure 5.15

As we can see, distribution *r* in the preceding diagram is the more spread of the three distributions and is also the one with the largest entropy. I suggest that you play with the code and explore how entropy changes (see exercise 10 for more information).

Following the previous example, you may be tempted to declare entropy as a weird form of measuring the variance of a distribution. While both concepts are related, they are not the same. Under some circumstances, an increase of entropy means an increase of the variance. This will be the case for a Gaussian distribution. However, we can also have examples where the variance increases and the entropy doesn't. We can understand why this happens without being very rigorous. Let's suppose that we have a distribution that is a mixture of two Gaussian's (we will discuss mixture distribution in detail in [Chapter 6, Mixture Models](#)). As we increase the distance between the modes, we increase the distance of the *bulk of points* from the mean, and the variance is precisely the average distance of all *points* to the mean. So, if we keep increasing the distance, the variance will keep increasing without limit. The entropy will be less affected because as we increase the distance between the modes, the *points* between the modes have less and less probability and thus their contribution to the total entropy will be *negligible*. From the perspective of the entropy, if we start from two overlapped Gaussian's and then *move* one in respect to the other, at some point, we will have two separated Gaussian's.

Entropy is also related to the concept of information, or its counterpart, uncertainty. In fact, we have been saying through this book that a more spread or flat prior distribution is a less informative one. This is not only intuitively correct but also has the theoretical support of the concept of entropy. In fact, there is a tribe among Bayesians that use entropy to justify their weakly informative or regularizing priors. This is usually known as the Maximum Entropy principle. We want to find the distribution with the largest possible entropy (the least informative), but we also want to take into account restraints that have been defined by our problem. This is an optimization problem that can be solved mathematically, but we will not look at the details of those computations. I will instead provide some examples. The distributions with the largest entropy under the following constraints are:

- **No constraints:** Uniform (continuous or discrete, according to the type of variable)
- **A positive mean:** Exponential
- **A given variance:** Normal distribution
- **Only two unordered outcomes and a constant mean:** Binomial, or the Poisson if we have rare events (remember that the Poisson is a special case of the binomial)

It is interesting to note that many of the generalized linear models like the ones we saw in Chapter 4, *Generalizing Linear Models* are traditionally defined using maximum entropy distributions, given the constraints of the models.

Kullback-Leibler divergence

Now, I would like to briefly talk about the **Kullback-Leibler (KL) divergence**, or just KL divergence. This is a concept that you may find when reading about statistics, machine learning, information theory, or statistical mechanics. You may argue that the reason for the recurrence of the KL-divergence as well as other concepts like entropy or the marginal likelihood is simply that, at least partially, all of these disciplines are discussing the same sets of problems, just under slightly different perspectives.

The KL divergence is useful because it is a way of measuring how close two distributions are, and is defined as follows:

$$D_{KL}(p||q) = \sum_i p_i \log \frac{p_i}{q_i} \quad (5.16)$$

This reads as the Kullback-Leibler divergence from q to p (yes, you have to read it *backwards*), where p and q are two probability distributions. For continuous variables, instead of a summation, you should compute an integral, but the main idea is the same.

We can interpret the $D_{KL}(p||q)$ divergence as the *extra* entropy or uncertainty that's introduced by using the probability distribution q to approximate the distribution p . In fact, the KL divergence is the difference between two entropies:

$$D_{KL}(p||q) = \underbrace{\sum_i p_i \log p_i}_{\text{entropy of } p} - \underbrace{\sum_i p_i \log q_i}_{\text{crossentropy of } p,q} = \sum_i p_i (\log p_i - \log q_i) \quad (5.17)$$

By using the properties of the logarithms, we can rearrange equation 5.17 to recover equation 5.16. For this reason, we can also read $D_{KL}(p||q)$ as the relative entropy of p with respect to q (this time, we read it *forward*).

As a simple example, we can use KL-divergence to evaluate which distribution, q or r , is a better approximation to `true_distribution`. Using Scipy, we can compute $D_{KL}(\text{true_distribution}||q)$ and $D_{KL}(\text{true_distribution}||r)$:

```
stats.entropy(true_distribution, q_pmf), stats.entropy(true_distribution,
r_pmf)
```

If you run the previous block of code, you will get $\approx 0.0096, \approx 0.7394$. Thus, we can conclude that q is a better approximation to `true_distribution` than r , because it is the one introducing less extra uncertainty. I hope that you agree with me regarding the fact that this numerical result agrees with what you expected by inspecting *Figure 5.15*.

You may be tempted to describe the KL-divergence as a distance, but it is not symmetric and thus is not a real distance. If you run the following block of code, you will get $\approx 2.7, \approx 0.7$. As you can see, these numbers are not the same. In this example, we can see that r is a better approximation of q —this is the other way around:

```
stats.entropy(r_pmf, q_pmf), stats.entropy(q_pmf, r_pmf)
```

$D_{KL}(p||q)$ indicates how well q approximates p , and we can also think of it in terms of surprise, that is, how surprised we will be if we see q when we expect p . How surprised we are about an event depends on the information we use to judge that event. I grew up in a very arid city with maybe one or two *real* rain storms a year. Then, I moved to another province to go to college and I was really *shocked* that, at least during the wet season, there was, on average, one *real* rain storm every week! Some of my classmates were from Buenos Aires, one of the most humid and rainy provinces in Argentina. For them, the frequency of rain was more or less expected. What's more, they thought that it could rain a little more as the air was not *that* humid.

We could use the KL-divergence to compare models, as this will give a measure of the posterior from which model is closer to the *true distribution*. The problem is that we do not know the *true distribution*. Therefore, the KL-divergence is not directly applicable.

Nevertheless, we can use it as an argument to justify the use of the deviance (expression 5.3). If we assume that a *true distribution* exists, as shown in the following equation, then the *true distribution* is independent of any model and constants, and thus it will affect the value of the KL-divergence in the same way, irrespective of the (posterior) distribution we use to approximate the *true distribution*. Thus, we can use the deviance, that is, the part that depends on each model, to estimate how close we are of the *true distribution*, even when we do not know it. From equation 5.17 and by using a little bit of algebra, we can see the following:

$$\begin{aligned} D_{KL}(p||q) - D_{KL}(p||r) &= \left(\sum_i p_i \log p_i - \sum_i p_i \log q_i \right) - \left(\sum_i p_i \log p_i - \sum_i p_i \log r_i \right) \\ &= \sum_i p_i \log q_i - \sum_i p_i \log r_i \end{aligned} \quad (5.18)$$

Even if we do not know p , we can conclude that the distribution with the larger $\log(\cdot)$ or $\log-$ likelihood or deviance if you want—is the distribution closer in KL-divergence to the *true distribution*. In practice, the log-likelihood/deviance is obtained from a model that's been fitted to a finite sample. Therefore, we must also add a penalization term to correct the overestimation of the deviance, and that leads us to WAIC and other information criteria.

Summary

Posterior predictive checks is a general concept and practice that can help us understand how well models are capturing the data and how well the model is capturing the aspects of a problem we are interested in. We can perform posterior predictive checks with just one model or with many models, and thus we can use it as a method for model comparison. Posterior predictive checks are generally done via visualizations, but numerical summaries like *Bayesian p-values* can also be helpful.

Good models have a good balance between complexity and predictive accuracy. We exemplified this feature by using the classical example of polynomial regression. We discussed two methods to estimate the out-of-sample accuracy without leaving data aside: *cross-validation* and *information criteria*. We focused our discussion on the latter. From a practical point of view, information criteria is a family of methods that's balancing two contributions: one that measures how well a model fits the data and the other penalizing complex models. From the many information criteria available, WAIC is the most useful for Bayesian models. Another measure that is useful and, in practice, provides very similar results to WAIC is PSIS-LOO-CV (or LOO). This is a method that's used to approximate leave-one-out cross validation without the high computational cost of actually refitting a model several times. WAIC and LOO can be used for model selection and can also be helpful for model averaging. Instead of selecting a single *best* model, model averaging is about combining all available models by taking a weighted average of them.

A different approach to model selection, comparison, and model averaging is Bayes factors, which are the ratio of the marginal likelihoods of two models. Bayes factor computations can be really challenging. In this chapter, we showed two routes to compute them with PyMC3: a hierarchical model where we directly try to estimate the relative probability of each model using a discrete index, and a sampling method known as Sequential Monte Carlo. We suggested using the latter.

Besides being computationally challenging to compute Bayes factors are problematic to use, given that they are very sensitive to prior specification, we also compare Bayes factors and Information Criteria, and walked you through an example whereby they solve two related but different questions—one puts the focus on identifying the *right* model and the other on the best predictions or lower generalization loss. None of these methods are free of problems, but WAIC and LOO are much more robust in practice.

We briefly discussed how priors are related to the subject of overfitting, bias, and regularization with regard to the important subject of building models with good generalization properties.

Finally, we closed this book with a more in-depth discussion of WAIC, including a commentary of the interrelated concepts of entropy, the maximum entropy principle, and the KL divergence.

Exercises

1. This exercise is about regularization priors. In the code that generates the data, change `order=2` to another value, such as `order=5`. Then, fit `model_p` and plot the resulting curve. Repeat this, but now using a prior for beta with `sd=100` instead of `sd=1` and plot the resulting curve. How are both curves different? Try this out with `sd=np.array([10, 0.1, 0.1, 0.1, 0.1])`, too.
2. Repeat the previous exercise but increase the amount of data to 500 data points.
3. Fit a cubic model (order 3), compute WAIC and LOO, plot the results, and compare them with the linear and quadratic models.
4. Use `pm.sample_posterior_predictive()` to rerun the PPC example, but this time, plot the values of `y` instead of the values of the mean.
5. Read and run the posterior predictive example from PyMC3's documentation at https://pymc-devs.github.io/pymc3/notebooks/posterior_predictive.html. Pay special attention to the use of the Theano shared variables.
6. Go back to the code that generated *Figure 5.5* and *Figure 5.6*, and modify it to get new sets of six data points. Visually evaluate how the different polynomials fit these new datasets. Relate the results to the discussions in this book.
7. Read and run the model averaging example from PyMC3's documentation at https://docs.pymc.io/notebooks/model_averaging.html.
8. Compute the Bayes factor for the coin problem using a uniform prior beta (1, 1) and priors such as beta (0.5, 0.5). Set 15 heads and 30 coins. Compare this result with the inference we got in the first chapter of this book.
9. Repeat the last example where we compare Bayes factors and Information Criteria, but now reduce the sample size.
10. For the entropy example, change the `q` distribution. Try this with distributions like `stats.binom(10, 0.5)` and `stats.binom(10, 0.25)`.

6

Mixture Models

"...the father has the form of a lion, the mother of an ant; the father eats flesh, and the mother herbs. And these breed the ant-lion..."

-From The Book of Imaginary Beings

The River Plate (also known as **La Plata River** or **Río de la Plata**) is the widest river on Earth and a natural border between Argentina and Uruguay. During the late 19th century, the port area along this river was a place where natives mixed with Africans (most of them slaves) and European immigrants. One consequence of this encounter was the mix of European music, such as the Waltz and Mazurka, with the African Candombe and Argentinian Milonga (which, in turn, is a mix of Afro-American rhythms), giving origin to a dance and music we now call the Tango.

Mixing previous existing elements is a great way to create new stuff, not only music. In statistics, mixture models are one common approach to model building. These models are built by mixing simpler distributions to obtain more complex ones. For example, we can combine two Gaussians to describe a bimodal distribution or many Gaussians to describe arbitrary distributions. While using Gaussians is very common, we can mix, in principle, any family of distributions we want. Mixture models are used for different purposes, such as directly modeling sub-populations or as a useful trick for handling complicated distributions that cannot be described with simpler distributions.

In this chapter, we will cover the following topics:

- Finite mixture models
- Infinite mixture models
- Continuous mixture models

Mixture models

Mixture models naturally arise when the overall population is a combination of distinct sub-populations. A very familiar example is the distribution of heights in a given adult human population, which can be described as a mixture of female and male sub-populations. Another classical example is the clustering of handwritten digits. In this case, it is very reasonable to expect 10 sub-populations, at least in a base 10 system! If we know to which sub-population each observation belongs, it is generally a good idea to use that information to model each sub-population as a separate group. However, when we do not have direct access to this information is when mixture models come in handy.



Many datasets cannot be properly described using a single probability distribution, but they can be described as a mixture of such distributions. Models that assume data comes from a mixture of distributions are known as **mixture models**.

When building a mixture model, it is not really necessary to *believe* we are describing *true* sub-populations in the data. Mixture models can also be used as a statistical *trick* to add flexibility to our toolbox. Take, for example the Gaussian distribution. We can use it as a reasonable approximation for many unimodal and more or less symmetrical distributions. But what about multimodal or skewed distributions? Can we use Gaussian distributions to model them? Yes, we can, if we use a mixture of Gaussians. In a Gaussian mixture model, each component will be a Gaussian with a different mean and generally (but not necessarily) a different standard deviation. By combining Gaussians, we can add flexibility to our model in order to fit complex data distributions. In fact, we can approximate any distribution we want by using a proper combination of Gaussians. The exact number of distributions will depend on the accuracy of the approximation and the details of the data. In fact, we have been applying this idea of a mixture of Gaussians in many of the plots throughout this book. The **Kernel Density Estimation (KDE)** technique is a non-Bayesian (and non-parametric) implementation of this idea. Conceptually, when we call `az.plot_kde`, the function is placing a Gaussian (with a fixed variance) *on top* of each data point and then it is summing all the individual Gaussians to approximate the empirical distribution of the data. *Figure 6.1* shows an actual example of how we can mix eight Gaussians to represent a complex distribution, like a boa constrictor digesting an elephant (if you do not get the reference I strongly recommend that you get a copy of the book, *The Little Prince*). In *Figure 6.1*, all Gaussians have the same variance and they are centered at the orange dots, which are representing sample points from a possible unknown population. If you look carefully at *Figure 6.1*, you may notice that two of the Gaussians are basically one on top of the other:

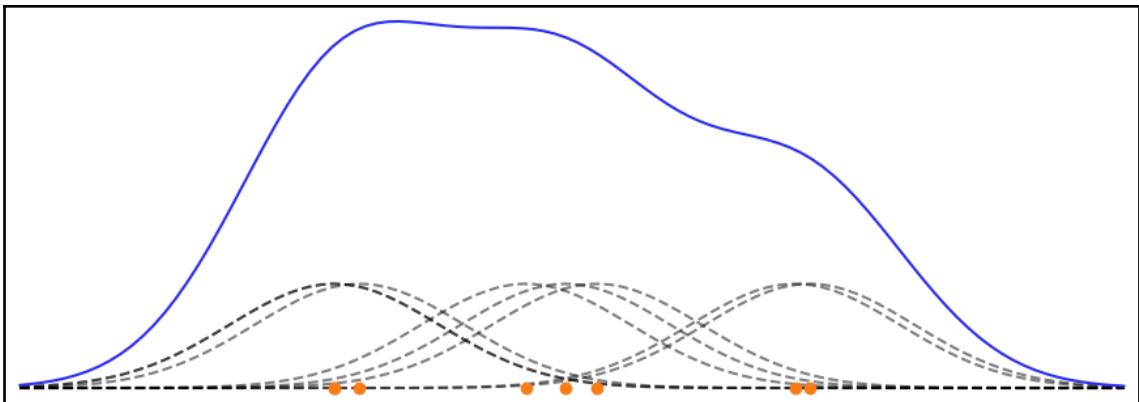


Figure 6.1

Whether we really believe in sub-populations, or we use them for mathematical convenience (or even something in the middle), mixture models are a useful way of adding flexibility to our models by using a mixture of distributions to describe the data.

Finite mixture models

One way to build mixture models is to consider a finite weighted mixture of two or more distributions. This is known as a **finite mixture model**. Thus, the probability density of the observed data is a weighted sum of the probability density for K subgroups of the data:

$$p(y | \theta) = \sum_{i=1}^K w_i p_i(y | \theta_i) \quad (6.1)$$

Here, w_i is the weight of each component (or class). We can interpret w_i as the probability of the component i , thus its values are restricted to the interval $[0, 1]$ and $\sum_i^K w_i = 1$. The components $p_i(y | \theta_i)$ can be virtually anything we may consider useful from simple distributions, such as a Gaussian or a Poisson, to more complex objects, such as hierarchical models or neural networks. For a finite mixture model, K is a finite number (usually, but not necessarily, a small number $K \lesssim 20$). In order to fit a finite mixture model, we need to provide a value of K , either because we really know the *correct* value beforehand, or because we have some educated guess.

Conceptually, to solve a mixture model, *all we need to do* is to properly assign each data point to one of the components. In a probabilistic model, we can do this by introducing a random variable z , whose function is to specify to which component a particular observation is assigned. This variable is generally referred to as a *latent* variable. We call it latent because it is not directly observable.

Let's start building mixture models by using the chemical shifts data we already saw in Chapter 2, *Programming Probabilistically*:

```
cs = pd.read_csv('../data/chemical_shifts_theo_exp.csv')
cs_exp = cs['exp']
az.plot_kde(cs_exp)
plt.hist(cs_exp, density=True, bins=30, alpha=0.3)
plt.yticks([])
```

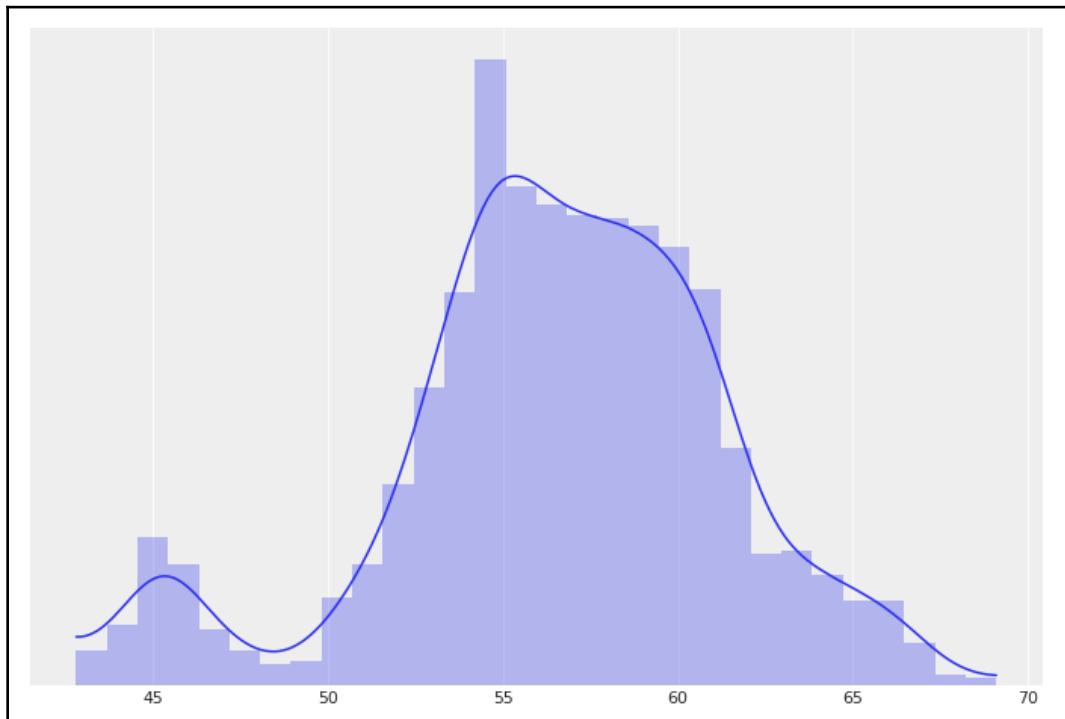


Figure 6.2

From *Figure 6.2*, we can see that this data cannot be properly described using a single Gaussian, but maybe three or four could do the trick. In fact, there are good theoretical reasons, that we will not discuss here, indicating that data really comes from a mixture of around 40 sub-populations, but with considerable overlap between them.

To develop the intuition behind mixture models, we can get ideas from the coin-flipping problem. For that model, we have two possible outcomes and we use the Bernoulli distribution to describe them. Since we did not know the probability of getting heads or tails, we use a beta distribution as a prior. A mixture model is similar, except that instead of two outcomes, like heads or tails, we now have K outcomes (or K -components). The generalization of the Bernoulli distribution to K -outcomes is the categorical distribution and the generalization of the beta distribution is the Dirichlet distribution. So, let me introduce these two new distributions.

The categorical distribution

The categorical distribution is the most general discrete distribution and is parameterized using a parameter specifying the probabilities of each possible outcome. *Figure 6.3* represents two possible instances of the categorical distribution. The dots represent the values of the categorical distribution, while the continuous lines are a visual aid to help us easily grasp the *shape* of the distribution:

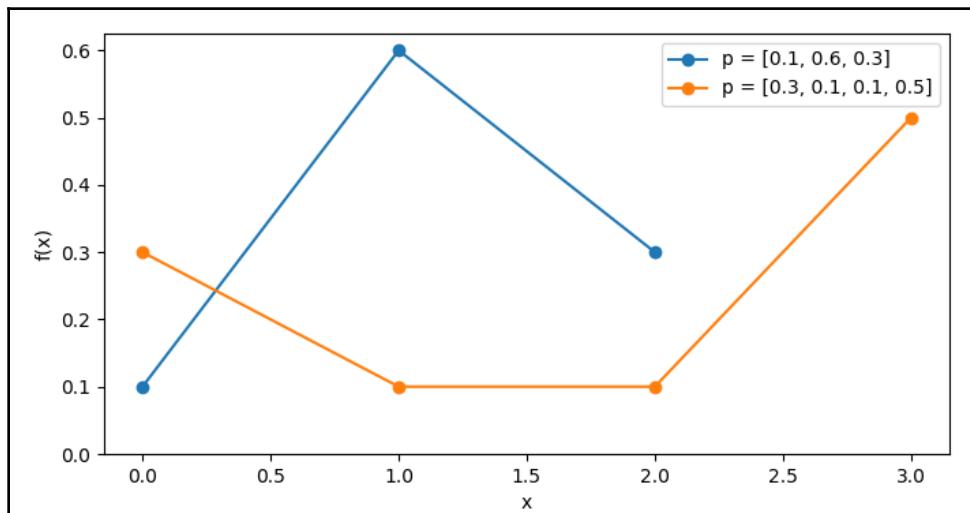


Figure 6.3

The Dirichlet distribution

The Dirichlet distribution lives in the simplex, which is like an n -dimensional triangle; a 1-simplex is a line, a 2-simplex is a triangle, a 3-simplex a tetrahedron, and so on. Why a simplex? Intuitively, because the output of this distribution is a K -length vector, whose elements are restricted to be zero or larger than zero and sum up to 1. As we said, the Dirichlet distribution is the generalization of the beta distribution. Thus, a good way to understand the former is to compare it to the latter. We use the beta for two outcome problems: one with probability p and the other $1 - p$. As we can, see $p + 1 - p = 1$. The beta returns a two-element vector $(p, 1 - p)$, but in practice, we omit $1 - p$, as the outcome is entirely determined once we know p . If we want to extend the beta distribution to three outcomes, we need a three-element vector (p, q, r) , where each element is larger than zero and $p + q + r = 1$ and thus $r = 1 - (p + q)$. We could use three scalars to parameterize such distribution and we may call them α , β , and γ , however, we could easily run out of Greek letters, as there are only 24 of them; instead, we can just use a vector named α with length K , where K is the number of outcomes. Note that we can think of the beta and Dirichlet as distributions over proportions. To get an idea of this distribution, pay attention to *Figure 6.4* and try to relate each triangular subplot to a beta distribution with similar parameters:

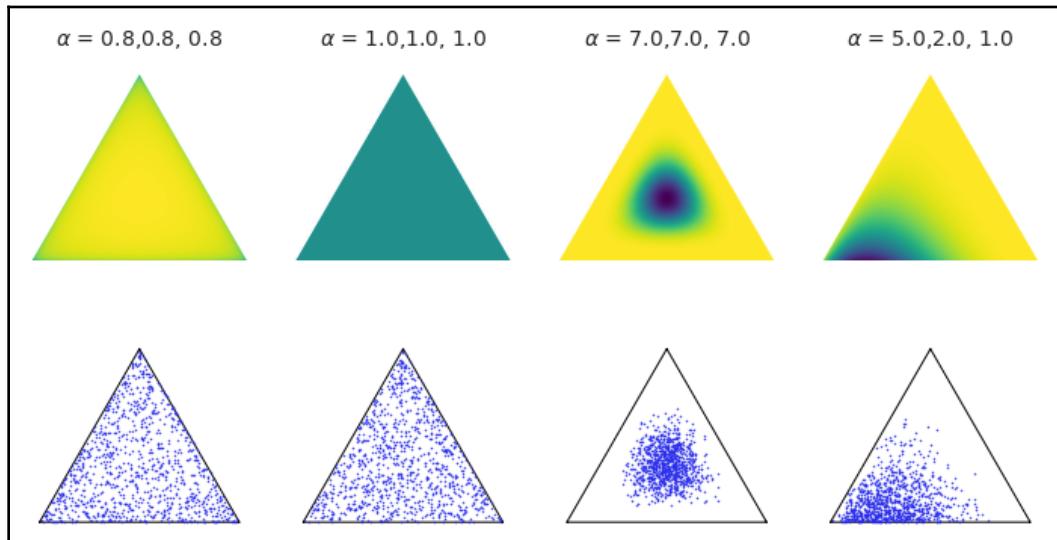


Figure 6.4

Now that we have a better grasp of the Dirichlet distribution, we have all the elements to build mixture models. One way to visualize them is as a K -side coin flip model *on top of* a Gaussian estimation model. Using Kruschke-style diagrams:

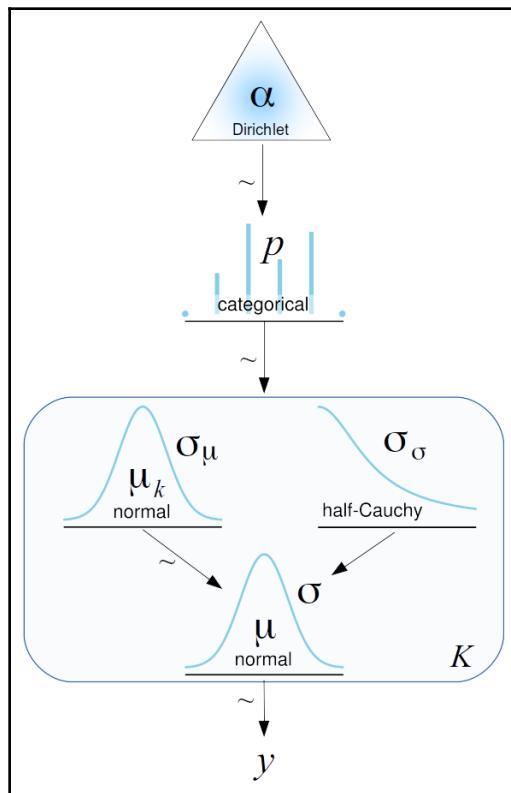


Figure 6.5

The rounded-corner box is indicating that we have K -components and the categorical variables decide which of them we use to describe a given data point.



Notice that in *Figure 6.5*, only μ_k depends on the different components, while σ_μ and σ_σ are shared for all of them. If necessary, we can change this and allow other parameters to be conditioned on each component.

This model (assuming `clusters = 2`) can be implemented using PyMC3, as follows:

```
with pm.Model() as model_kg:
    p = pm.Dirichlet('p', a=np.ones(clusters))
    z = pm.Categorical('z', p=p, shape=len(cs_exp))
    means = pm.Normal('means', mu=cs_exp.mean(), sd=10, shape=clusters)
    sd = pm.HalfNormal('sd', sd=10)

    y = pm.Normal('y', mu=means[z], sd=sd, observed=cs_exp)
    trace_kg = pm.sample()
```

If you run this code, you will find that it is very slow and the trace looks very bad (refer Chapter 8, *Inference Engines*, to learn more about diagnostics). The reason for such difficulties is that in `model_kg` we have explicitly included the latent variable z in the model. One problem with this *explicit* approach is that sampling the discrete variable z usually leads to slow mixing and ineffective exploration of the tails of the distribution. One way to solve these sampling problems is by reparametrizing the model.

Note that in a mixture model, the observed variable y is modeled conditionally on the latent variable z . That is, $p(y | z, \theta)$. We may think of the z latent variable as a nuisance variable that we can marginalize and get $p(y | \theta)$. Luckily for us, PyMC3 includes a `NormalMixture` distribution that we can use to write a Gaussian mixture model in the following way:

```
clusters = 2
with pm.Model() as model_mg:
    p = pm.Dirichlet('p', a=np.ones(clusters))
    means = pm.Normal('means', mu=cs_exp.mean(), sd=10, shape=clusters)
    sd = pm.HalfNormal('sd', sd=10)
    y = pm.NormalMixture('y', w=p, mu=means, sd=sd, observed=cs_exp)
    trace_mg = pm.sample(random_seed=123)
```

Let's use ArviZ to see how the trace looks like, we will compare this trace with the one obtained with `model_mgp` in the next section:

```
varnames = ['means', 'p']
az.plot_trace(trace_mg, varnames)
```

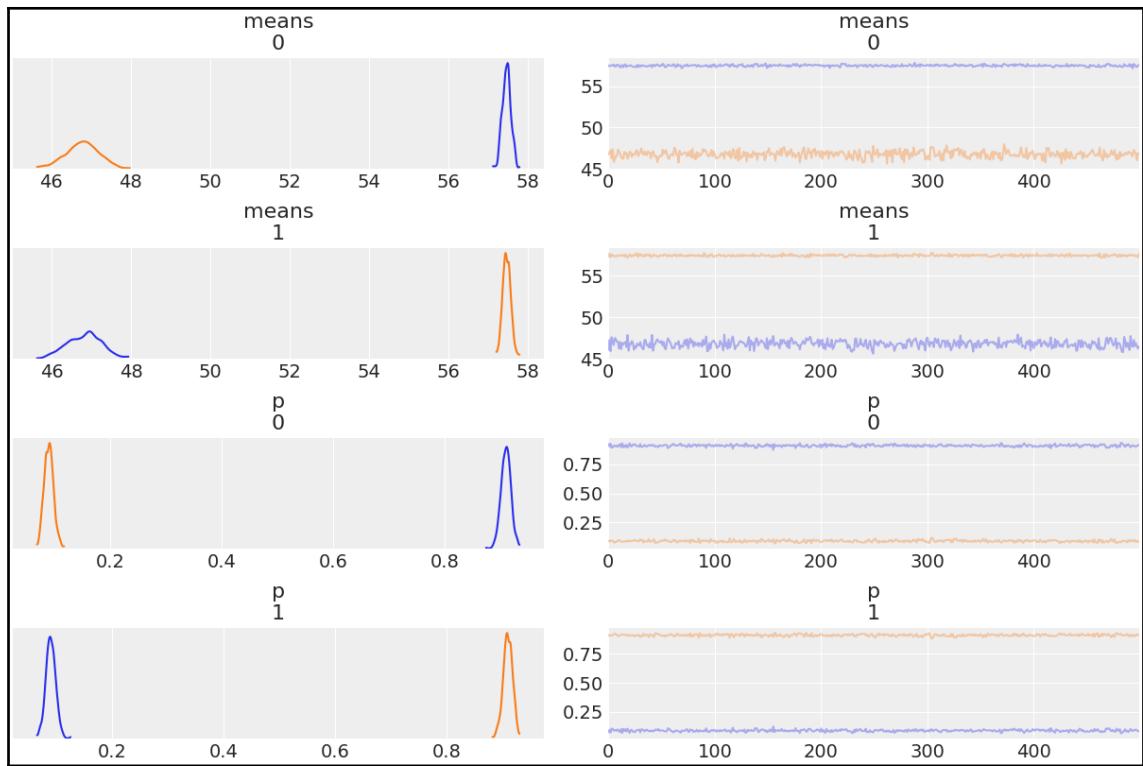


Figure 6.6

Let's also compute the summary for this model, we will compare this summary with the one obtained with `model_mgp` in the next section:

```
az.summary(trace_mg, varnames)
```

	mean	sd	mc error	hpd 3%	hpd 97%	eff_n	r_hat
means[0]	52.12	5.35	2.14	46.24	57.68	1.0	25.19
means[1]	52.14	5.33	2.13	46.23	57.65	1.0	24.52
p[0]	0.50	0.41	0.16	0.08	0.92	1.0	68.91
p[1]	0.50	0.41	0.16	0.08	0.92	1.0	68.91

Non-identifiability of mixture models

If you carefully check *Figure 6.6*, you will find something funny going on. Both means are estimated as bimodal distributions with values around (47, 57.5) and if you check the summary obtained with `az.summary`, the averages of the means are almost equal and around 52. We can see something similar with the values of \mathcal{P} . This is an example of a phenomenon known in statistics as **parameter non-identifiability**. This happen because the model is the same if component 1 has mean 47 and component 2 has a mean 57.5 and vice versa; both scenarios are fully equivalent. In the context of mixture models, this is also known as the **label-switching problem**. We have already found an example of parameter non-identifiability in [Chapter 3, Modeling with Linear Regression](#) when we discussed linear models and variables with high-correlation.

When possible, the model should be defined to remove non-identifiability. With mixture models, there are at least two ways of doing this:

- Force the components to be ordered; for example, arrange the means of the components in strictly increasing order
- Use informative priors



Parameters in a model are not identified if the same likelihood function is obtained for more than one choice of the model parameters.

Using PyMC3, one easy way to enforce the components to be ordered is by using `pm.potential()`. A potential is an arbitrary factor we add to the likelihood, without adding a variable to the model. The main difference between a likelihood and a potential is that the potential does not necessarily depend on data, while the likelihood does. We can use a potential to enforce a constraint. For example we can define the potential in such a way that if the constraint is not violated, we add a factor of zero to the likelihood; otherwise, we add a factor of $-\infty$. The net result is that the model considers the parameters (or combination of parameters) violating the constraints to be impossible, while the model is unperturbed about the rest of the values:

```
clusters = 2
with pm.Model() as model_mgp:
    p = pm.Dirichlet('p', a=np.ones(clusters))
    means = pm.Normal('means', mu=np.array([.9, 1]) * cs_exp.mean(),
                      sd=10, shape=clusters)
    sd = pm.HalfNormal('sd', sd=10)
    order_means = pm.Potential('order_means',
                                tt.switch(means[1]-means[0] < 0,
```

```
-np.inf, 0))  
y = pm.NormalMixture('y', w=p, mu=means, sd=sd, observed=cs_exp)  
trace_mgp = pm.sample(1000, random_seed=123)  
  
varnames = ['means', 'p']  
az.plot_trace(trace_mgp, varnames)
```

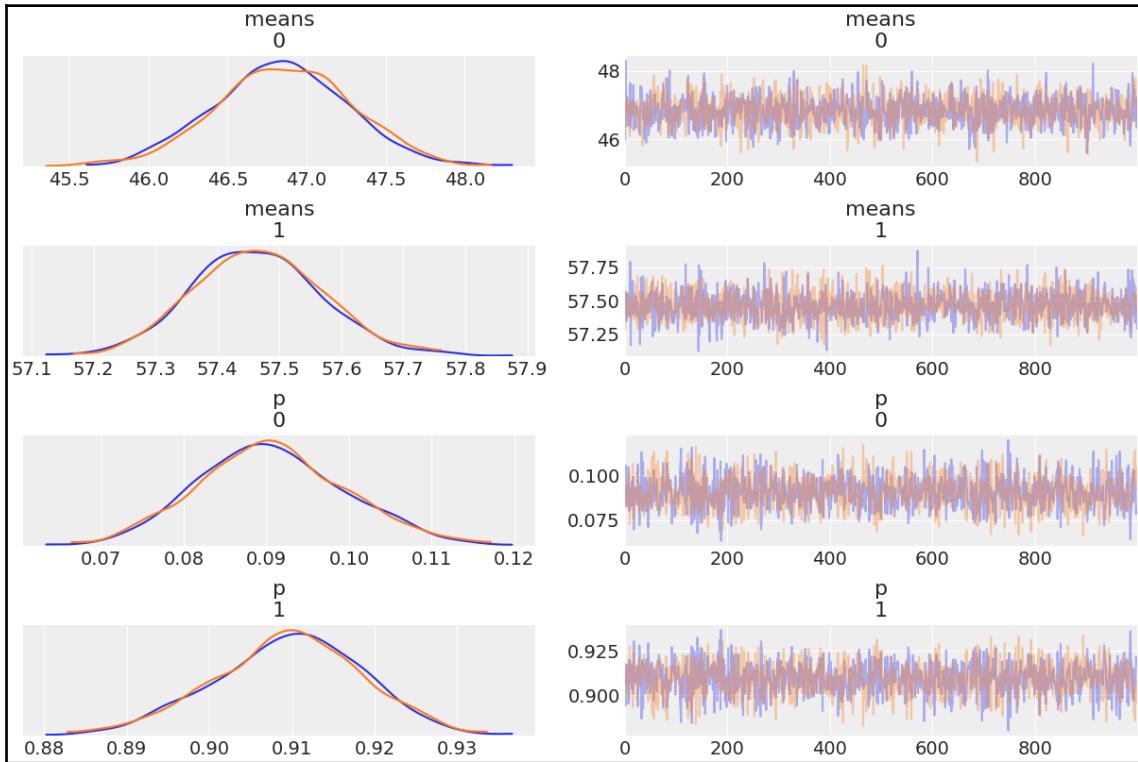


Figure 6.7

Let's also compute the summary for this model:

```
az.summary(trace_mgp)
```

	mean	sd	mc error	hpdi 3%	hpdi 97%	eff_n	r_hat
means[0]	46.84	0.42	0.01	46.04	47.61	1328.0	1.0
means[1]	57.46	0.10	0.00	57.26	57.65	2162.0	1.0
p[0]	0.09	0.01	0.00	0.07	0.11	1365.0	1.0
p[1]	0.91	0.01	0.00	0.89	0.93	1365.0	1.0
sd	3.65	0.07	0.00	3.51	3.78	1959.0	1.0

Another constraint that we may find useful to add is one ensuring all components have a not null probability, or in other words that each components in the mixture get at least one observation attached to it. You can do this with the following expression:

```
p_min = pm.Potential('p_min', tt.switch(tt.min(p) < min_p, -np.inf, 0))
```

Here, you can set `min_p` to some arbitrary, but reasonable value, such as 0.1 or 0.01

As we can see from *Figure 6.4*, the value of α controls the concentration of the Dirichlet distribution. A flat prior distribution on the simplex is obtained with $\alpha = 1$, as used in `model_mgp`. Larger values of α means more informative priors. Empirical evidence suggests that values of $\alpha \approx 4$ or 10 are generally a good default choice as these values usually lead to posteriors distributions with each component having at least one data point assigned to them while reducing the chance of overestimating the number of components.

How to choose K

One of the main concerns with finite mixture models is how to decide the number of components. A rule of thumb is to begin with a relatively small number of components and then increase it in order to improve the model-fit evaluation. As usual, model-fit is evaluated using posterior-predictive checks, measures such as WAIC or LOO, and on the basis of the expertise of the modeler(s).

Let us compare the model for $K = \{3, 4, 5, 6\}$. In order to do this, we are going to fit the model four times, and we are going to save the `trace` and `model` objects for later use:

```
clusters = [3, 4, 5, 6]

models = []
traces = []
for cluster in clusters:
    with pm.Model() as model:
        p = pm.Dirichlet('p', a=np.ones(cluster))
        means = pm.Normal('means',
                           mu=np.linspace(cs_exp.min(), cs_exp.max(),
                           cluster),
                           sd=10, shape=cluster,
                           transform=pm.distributions.transforms.ordered)
        sd = pm.HalfNormal('sd', sd=10)
        y = pm.NormalMixture('y', w=p, mu=means, sd=sd, observed=cs_exp)
        trace = pm.sample(1000, tune=2000, random_seed=123)
        traces.append(trace)
        models.append(model)
```

To better display how K affects the inference, we are going to compare the fit of these models with the one obtained with `az.plot_kde`. We are also going to plot the Gaussian components of the mixture model:

```
_, ax = plt.subplots(2, 2, figsize=(11, 8), constrained_layout=True)

ax = np.ravel(ax)
x = np.linspace(cs_exp.min(), cs_exp.max(), 200)
for idx, trace_x in enumerate(traces):
    x_ = np.array([x] * clusters[idx]).T

    for i in range(50):
        i_ = np.random.randint(0, len(trace_x))
        means_y = trace_x['means'][i_]
        p_y = trace_x['p'][i_]
        sd = trace_x['sd'][i_]
        dist = stats.norm(means_y, sd)
        ax[idx].plot(x, np.sum(dist.pdf(x_) * p_y, 1), 'C0', alpha=0.1)

means_y = trace_x['means'].mean(0)
p_y = trace_x['p'].mean(0)
sd = trace_x['sd'].mean()
dist = stats.norm(means_y, sd)
ax[idx].plot(x, np.sum(dist.pdf(x_) * p_y, 1), 'C0', lw=2)
ax[idx].plot(x, dist.pdf(x_) * p_y, 'k--', alpha=0.7)
az.plot_kde(cs_exp, plot_kwarg={'linewidth':2, 'color':'k'},
```

```
ax=ax[idx]
ax[idx].set_title('K = {}'.format(clusters[idx]))
ax[idx].set_yticks([])
ax[idx].set_xlabel('x')
```

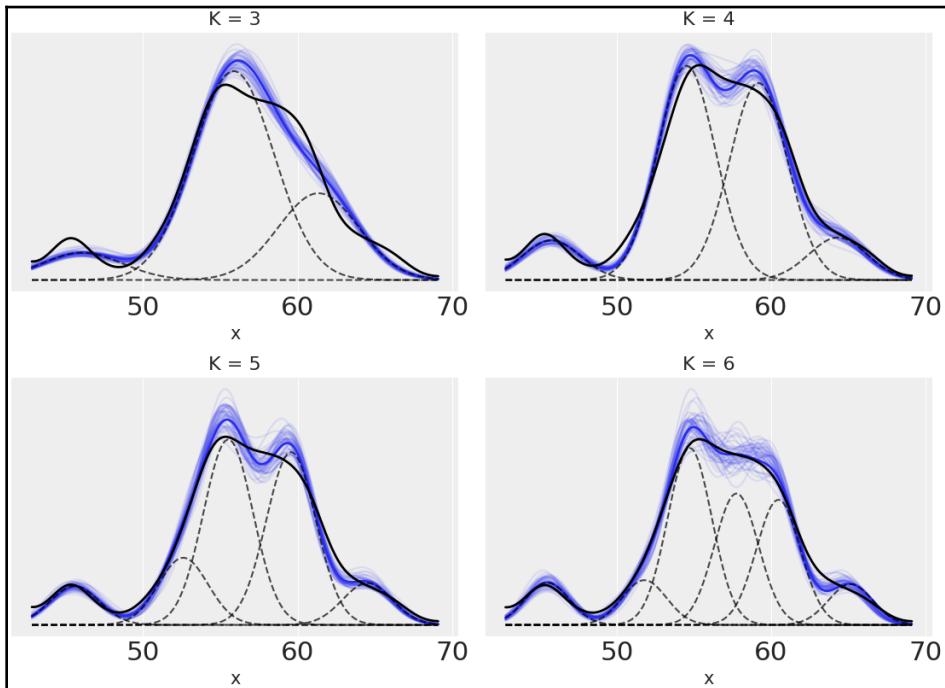


Figure 6.8

The *Figure 6.8* shows a KDE plot of the data, black solid line together with the mean fit wider (blue) line and samples from the posterior semitransparent (blue) lines. Also the mean-Gaussian components are represented using a dashed black line. In the Figure 6.8, it seems that $K = 3$ is too low, and 4, 5, or 6 could be a better choice.

Notice that the Gaussian mixture model shows two central peaks/bumps (more or less around 55-60) while the KDE predicts less marked (more flattened) peaks. Notice this is not necessarily a bad fit of the Gaussian mixture model, since KDEs are generally tuned to provide smoother densities. Instead of a KDE, you could use a histogram but histograms are also methods to approximate densities. As we have already discussed in Chapter 5, *Model Comparison* you could try to compute predictive posterior plots of test quantities of interest and compute Bayesian *p*-values. *Figure 6.9* shows an example of such a calculation and visualization:

```
ppc_mm = [pm.sample_posterior_predictive(traces[i], 1000, models[i])
          for i in range(4)]

fig, ax = plt.subplots(2, 2, figsize=(10, 6), sharex=True,
                      constrained_layout=True)
ax = np.ravel(ax)
def iqr(x, a=0):
    return np.subtract(*np.percentile(x, [75, 25], axis=a))

T_obs = iqr(cs_exp)
for idx, d_sim in enumerate(ppc_mm):
    T_sim = iqr(d_sim['y'][:100].T, 1)
    p_value = np.mean(T_sim >= T_obs)
    az.plot_kde(T_sim, ax=ax[idx])
    ax[idx].axvline(T_obs, 0, 1, color='k', ls='--')
    ax[idx].set_title(f'K = {clusters[idx]}\n p-value {p_value:.2f}')
    ax[idx].set_yticks([])
```

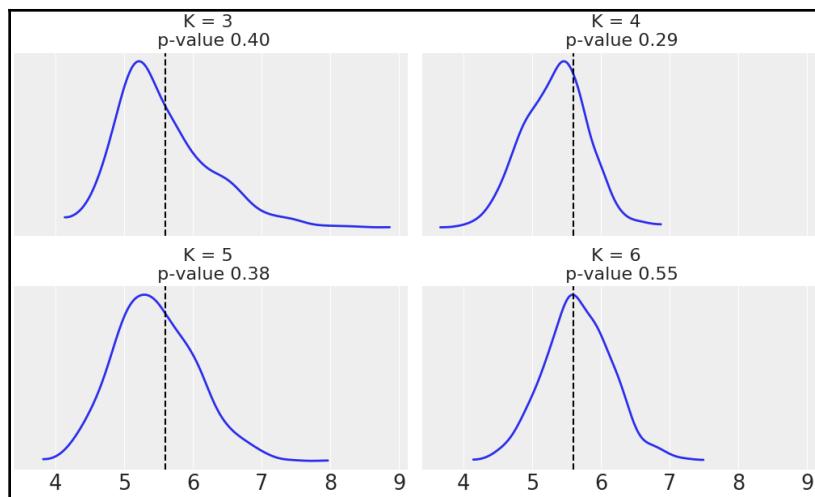


Figure 6.9

From *Figure 6.9*, we can see that the number $K = 6$ is a good choice with a Bayesian p -value very close to 0.5. As we can see in the following DataFrame and in *Figure 6.10*, WAIC also spots $K = 6$ as the better model (among the evaluated ones):

```
comp = az.compare(dict(zip(clusters, traces)), method='BB-pseudo-BMA')
comp
```

	waic	pwaic	dwaic	weight	se	dse	warning
6	10250	12.368	0	0.948361	62.7354	0	0
5	10259.7	10.3531	9.69981	0.0472388	61.3804	4.6348	0
4	10278.9	7.45718	28.938	0.00440011	60.7985	9.82746	0
3	10356.9	5.90559	106.926	3.19235e-13	60.9242	18.5501	0

Most often than not reading a plot is way easier than reading a table, so let's make a plot to spot how different models are according to WAIC. As we can see from *Figure 6.10*, while the model with six components has a lower WAIC than the rest but there is considerable overlap when we consider the estimated standard error (`se`), especially with regard to the model with five components:

```
az.plot_compare(comp)
```

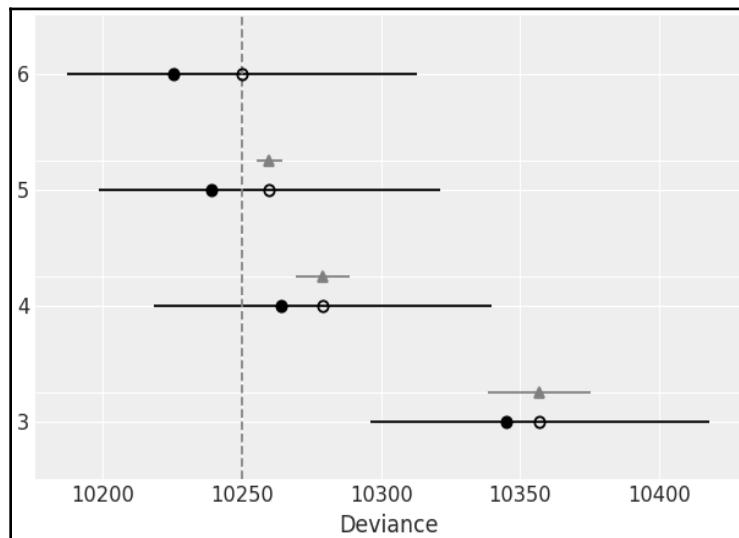


Figure 6.10

Mixture models and clustering

Clustering is part of the unsupervised family of statistical or machine learning tasks and is similar to classification, but a little bit more difficult since we do not know the correct labels!

Clustering or cluster analysis is the data analysis task of grouping objects in such a way that objects in a given group are closer to each other than to those in the other groups. The groups are called **clusters** and the degree of closeness can be computed in many different ways; for example, by using metrics, such as the Euclidean distance. If instead we take the probabilistic route, then a mixture model arises as a *natural* candidate to solve clustering tasks.

Performing clustering using probabilistic models is usually known as **model-based clustering**. Using a probabilistic model allows us to compute the probability of each data point belonging to each one of the clusters. This is known as **soft-clustering** as opposed to hard-clustering, where each data point belongs to a cluster with a probability of 0 or 1. We can turn soft-clustering into hard-clustering by introducing some rule or boundary, in fact you may remember that this is exactly what we do to turn logistic regression into a classification method, where we use as the default boundary the value of 0.5. For clustering, a reasonable choice is to assign a data point to the cluster with the highest probability.

In summary, when people talk about clustering, they are generally taking about grouping objects and when people talk about mixture models, they talk about using a mix of simple distributions to model a more complex distribution, either to identify subgroups or just to have a more flexible model to describe the data.

Non-finite mixture model

For some problems, such as trying to clusterize handwritten digits, it is easy to justify the number of groups we expect to find in the data. For other problems we can have good guesses; for example, we may know that our sample of Iris flowers was taken from a region where only three species of Iris grow, thus using three components is a reasonable starting point. When we are not that sure about the number of components we can use model selection to help us choose the number of groups. Nevertheless for other problems, choosing *a priori* the number of groups can be a shortcoming and we instead are interested in estimating this number from the data. A Bayesian solution for this type of problem is related to the **Dirichlet process**.

Dirichlet process

All models that we have seen so far were parametric models. These are models with a fixed number of parameters that we are interested in estimating, like a fixed number of clusters. We can also have non-parametric models, probably a better name for these models will be non-fixed-parametric models, but someone already decided the name for us. Non-parametric models are models with a theoretically infinite number of parameters. In practice, we somehow *let the data to reduce* the theoretically infinite number of parameters to some finite number, in other words the data *decides* the actual number of parameters, thus non-parametric models are very flexible. In this book we are going to see two examples of such models: the Gaussian process (this is the subject of the next chapter) and the Dirichlet process, which we will start discussing in the next paragraph.

As the Dirichlet distribution is the n-dimensional generalization of the beta distribution the **Dirichlet process (DP)** is the infinite-dimensional generalization of the Dirichlet distribution. The Dirichlet distribution is a probability distribution on the space of probabilities, while the DP is a probability distribution on the space of distributions, this means that a single draw from a DP is actually a distribution. For finite mixture models we used the Dirichlet distribution to assign a prior for the fixed number of clusters or groups. A DP is a way to assign a prior distribution to a non-fixed number of clusters, even more we can think of a DP as a way to sample from a prior distribution of distributions.

Before we move to the actual non-parametric mixture model let us take a moment to discuss a little bit some of the details of the DP. The formal definition of a DP is somehow obscure, unless you know your probability theory very well, so instead let me describe some of the properties of a DP that are relevant to understand its role in modeling mixture models:

- A DP is a distribution whose realizations are probability distributions, instead of say real numbers like for a Gaussian distribution.
- A DP is specified by a base distribution \mathcal{H} and a positive real number α called the **concentration parameter** (this is analogous to the concentration parameter in the Dirichlet distribution).
- \mathcal{H} is the expected value of the DP, this means that a DP will generate distributions around the base distribution, this is somehow equivalent to the mean of a Gaussian distribution.
- As α increases the realizations become less and less concentrated.
- In practice a DP always generates discrete distributions.

- In the limit $\alpha \rightarrow \infty$ the realizations from a DP will be equal to the base distribution, thus if the base distribution is continuous the DP will generate a continuous distribution. For this reason mathematicians say that the distributions generated from a DP are *almost surely* discrete. In practice, as α will be a finite number we always will work with discrete distributions.

To make these properties more concrete let us take a look again at the categorical distribution in *Figure 6.3*. We can completely specify such distribution by indicating the position on the x axis and the height on the y axis. For the categorical distribution the positions on the x axis are restricted to be integers and the sum of the heights has to be 1. Let keep the last restriction but relax the former one. To generate the positions on the x axis we are going to sample from a base distribution H . In principle H can be any distribution we want, thus if we choose a Gaussian the locations could be in principle any value from the real line, instead if we choose a beta, the locations will be restricted to the interval $[0, 1]$ and if we choose a Poisson as the base distribution the locations will be restricted to the non-negative integers $\{0, 1, 2, \dots\}$.

So far so good, how we choose the values on the y axis? We follow a *Gedankenexperiment* known as the **stick-breaking process**. Imagine we have a stick of length 1, then we break it in two parts (not necessarily equal). We set one part aside and we break the other part into two, and then we just keep doing this for ever and ever. In practice, as we cannot really repeat the process infinitely we truncate it at some predefined K value, but the general idea holds. To control the stick-breaking process we use a parameter α . As we increase the value of α we will break the stick in smaller and smaller portions. Thus in the $\lim_{\alpha \rightarrow 0}$ we will not break the stick and when $\lim_{\alpha \rightarrow \infty}$ we will break it into infinite pieces. *Figure 6.11* shows four draws from a DP, for four different values of α . I will explain the code that generate that figure in a moment, let focus first on understanding what these samples tell us about a DP:

```
def stick_breaking_truncated(alpha, H, K):
    """
    Truncated stick-breaking process view of a DP
    Parameters
    -----
    alpha : float
        concentration parameter
    H : scipy distribution
        Base distribution
    K : int
        number of components
    Returns
    -----
    locs : array
        locations
    w : array
```

```

    probabilities
"""
βs = stats.beta.rvs(1, α, size=K)
w = np.empty(K)
w = βs * np.concatenate(([1.], np.cumprod(1 - βs[:-1])))
locs = H.rvs(size=K)
return locs, w

# Parameters DP
K = 500
H = stats.norm
alphas = [1, 10, 100, 1000]

_, ax = plt.subplots(2, 2, sharex=True, figsize=(10, 5))
ax = np.ravel(ax)
for idx, α in enumerate(alphas):
    locs, w = stick_breaking_truncated(α, H, K)
    ax[idx].vlines(locs, 0, w, color='C0')
    ax[idx].set_title('α = {}'.format(α))

plt.tight_layout()

```

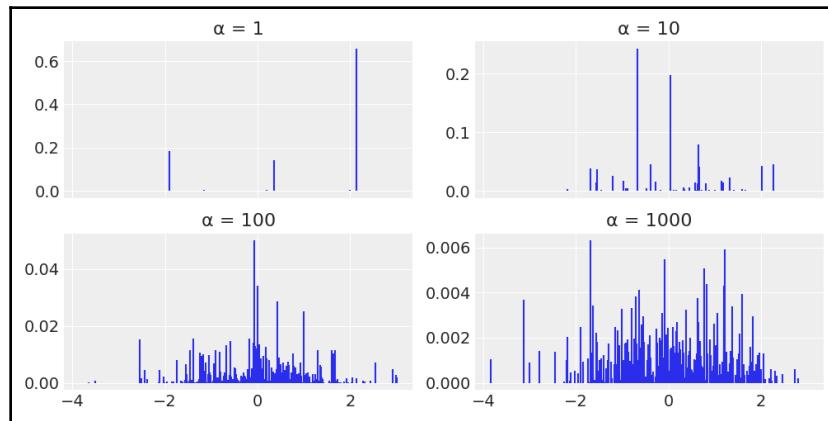


Figure 6.11

We can see from *Figure 6.10* that the DP is a discrete distribution. When α increases we obtain a more spread distribution and smaller pieces of the stick, notice the change on the scale of the y axis and remember that the total length is fixed at 1. The base distribution controls the locations, as the locations are sampled from the base distribution, we can see from *Figure 6.10* that as α increases the shape of the DP distributions resembles more and more the base distribution H , from this we can hopefully see that in the $\lim_{\alpha \rightarrow \infty}$ we should exactly obtain the base distribution.



We can think of a DP as the prior on a random distribution f , where the base distribution μ is what we expected f to be and the concentration parameter α represents how confident we are about our prior guess.

Figure 6.1 shows that if you place a Gaussian on top of each data point and then sum all the Gaussians you can approximate the distribution of the data. We can use a DP to do something similar, but instead of placing a Gaussian on top of each data point we can place a Gaussian at the location of each *substick* from a DP realization and we scale or weight that Gaussian by the length of that *substick*. This procedure provides a general recipe for an infinite-Gaussian-mixture-model. Alternatively, we can replace the Gaussian for any other distribution and we will have a general recipe for a general infinite-mixture-model. *Figure 6.12*, show an example of such model, were we use a mixture of Laplace distributions. I choose a Laplace distribution arbitrarily just to reinforce the idea that you are by no-means restricted to do Gaussian-mixture-models:

```
a = 10
H = stats.norm
K = 5

x = np.linspace(-4, 4, 250)
x_ = np.array([x] * K).T
locs, w = stick_breaking_truncated(a, H, K)

dist = stats.laplace(locs, 0.5)
plt.plot(x, np.sum(dist.pdf(x_) * w, 1), 'C0', lw=2)
plt.plot(x, dist.pdf(x_) * w, 'k--', alpha=0.7)
plt.yticks([])
```

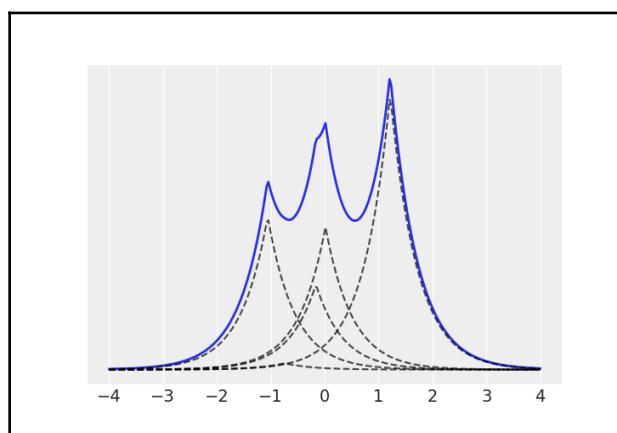


Figure 6.12

I hope at this point you have a good intuition of the DP, the only detail still missing is to understand the function `stick_break_truncated`. Mathematically the stick breaking process view of the DP can be represented in the following way:

$$\sum_{k=1}^{\infty} w_k \cdot \delta_{\theta_k}(\theta) = f(\theta) \sim DP(\alpha, H) \quad (6.2)$$

Where:

- δ_{θ_k} is the indicator function which evaluates to zero everywhere, except for $\delta_{\theta_k}(\theta_k) = 1$, this represent the locations sampled from the base distribution \mathcal{H}
- The probabilities w_k are given by:

$$w_k = \beta'_k \cdot \prod_{i=1}^{k-1} (1 - \beta'_i) \quad (6.3)$$

Where:

- w_k is the length of a substick
- $\prod_{i=1}^{k-1} (1 - \beta'_i)$ is the length of the remaining portion, the one we need to keep breaking
- β'_k indicates how to break the remaining portion
- $\beta'_k \sim \text{Beta}(1, \alpha)$, from this expression we can see that when α increases β'_k will be on average smaller

Now we are more than ready to try to implement a DP in PyMC3. Let first define a `stick_breaking` function that works with PyMC3:

```
N = cs_exp.shape[0]
K = 20

def stick_breaking(alpha):
    beta = pm.Beta('beta', 1., alpha, shape=K)
    w = beta * pm.math.concatenate([[1.],
                                    tt.extra_ops.cumprod(1. - beta)[:-1]])
    return w
```

We have to define a prior for α , the concentration parameter. A common choice for this is a Gamma distribution:

```
with pm.Model() as model:
    alpha = pm.Gamma('alpha', 1., 1.)
    w = pm.Deterministic('w', stick_breaking(alpha))
    means = pm.Normal('means', mu=cs_exp.mean(), sd=10, shape=K)
    sd = pm.HalfNormal('sd', sd=10, shape=K)
    obs = pm.NormalMixture('obs', w, means, sd=sd, observed=cs_exp.values)
    trace = pm.sample(1000, tune=2000, nuts_kwargs={'target_accept': 0.9})
```

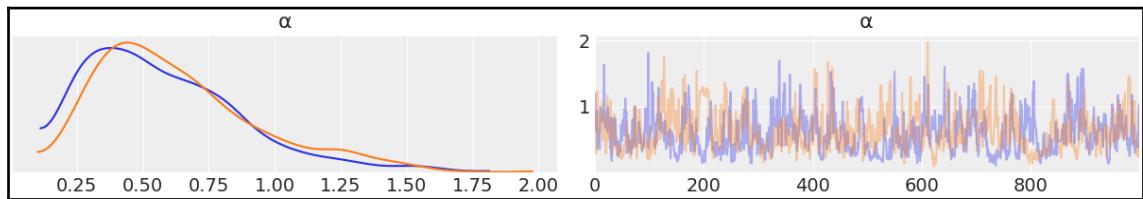


Figure 6.13

From *Figure 6.13* we can see that the value of α is rather low indicating that a few component are necessary to describe the data.

Because we are approximating the infinite DP by a truncated stick-breaking procedure is important to check that the truncation value ($K = 20$ in this example) is not introducing any bias. A simple way to do this is to plot the average weight of each component, to be on the safe side we should have several components with negligible weight, otherwise we must increase the truncation value. An example of this type of plot is *Figure 6.14*. We can see that only a few of the first components are important and thus we can confident that the chosen upper-value of $K = 20$ is large enough for this model and data:

```
plt.figure(figsize=(8, 6))
plot_w = np.arange(K)
plt.plot(plot_w, trace['w'].mean(0), 'o-')
plt.xticks(plot_w, plot_w+1)
plt.xlabel('Component')
plt.ylabel('Average weight')
```

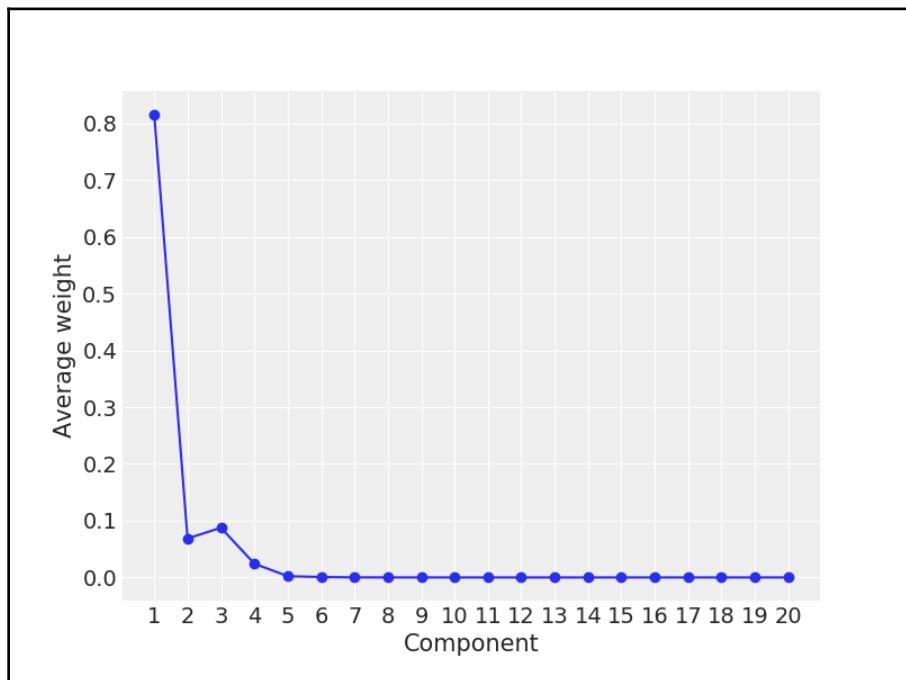


Figure 6.14

Figure 6.15 shows the mean density estimated using the DP model (black) line together with samples from the posterior (grey) lines to reflect the uncertainty in the estimation. This model also shows a less smooth density compared to the KDE from Figure 6.2 and Figure 6.8:

```
x_plot = np.linspace(cs.exp.min()-1, cs.exp.max()+1, 200)

post_pdf_contribs = stats.norm.pdf(np.atleast_3d(x_plot),
                                    trace['means'][:, np.newaxis, :],
                                    trace['sd'][:, np.newaxis, :])
post_pdfs = (trace['w'][:, np.newaxis, :] *
              post_pdf_contribs).sum(axis=-1)

plt.figure(figsize=(8, 6))

plt.hist(cs_exp.values, bins=25, density=True, alpha=0.5)
plt.plot(x_plot, post_pdfs[::-1].T, c='0.5')
plt.plot(x_plot, post_pdfs.mean(axis=0), c='k')

plt.xlabel('x')
plt.yticks([])
```

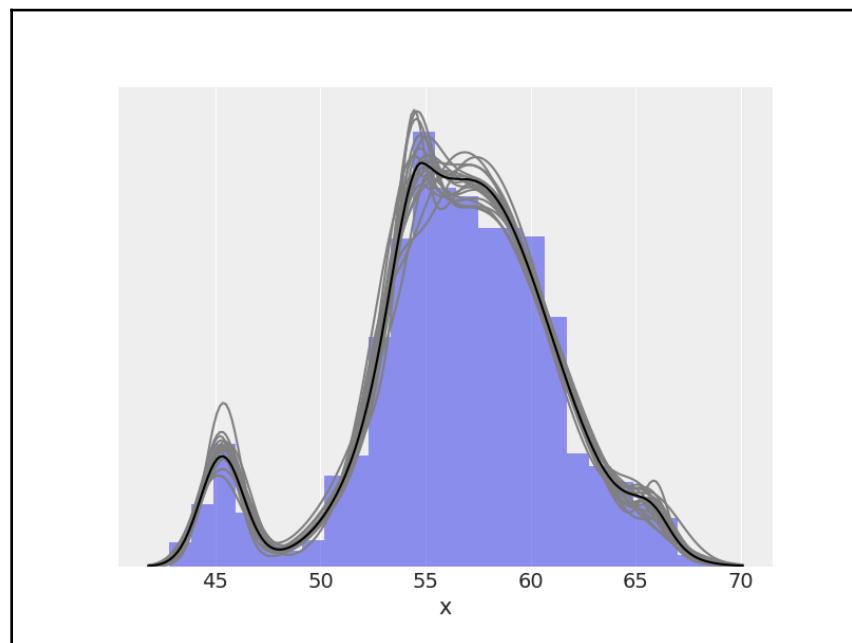


Figure 6.15

Continuous mixtures

This chapter was focused on discrete mixture models but we can also have continuous mixture models. And indeed we already know some of them, like the zero-inflated distribution from Chapter 4, *Generalizing Linear Models*, where we had a mixture of a Poisson distribution and a zero-generating process. Another example was the robust logistic regression model from the same chapter, that model is a mixture of two components: a logistic on one hand and a random guessing on the other. Note that the parameter π is not an on/off switch, but instead is more like a mix-knob controlling how much random guessing and how much logistic regression we have in the mix. Only for extreme values of π do we have a pure random-guessing or pure logistic regression.

Hierarchical models can be also be interpreted as continuous mixture models where the parameters in each group come from a continuous distribution in the upper level. To make it more concrete, think about performing linear regression for several groups. We can assume that each group has its own slope or that all the groups share the same slope. Alternatively, instead of framing our problem as two extreme and discrete options a hierarchical model allow us to effectively model a continuous mixture of these extreme options, thus the extreme options are just particular cases of this larger hierarchical model.

Beta-binomial and negative binomial

The beta-binomial is a discrete distribution generally used to describe the number of success y for n Bernoulli trials when the probability of success p at each trial is unknown and assumed to follow a beta distribution with parameters α and β :

$$\text{BetaBinomial}(y \mid n, \alpha, \beta) = \int_0^1 \text{Bin}(y \mid p, n) \text{Beta}(p \mid \alpha, \beta) dp \quad (6.4)$$

That is, to find the probability of observing the outcome y , we average over all the possible (and continuous) values of p . And hence the beta-binomial can be considered as a continuous mixture model. If the beta-binomial model sounds familiar to you, it is because you have been paying attention to the first two chapters of the book! This is the model we use for the coin-flipping problem, although we explicitly use a beta and a binomial distribution, instead of using the already mixed beta-binomial distribution.

In a similar fashion, we have the negative-binomial distribution, which can be understood as a gamma-Poisson mixture. For this model we have a mixture of Poisson distributions where the rate parameter is gamma distributed. This distribution is often used to circumvent a common problem encountered when dealing with count data. This problem is known as **over-dispersion**. Suppose you are using a Poisson distribution to model count data, and then you realize that the variance in your data exceeds that of the model; the problem with using a Poisson distribution is that mean and variance are linked (in fact they are described by the same parameter). So one way to solve this problem is to model the data as a (continuous) mixture of Poisson distributions with rates coming from a gamma distribution, which gives us the rationale to use the negative-binomial distribution. Since we are now considering a mixture of distributions, our model has more flexibility and can better accommodate the observed mean and variance of the data. Both the beta-binomial and the negative-binomial can be used as part of linear models and both also have zero-inflated versions of them. And also, both are implemented on PyMC3 as ready-to-use distributions.

The Student's t-distribution

We introduce the Student's t-distribution as a robust alternative to the Gaussian distribution. It turns out that the Student's t-distribution can also be thought of as a continuous mixture. In this case we have:

$$t_\nu(y | \mu, \sigma) = \int_0^\infty N(y | \mu, \sigma) \text{Inv}\chi^2(\sigma | \nu) d\nu \quad (6.5)$$

Notice this is similar to the previous expression for the negative-binomial, except here we have a Normal distribution with the parameters μ and σ and the $\text{Inv}\chi^2$ distribution with the parameter ν from which we sample the values of σ is the parameter known as a degree of freedom, or as we prefer to call it, the normality parameter. The parameter ν , as well as p for the beta-binomial, is the equivalent of the z latent variable for finite mixture models. For some finite mixture models, it is also possible to marginalize the distribution respect to the latent variable before doing inference, which may lead to an easier to sample model, as we already saw with the marginalized mixture model example.

Summary

Many problems can be described as an overall populations composed of distinct sub-populations. When we know to which sub-population each observation belongs, we can specifically model each sub-population as a separate group. However, many times we do not have direct access to this information, thus it may be more appropriate to model that data using mixture models. We can use mixture models, to try to capture true sub-populations in the data or as a general statistical trick to model complex distributions by combining simpler distributions. We may even try to do something in the middle.

In this chapter we divide mixture models into three classes—finite mixture models, infinite mixture models, and continuous mixture models. A finite mixture model is a finite weighted mixture of two or more distributions, each distribution or component representing a subgroup of the data. In principle the components can be virtually anything we may consider useful from simple distributions, such as a Gaussian or a Poisson, to more complex objects, such as hierarchical models or neural networks. Conceptually, to solve a mixture model, all we need to do is to properly assign each data point to one of the components, we can do this introducing a latent variable z . We use a categorical distribution for z , which is the most general discrete distribution, with a Dirichlet prior, which is the n-dimensional generalization of the beta distribution. Sampling the discrete variable z can be problematic, thus it may be convenient to marginalize it. PyMC3 includes a normal mixture distribution and a mixture distribution that perform this marginalization for us, making easier to build mixture models with PyMC3. One common problem when working with mixture models is that these models can lead to the label-switching problem, a form of non-identifiability. One way to remove non-identifiability is to force the components to be ordered, with PyMC3 we can achieve this by using `pm.potential()` or an ordered transformation (see the accompanying Jupyter Notebook).

One challenge with finite mixture models is how to decide on the number of components. One solution is to perform model comparison for a set of models around an estimated number of components, that estimation should be guided when possible by our knowledge of the problem at hand. Another option is to try to automatically estimate the number of components from the data. For this reason we introduce the concept of DP, which we use to think in terms of an infinite mixture model. A DP is an infinite-dimensional version of the Dirichlet distribution that we can use to build a non-parametric mixture model.

To close the chapter we briefly discussed how many models such as the beta-binomial (the one used for the coin-flipping problem), the negative-binomial, the Student's t-distribution and even hierarchical models can be interpreted as continuous mixture models.

Exercises

1. Generate synthetic from a mixture of three Gaussians. Check the accompanying Jupyter Notebook for this chapter for an example on how to do this. Fit a finite Gaussian mixture model with 2, 3, or 4 components.
2. Use WAIC and LOO to compare the results from exercise 1.
3. Read and run the following examples about mixture models from the PyMC3 documentation (<https://pymc-devs.github.io/pymc3/examples>):
 - *Marginalized Gaussian Mixture Model* (https://docs.pymc.io/notebooks/marginalized_gaussian_mixture_model.html)
 - *Dependent density regression* (https://docs.pymc.io/notebooks/dependent_density_regression.html)
 - *Gaussian Mixture Model with ADVI* (<https://docs.pymc.io/notebooks/gaussian-mixture-model-advi.html>) (you will find more information about ADVI in Chapter 8, *Inference Engines*)
4. Repeat exercise 1 using a Dirichlet process.
5. Assuming for a moment that you do not know the correct species/labels for the Iris dataset, use a mixture model to cluster the three iris species, using one feature of your choice (like the length of the sepal).
6. Repeat exercise 5 but this time use two features.

7

Gaussian Processes

"Lonely? You have yourself. Your infinite selves."

- Rick Sanchez (*at least the one from dimension C-137*)

In the last chapter, we learned about the Dirichlet process, an infinite-dimensional generalization of the Dirichlet distribution that can be used to set a prior on unknown continuous distributions. In this chapter, we will learn about the Gaussian process, an infinite-dimensional generalization of the Gaussian distribution that can be used to set a prior on unknown functions. Both the Dirichlet process and the Gaussian process are used in Bayesian statistics to build flexible models where the number of parameters is allowed to increase with the size of the data.

In this chapter, we will cover the following topics:

- Functions as probabilistic objects
- Kernels
- Gaussian processes with Gaussian likelihoods
- Gaussian processes with non-Gaussian likelihoods

Linear models and non-linear data

In Chapter 3, *Modeling with Linear Regression*, and Chapter 4, *Generalizing Linear Models*, we learned to build models of the general form:

$$\theta = \psi(\phi(X)\beta) \quad (7.1)$$

Here, θ is a parameter for some probability distribution (for example, the mean of a Gaussian), the p parameter of a binomial, the rate of a Poisson distribution, and so on. We call ψ the inverse link function and ϕ is a function that is the square root or a polynomial function. For the simple linear regression case, ψ is the identity function.

Fitting (or learning) a Bayesian model can be seen as finding the posterior distribution of the *weights* β , and thus, this is known as the weight-view of approximating functions. As we have already seen, with the polynomial regression example, by letting ϕ be a non-linear function, we can map the inputs onto a *feature space*. We then fit a linear relation in the feature space that is not linear in the *actual space*. We saw that by using a polynomial of the proper degree, we can perfectly fit any function. But unless we apply some form of regularization (for example, using prior distributions), this will lead to models that memorize the data or, in other words, models with very poor generalization properties.

Gaussian processes provide a principled solution to modeling arbitrary functions by effectively letting the data decide on the complexity of the function, while avoiding, or at least minimizing, the chance of overfitting.

The following sections explain Gaussian processes from a very practical point of view; we avoid covering almost all the mathematics surrounding them. For a more formal explanation, please check the resources listed in Chapter 9, *Where To Go Next?*.

Modeling functions

We will begin our discussion of Gaussian processes by first describing a way to represent functions as probabilistic objects. We may think of a function, f , as a mapping from a set of inputs, x , to a set of outputs, y . Thus, we can write:

$$y = f(x) \quad (7.2)$$

One way to represent functions is by listing for each x_i value its corresponding y_i value. In fact, you may remember this way of representing functions from elementary school:

x	y
0.00	0.46
0.33	2.60
0.67	5.90
1.00	7.91

As a general case, the values of x and y will live on the real line; thus, we can see a function as a (potentially) infinite and ordered list of paired (x_i, y_i) values. The order is important because, if we shuffle the values, we will get different functions.

A function can also be represented as a (potentially) infinite array indexed by the values of x , with the important distinction that the values of x are not restricted to be integers, but they can take on real numbers.

Using these descriptions, we can represent, numerically, any specific function we want. But what if we want to represent functions probabilistically? Well, we can do so by letting the mapping be of a probabilistic nature. Let me explain this; we can let each y_i value be a random variable distributed as a Gaussian with a given mean and variance. In this way, we no longer have the description of a single specific function, but the description of a family of distributions.

To make this discussion concrete, let's use some Python code to build and plot two examples of such functions:

```
np.random.seed(42)
x = np.linspace(0, 1, 10)

y = np.random.normal(0, 1, len(x))
plt.plot(x, y, 'o-', label='the first one')

y = np.zeros_like(x)
for i in range(len(x)):
    y[i] = np.random.normal(y[i-1], 1)
plt.plot(x, y, 'o-', label='the second one')

plt.legend()
```

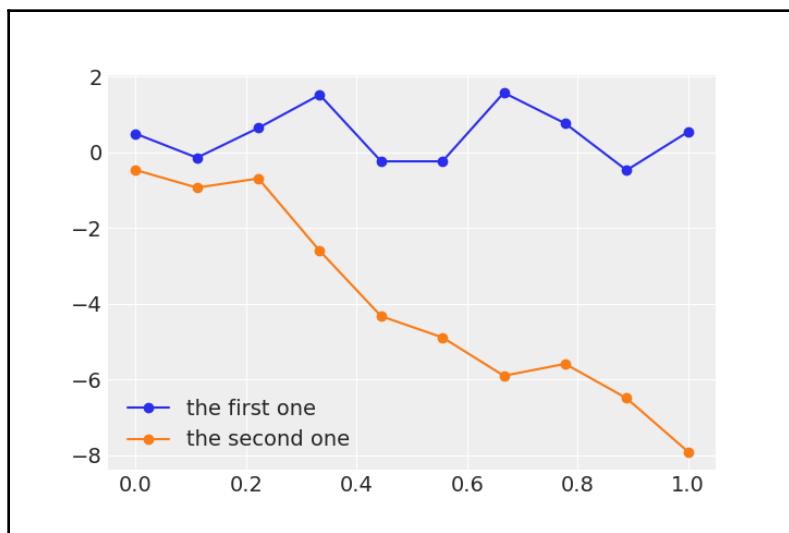


Figure 7.1

Figure 7.1 shows that encoding functions using samples from Gaussian distributions is not that crazy or foolish, so we may be on the right track. Nevertheless, the approach used to generate Figure 7.1 is limited and not sufficiently flexible. While we expect real functions to have some structure or pattern, the way we express **the first** function does not let us encode any relation between data points. In fact, each point is completely independent of the others, as we just get them as 10 independent samples from a common one-dimensional Gaussian distribution. For **the second** function, we introduce some dependency. The mean of the point y_{i+1} is the value y_i . Nevertheless, we will see next that there is a more general approach to capture dependencies (and not only between consecutive points).

Before continuing, let me stop for a moment and ask you why are we using Gaussians and not any other probability distribution? First, because by restricting us to work with Gaussians, we do not lose any flexibility to specify functions of different *shapes*, as each point has potentially its own mean and variance and, second, because working with Gaussians is nice from a mathematical point of view.

Multivariate Gaussians and functions

In *Figure 7.1*, we represented a function using a 1-dimensional Gaussian to get n samples. One alternative is to use an n -dimensional multivariate Gaussian distribution to get a sample vector of length n . In fact, you may want to try to generate a figure such as *Figure 7.1* by replacing `np.random.normal(0, 1, len(x))` with `np.random.multivariate_normal(np.zeros_like(x), np.eye(len(x)))`.

You will see that the first statement is equivalent to the second, but now we can use the covariance matrix as a way to encode information about how data points are related to each other. By allowing the covariance matrix to be `np.eye(len(x))`, we are basically saying that each of the 10 points has a variance of 1 and the variance between them (that is, their covariances) is 0 (thus, they are independent). If we replace those zeros with other (positive) numbers, we could get covariances telling a different story. Thus, to represent functions probabilistically, we just need a multivariate Gaussian with a *suitable* covariance matrix, as we will see in the next section.

Covariance functions and kernels

In practice, covariance matrices are specified using functions known as **kernels**. You may find more than one definition of kernel in the statistical literature, with slightly different mathematical properties. For the purpose of our discussion, we are going to say that a kernel is basically a symmetric function that takes two inputs and returns a value of zero if the inputs are the same or positive otherwise. If these conditions are met, we can interpret the output of a kernel function as a measure of similarity between the two inputs.

Among the many useful kernels available, a popular one used is the exponentiated quadratic kernel:

$$K(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\ell^2}\right) \quad (7.3)$$

Here, $\|x - x'\|^2$ is the squared Euclidean distance:

$$\|x - x'\|^2 = (x_1 - x'_1)^2 + (x_2 - x'_2)^2 + \cdots + (x_n - x'_n)^2 \quad (7.4)$$

It may not be obvious at first sight, but the exponentiated quadratic kernel has a similar formula as the Gaussian distribution (see expression 1.3). For this reason, you may find people referring to this kernel as the Gaussian kernel. The term ℓ is known as the length-scale (or bandwidth or variance) and controls the width of the kernel.

To better understand the role of kernels, let's define a Python function to compute the exponentiated quadratic kernel:

```
def exp_quad_kernel(x, knots, ℓ=1):
    """exponentiated quadratic kernel"""
    return np.array([np.exp(-(x-k)**2 / (2*ℓ**2)) for k in knots])
```

The following code and *Figure 7.2* are designed to show how a 4×4 covariance matrix looks for different inputs. The input I choose is rather simple and consists of the values $[-1, 0, 1, 2]$. Once you understand this example, you should try it with other inputs (see exercise 1):

```
data = np.array([-1, 0, 1, 2])
cov = exp_quad_kernel(data, data, 1)

_, ax = plt.subplots(1, 2, figsize=(12, 5))
ax = np.ravel(ax)

ax[0].plot(data, np.zeros_like(data), 'ko')
ax[0].set_yticks([])
for idx, i in enumerate(data):
    ax[0].text(i, 0+0.005, idx)
ax[0].set_xticks(data)
ax[0].set_xticklabels(np.round(data, 2))
#ax[0].set_xticklabels(np.round(data, 2), rotation=70)

ax[1].grid(False)
im = ax[1].imshow(cov)
colors = ['w', 'k']
for i in range(len(cov)):
    for j in range(len(cov)):
        ax[1].text(j, i, round(cov[i, j], 2),
                   color=colors[int(im.norm(cov[i, j]) > 0.5)],
                   ha='center', va='center', fontdict={'size': 16})
ax[1].set_xticks(range(len(data)))
ax[1].set_yticks(range(len(data)))
ax[1].xaxis.tick_top()
```

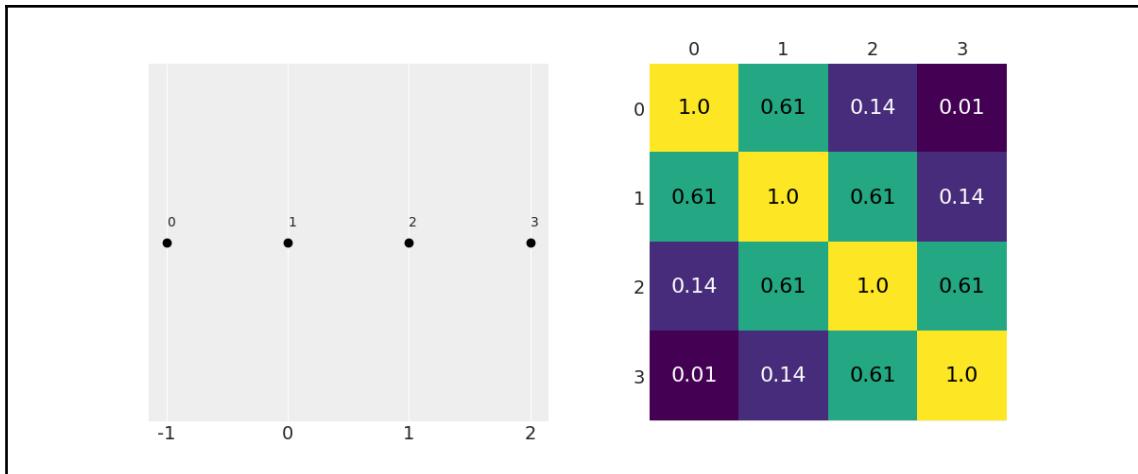


Figure 7.2

The panel on the left of *Figure 7.2* shows the input, the values on the x axis represent the values of each data point, and the text annotations show the order of the data points (starting from zero). On the right-hand panel, we have a heatmap representing the covariance matrix obtained using the exponentiated quadratic kernel. The lighter color means a larger covariance. As you can see, the heatmap is symmetric, with the diagonal taking the larger values. The value of each element in the covariance matrix is inversely proportional to the distance between the points, as the diagonal is the result of comparing each data point with itself. We get the closest distance, 0, and higher covariance values, 1, for this kernel. Other values are possible for other kernels.



The kernel is translating the distance of the data points along the x axis to values of covariances for values of the expected function (on the y axis). Thus, the closer two points are on the x axis, the more similar we expect their values to be on the y axis.

In summary, we have seen so far that we can use multivariate normal distributions with a given covariance to model functions. And we can use kernel functions to build the covariances. In the following example, we use the `exp_quad_kernel` function to define the covariance matrix of a multivariate normal, and then use samples from that distribution to represent functions:

```
np.random.seed(24)
test_points = np.linspace(0, 10, 200)
fig, ax = plt.subplots(2, 2, figsize=(12, 6), sharex=True,
                      sharey=True, constrained_layout=True)
ax = np.ravel(ax)
```

```

for idx, ℓ in enumerate((0.2, 1, 2, 10)):
    cov = exp_quad_kernel(test_points, test_points, ℓ)
    ax[idx].plot(test_points, stats.multivariate_normal.rvs(cov=cov,
size=2).T)
    ax[idx].set_title(f'ℓ = {ℓ}')
fig.text(0.51, -0.03, 'x', fontsize=16)
fig.text(-0.03, 0.5, 'f(x)', fontsize=16)

```

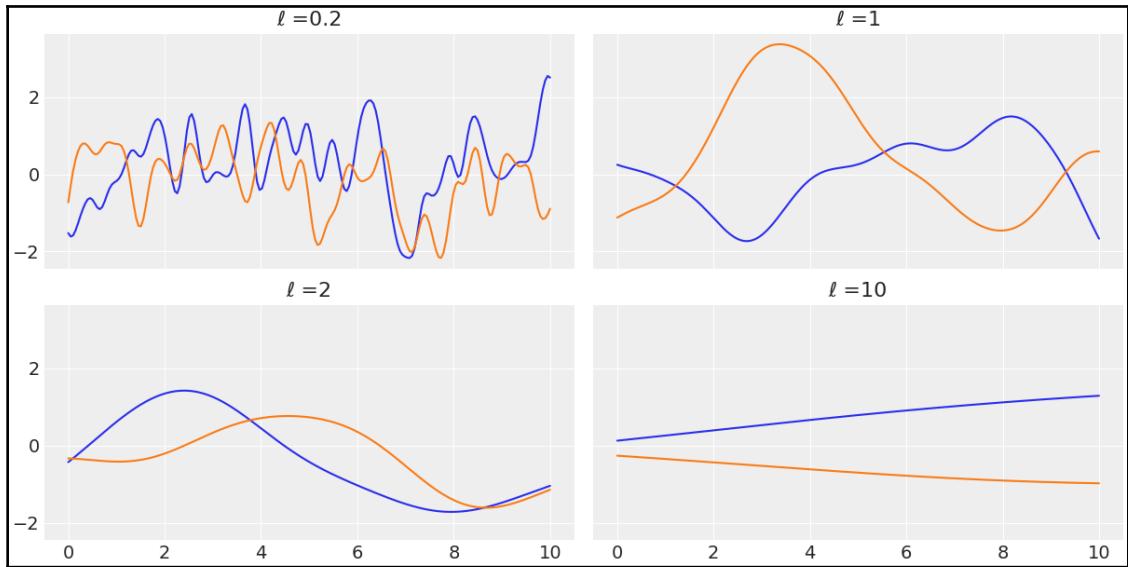


Figure 7.3

As you can see in *Figure 7.3*, a Gaussian kernel implies a wide variety of functions with the parameter ℓ controlling the *smoothness* of the functions. The larger the value of ℓ , the smoother the function.

Gaussian processes

Now we are ready to understand what **Gaussian processes (GPs)** are and how they are used in practice.

A somewhat formal definition of GPs, taken from Wikipedia, is as follows:

"The collection of random variables indexed by time or space, such that every finite collection of those random variables has a multivariate normal distribution, i.e. every finite linear combination of them is normally distributed."

The trick to understanding Gaussian processes is to realize that the concept of GP is a mental (and mathematical) scaffold, since, in practice, we do not need to directly work with this infinite mathematical object. Instead, we only evaluate the GPs at the points where we have data. By doing this, we *collapse* the infinite-dimensional GP into a finite multivariate Gaussian distribution with as many dimensions as data points. Mathematically, this *collapse* is effected by marginalization over the infinitely unobserved dimensions. Theory guarantees us that it is OK to *omit* (actually marginalize over) all points, except the ones we are observing. It also guarantees that we will always get a multivariate Gaussian distribution. Thus, we can rigorously interpret *Figure 7.3* as actual samples from a Gaussian process!

Notice that we set the mean of the multivariate Gaussian at zero and model the functions using just the covariance matrix, through the exponentiated quadratic kernel. Setting the mean of a multivariate Gaussian at zero is common practice when working with Gaussian processes.



Gaussian processes are useful for building Bayesian non-parametric models, as we can use them as prior distributions over functions.

Gaussian process regression

Let's assume we can model a value y as a function f of x plus some noise:

$$y \sim \mathcal{N}(\mu = f(x), \sigma = \epsilon) \quad (7.5)$$

Here $\epsilon \sim \mathcal{N}(0, \sigma_\epsilon)$

This is similar to the assumption that we made in *Chapter 3, Modeling with Linear Regression*, for linear regression models. The main difference is that now we will put a prior distribution over f . Gaussian processes can work as such a prior, thus we can write:

$$f(x) \sim \mathcal{GP}(\mu_x, K(x, x')) \quad (7.6)$$

Here, \mathcal{GP} represents a Gaussian process distribution, with μ_x being the mean function and $K(x, x')$ the kernel, or covariance, function. Here, we have used the word *function* to indicate that, mathematically, the mean and covariance are infinite objects, even when, in practice, we always work with finite objects.

If the prior distribution is a GP and the likelihood is a normal distribution, then the posterior is also a GP and we can compute it analytically:

$$p(f(X_*) | X_*, X, y) \sim \mathcal{N}(\mu, \Sigma) \quad (7.7)$$

$$\mu = K_*^T K^{-1} y \quad (7.8)$$

$$\Sigma = K_{**} - K_*^T K^{-1} K_*$$

Here:

- $K = K(X, X)$
- $K_* = K(X_*, X)$
- $K_{**} = K(X_*, X_*)$

X is the observed data point and X_* represents the test points; that is, the *new* points where we want to know the value of the inferred function.

As usual, PyMC3 allows us to perform inference by taking care of almost all the mathematical details for us, and Gaussian processes are no exception. So, let's proceed to create some data and then a PyMC3 model:

```
np.random.seed(42)
x = np.random.uniform(0, 10, size=15)
y = np.random.normal(np.sin(x), 0.1)
plt.plot(x, y, 'o')
true_x = np.linspace(0, 10, 200)
plt.plot(true_x, np.sin(true_x), 'k--')
plt.xlabel('x')
plt.ylabel('f(x)', rotation=0)
```

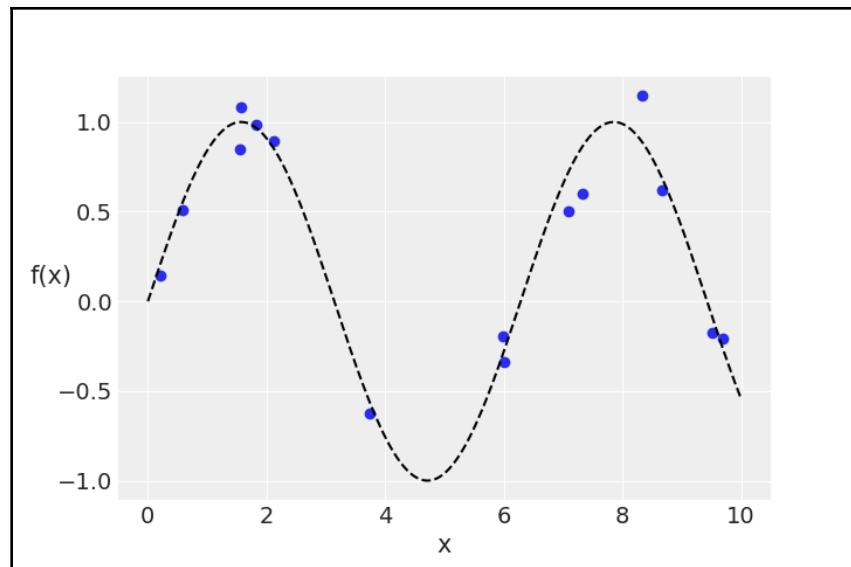


Figure 7.4

In *Figure 7.4*, we see the *true* unknown function as a dashed black line, while the dots represent samples (with noise) from the unknown function.

Notice that in order to code equations 7.7 and 7.8 into a PyMC3 model, we only need to find out the parameter ϵ , the variance of the normal likelihood, ℓ , and the length-scale parameter of the exponentiated quadratic kernel.

GPs are implemented in PyMC3 as a series of Python classes that deviate a little bit from what we have seen in previous models; nevertheless, the code is still very *PyMC3onic*. I have added a few comments in the following code to guide you through the key steps of defining a GP with PyMC3:

```
# A one dimensional column vector of inputs.
X = x[:, None]

with pm.Model() as model_reg:
    # hyperprior for lengthscale kernel parameter
    l = pm.Gamma('l', 2, 0.5)
    # instantiate a covariance function
    cov = pm.gp.cov.ExpQuad(1, ls=l)
    # instantiate a GP prior
    gp = pm.gp.Marginal(cov_func=cov)
    # prior
    epsilon = pm.HalfNormal('epsilon', 25)
```

```
# likelihood
y_pred = gp.marginal_likelihood('y_pred', X=X, y=y, noise=epsilon)
```

Notice that instead of a normal likelihood, as expected from expression 7.7, we have used the `gp.marginal_likelihood` method. As you may remember from Chapter 1, *Thinking Probabilistically*, (equation 1.1) and from Chapter 5, *Model Comparison*, (equation 5.13), the marginal likelihood is the integral of the likelihood and the prior:

$$p(y | X, \theta) \sim \int p(y | f, X, \theta) p(f | X, \theta) df \quad (7.9)$$

As usual, θ represent all the unknown parameters, x is the *independent* variable, and y is the *dependent* variable. Notice that we are marginalizing over the value of the function f . For a GP prior and a normal likelihood, the marginalization can be performed analytically.

According to Bill Engels, core developer of PyMC3, and the main contributor to the GP module, for length-scale parameters, priors avoiding zero often work better. A useful default prior for ℓ is `pm.Gamma(2, 0.5)`. You can read more advice about default useful priors from the Stan team: <https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations>:

```
az.plot_trace(trace_reg)
```

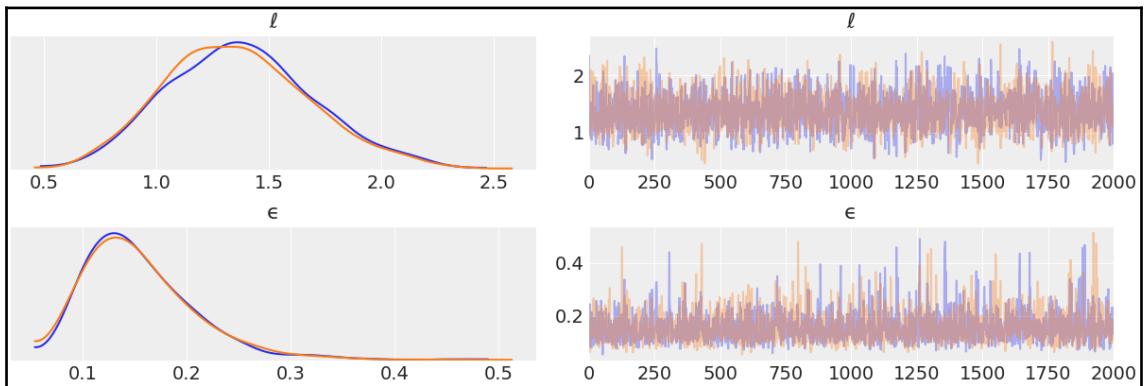


Figure 7.5

Now that we have found the values of ℓ and ϵ , we may want to get samples from the GP posterior; that is, samples of the functions fitting the data. We can do this by computing the *conditional* distribution evaluated over new input locations using the `gp.conditional` function:

```
x_new = np.linspace(np.floor(x.min()), np.ceil(x.max()), 100)[:,None]

with model_reg:
    f_pred = gp.conditional('f_pred', x_new)
```

As a result, we get a new PyMC3 random variable, `f_pred`, that we can use to get samples from the posterior predictive distribution (evaluated at the `x_new` values):

```
with model_reg:
    pred_samples = pm.sample_posterior_predictive(trace_reg, vars=[f_pred],
samples=82)
```

And now we can plot the fitted functions over the original data, to visually inspect how well they fit the data and the associated uncertainty in our predictions:

```
_, ax = plt.subplots(figsize=(12,5))
ax.plot(X_new, pred_samples['f_pred'].T, 'C1-', alpha=0.3)
ax.plot(X, y, 'ko')
ax.set_xlabel('X')
```

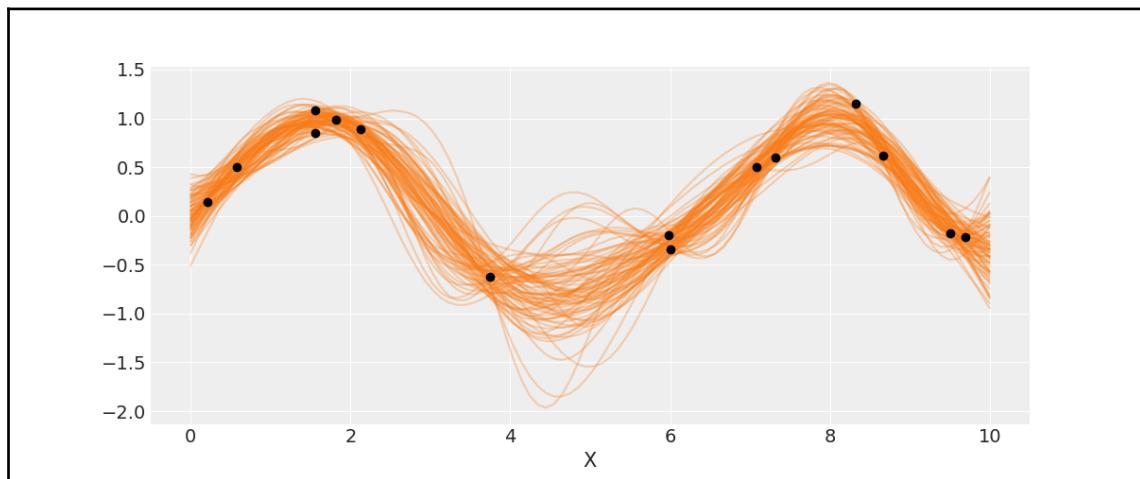


Figure 7.6

Alternatively, we can use the `pm.gp.util.plot_gp_dist` function to get some nice plots. Each plot represents a percentile ranging from 51 (lighter color) to 99 (darker color):

```
_ , ax = plt.subplots(figsize=(12, 5))

pm.gp.util.plot_gp_dist(ax, pred_samples['f_pred'], X_new,
palette='viridis', plot_samples=False);

ax.plot(X, y, 'ko')
ax.set_xlabel('x')
ax.set_ylabel('f(x)', rotation=0, labelpad=15)
```

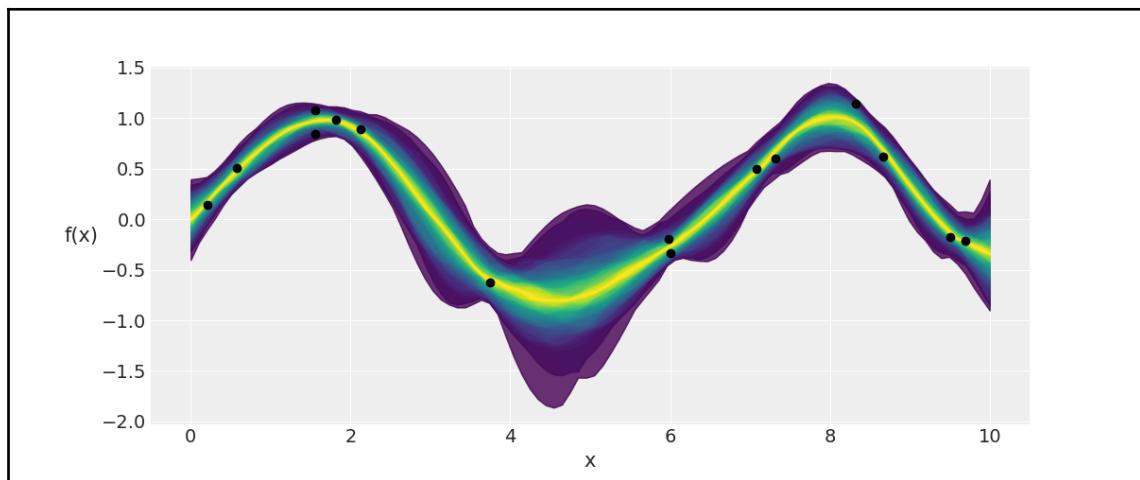


Figure 7.7

Yet another alternative is to compute the mean vector and standard deviation of the conditional distribution evaluated at a given point in the parameter space. In the following example, we use the mean (over the samples in the trace) for ℓ and ϵ . We can compute the mean and variance using the `gp.predict` function. We can do this because PyMC3 has computed the posterior analytically:

```
_ , ax = plt.subplots(figsize=(12, 5))

point = {' $\ell\ell$ '].mean(), ' $\epsilon\epsilon$ '].mean()}
mu, var = gp.predict(X_new, point=point, diag=True)
sd = var**0.5

ax.plot(X_new, mu, 'C1')
ax.fill_between(X_new.flatten(),
    mu - sd, mu + sd,
```

```
color="C1",
alpha=0.3)

ax.fill_between(X_new.flatten(),
                 mu - 2*sd, mu + 2*sd,
                 color="C1",
                 alpha=0.3)

ax.plot(X, y, 'ko')
ax.set_xlabel('X')
```

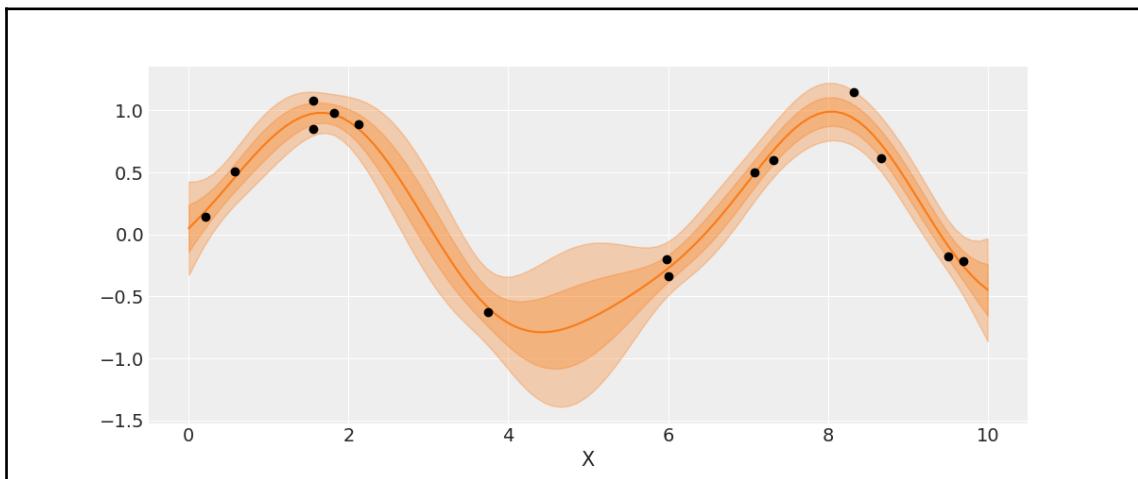


Figure 7.8

As we saw in Chapter 4, *Generalizing Linear Models*, we can use a linear model with a non-Gaussian likelihood and a proper inverse link function to extend the range of linear models. We can do the same for GPs. We can, for example, use a Poisson likelihood with an exponential inverse link function. For a model like this, the posterior is no longer analytically tractable, but, nevertheless, we can use numerical methods to approximate it. In the following sections, we will discuss this type of model.

Regression with spatial autocorrelation

The following example is taken from the book, *Statistical Rethinking*, by Richard McElreath. The author kindly allowed me to reuse his example here. I strongly recommend reading his book, as you will find many good examples like this and very good explanations. The only caveat is that the book examples are in R/Stan, but don't worry and keep sampling; you will find the Python/PyMC3 version of those examples in <https://github.com/pymc-devs/resources>.

Well, going back to the example, we have 10 different island-societies; for each one of them, we have the number of tools they use. Some theories predict that larger populations develop and sustain more tools than smaller populations. Another important factor is the contact rates among populations.

As we have *number of tools* as dependent variable, we can use a Poisson regression with the population, as an independent variable. In fact, we can use the logarithm of the population because what really matters (according to theory) is the order of magnitude of the populations and not the absolute size. One way to include the contact rate in our model is to gather information about how frequent these societies were in contact through history and create a categorical variable such as low/high rate (see the `contact` column in the `islands` DataFrame). Yet another way is to use the distance between societies as a proxy of the contact rate, since it is reasonable to assume that the closest societies come into contact more often than the distant ones.

We can access a distance matrix with the values expressed in thousands of kilometers, by reading the `islands_dist.csv` file that comes with this book:

```
islands_dist = pd.read_csv('../data/islands_dist.csv',
                           sep=',', index_col=0)
islands_dist.round(1)
```

	MI	Ti	SC	Ya	Fi	Tr	Ch	Mn	To	Ha
Malekula	0.0	0.5	0.6	4.4	1.2	2.0	3.2	2.8	1.9	5.7
Tikopia	0.5	0.0	0.3	4.2	1.2	2.0	2.9	2.7	2.0	5.3
Santa Cruz	0.6	0.3	0.0	3.9	1.6	1.7	2.6	2.4	2.3	5.4
Yap	4.4	4.2	3.9	0.0	5.4	2.5	1.6	1.6	6.1	7.2
Lau Fiji	1.2	1.2	1.6	5.4	0.0	3.2	4.0	3.9	0.8	4.9
Trobriand	2.0	2.0	1.7	2.5	3.2	0.0	1.8	0.8	3.9	6.7
Chuuk	3.2	2.9	2.6	1.6	4.0	1.8	0.0	1.2	4.8	5.8
Manus	2.8	2.7	2.4	1.6	3.9	0.8	1.2	0.0	4.6	6.7
Tonga	1.9	2.0	2.3	6.1	0.8	3.9	4.8	4.6	0.0	5.0
Hawaii	5.7	5.3	5.4	7.2	4.9	6.7	5.8	6.7	5.0	0.0

As you can see, the main diagonal is filled with zeros. Each island society is at zero kilometers of itself. The matrix is also symmetrical; both the upper triangle and the lower triangle have the same information. This is a direct consequence of the fact that the distance from point A to B is the same as point B to A.

The number of tools and the population size is stored in another file, `islands.csv`, which is also distributed with the book:

```
islands = pd.read_csv('../data/islands.csv', sep=',')
islands.head().round(1)
```

	culture	population	contact	total_tools	mean_TU	lat	lon	lon2	logpop
0	Malekula	1100	low	13	3.2	-16.3	167.5	-12.5	7.0
1	Tikopia	1500	low	22	4.7	-12.3	168.8	-11.2	7.3
2	Santa Cruz	3600	low	24	4.0	-10.7	166.0	-14.0	8.2
3	Yap	4791	high	43	5.0	9.5	138.1	-41.9	8.5
4	Lau Fiji	7400	high	33	5.0	-17.7	178.1	-1.9	8.9

From this DataFrame, we are only going to use the columns `culture`, `total_tools`, `lat`, `lon2`, and `logpop`:

```
islands_dist_sqr = islands_dist.values**2
culture_labels = islands.culture.values
index = islands.index.values
log_pop = islands.logpop
total_tools = islands.total_tools
x_data = [islands.lat.values[:, None], islands.lon.values[:, None]]
```

The model we are going to build is:

$$f \sim \mathcal{GP}([0, \dots, 0], K(x, x')) \quad (7.10)$$

$$\mu \sim \exp(\alpha + \beta x + f) \quad (7.11)$$

$$y \sim \text{Poisson}(\mu) \quad (7.12)$$

Here, we are omitting the priors for α and β , as well as the kernel's hyperpriors. x is the log population and y is the total number of tools.

Basically, this model is a Poisson regression with the novelty, compared to the models in Chapter 4, *Generalizing Linear Models*, that one of the terms in the linear model f comes from a GP. For computing the kernel of the GP, we will use the distance matrix, `islands_dist`. In this way, we will be effectively incorporating a measure of similarity in technology exposure (estimated from the distance matrix). Thus, instead of assuming the total number is just a consequence of population alone and independent from one society to the next, we will be modeling the number of tools in each society as a function of their geographical similarity.

This model, including priors, looks like the following code in PyMC3:

```
with pm.Model() as model_islands:
    η = pm.HalfCauchy('η', 1)
    ℓ = pm.HalfCauchy('ℓ', 1)
    cov = η * pm.gp.cov.ExpQuad(1, ls=ℓ)
    gp = pm.gp.Latent(cov_func=cov)
    f = gp.prior('f', X=islands_dist_sqr)

    α = pm.Normal('α', 0, 10)
    β = pm.Normal('β', 0, 1)
    μ = pm.math.exp(α + f[index] + β * log_pop)
    tt_pred = pm.Poisson('tt_pred', μ, observed=total_tools)
    trace_islands = pm.sample(1000, tune=1000)
```

In order to understand the posterior distribution of covariance functions in terms of distances, we can plot some samples from the posterior distribution:

```
trace_η = trace_islands['η']
trace_ℓ = trace_islands['ℓ']

_, ax = plt.subplots(1, 1, figsize=(8, 5))
xrange = np.linspace(0, islands_dist.values.max(), 100)

ax.plot(xrange, np.median(trace_η) *
        np.exp(-np.median(trace_ℓ) * xrange**2), lw=3)

ax.plot(xrange, (trace_η[::20][:, None] * np.exp(-trace_ℓ[::20][:, None] *
        xrange**2)).T,
        'C0', alpha=.1)

ax.set_xlim(0, 1)
ax.set_xlabel('distance (thousand kilometers)')
ax.set_ylabel('covariance')
```

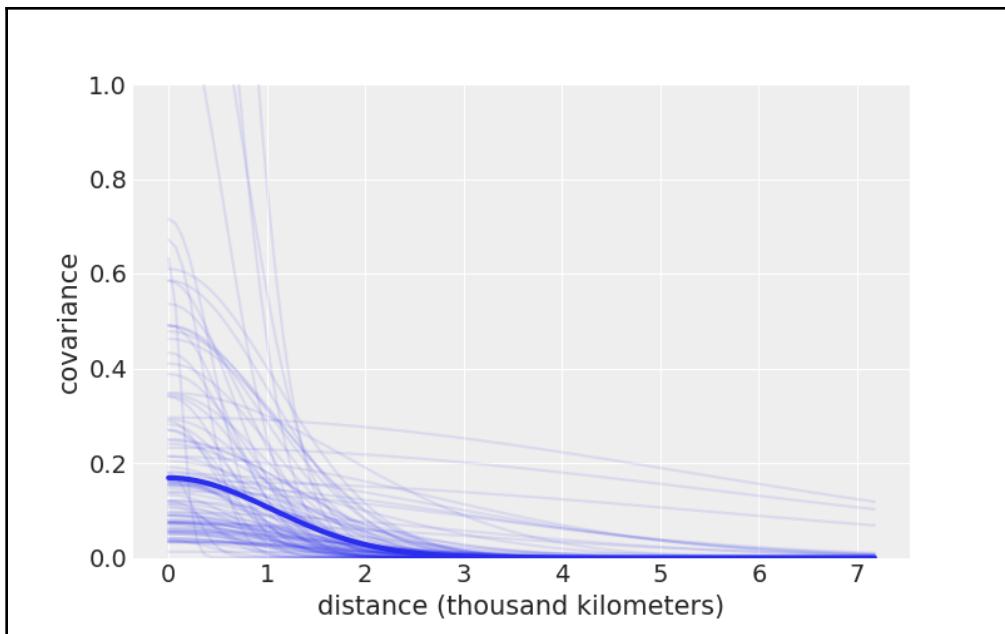


Figure 7.9

The thick line in *Figure 7.9* is the posterior median of the covariance between pairs of societies as a function of distance. We use the median because the distribution for ℓ and η is very skewed. We can see that the covariance is, on average, not that high and also drops to almost 0 at about 2,000 kilometers. The thin lines represent the uncertainty and we can see that there is a lot of uncertainty.

You may find it interesting to compare `model_islands`, and the posterior computed from it, with `model_m_10_10` in <https://github.com/pymc-devs/resources>. You may want to use ArviZ functions, such as `az.summary` or `az.plot_forest`. Model `m_10_10` is similar to `model_islands`, but without including a Gaussian process term.

We are now going to explore how strong the islands-societies are correlated among them according to our model. In order to do this, we have to turn the covariance matrix into a correlation matrix:

```
# compute posterior median covariance among societies
Σ = np.median(trace_η) * (np.exp(-np.median(trace_ℓ) * islands_dist_sqr))
# convert to correlation matrix
Σ_post = np.diag(np.diag(Σ)**-0.5)
ρ = Σ_post @ Σ @ Σ_post
ρ = pd.DataFrame(ρ, index=islands_dist.columns,
```

```
columns=islands_dist.columns)
p.round(2)
```

	MI	Ti	SC	Ya	Fi	Tr	Ch	Mn	To	Ha
MI	1.00	0.90	0.84	0.00	0.50	0.16	0.01	0.03	0.21	0.0
Ti	0.90	1.00	0.96	0.00	0.50	0.16	0.02	0.04	0.18	0.0
SC	0.84	0.96	1.00	0.00	0.34	0.27	0.05	0.08	0.10	0.0
Ya	0.00	0.00	0.00	1.00	0.00	0.07	0.34	0.31	0.00	0.0
Fi	0.50	0.50	0.34	0.00	1.00	0.01	0.00	0.00	0.77	0.0
Tr	0.16	0.16	0.27	0.07	0.01	1.00	0.23	0.72	0.00	0.0
Ch	0.01	0.02	0.05	0.34	0.00	0.23	1.00	0.52	0.00	0.0
Mn	0.03	0.04	0.08	0.31	0.00	0.72	0.52	1.00	0.00	0.0
To	0.21	0.18	0.10	0.00	0.77	0.00	0.00	0.00	1.00	0.0
Ha	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.0

Two observations that pop up from the rest is that Hawaii is very *lonely*. This makes sense, as Hawaii is very far away from the rest of the islands-societies. Also, we can see that **Malekula (MI)**, **Tikopia (Ti)**, and **Santa Cruz (SC)**, are highly correlated with one another. This also makes sense, as these societies are very close together, and they also have a similar number of tools.

Now we are going to use the latitude and longitude information to plot the islands-societies in their relative positions:

```
# scale point size to logpop
logpop = np.copy(log_pop)
logpop /= logpop.max()
psize = np.exp(logpop*5.5)
log_pop_seq = np.linspace(6, 14, 100)
lambda_post = np.exp(trace_islands['α'][:, None] +
                      trace_islands['β'][:, None] * log_pop_seq)

_, ax = plt.subplots(1, 2, figsize=(12, 6))

ax[0].scatter(islands.lon2, islands.lat, psize, zorder=3)
ax[1].scatter(islands.logpop, islands.total_tools, psize, zorder=3)

for i, itext in enumerate(culture_labels):
    ax[0].text(islands.lon2[i]+1, islands.lat[i]+1, itext)
    ax[1].text(islands.logpop[i]+1, islands.total_tools[i]-2.5, itext)

ax[1].plot(log_pop_seq, np.median(lambda_post, axis=0), 'k--')

az.plot_hpd(log_pop_seq, lambda_post, fill_kwarg={'alpha':0},
```

```

plot_kwarg={ 'color':'k', 'ls': '--', 'alpha':1})

for i in range(10):
    for j in np.arange(i+1, 10):
        ax[0].plot((islands.lon2[i], islands.lon2[j]),
                    (islands.lat[i], islands.lat[j]), 'C1-',
                    alpha=p.iloc[i, j]**2, lw=4)
        ax[1].plot((islands.logpop[i], islands.logpop[j]),
                    (islands.total_tools[i], islands.total_tools[j]), 'C1-',
                    alpha=p.iloc[i, j]**2, lw=4)
ax[0].set_xlabel('longitude')
ax[0].set_ylabel('latitude')

ax[1].set_xlabel('log-population')
ax[1].set_ylabel('total tools')
ax[1].set_xlim(6.8, 12.8)
ax[1].set_ylim(10, 73)

```

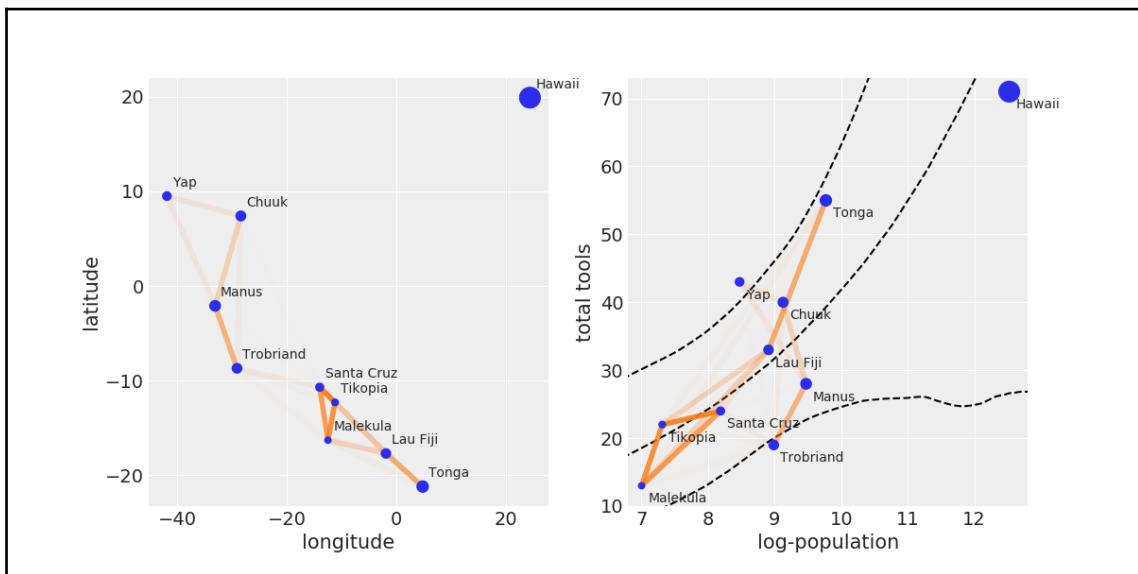


Figure 7.10

The left-hand panel of *Figure 7.10* shows the lines the posterior median correlations we previously computed among societies in the context of the relative geographical positions. Some of the lines are not visible, since we have used the correlation to set the opacity of the lines (with matplotlib's `alpha` parameter). On the right panel, we have again the posterior median correlations, but this time plotted in terms of the log-population versus the total number of tools. The dashed lines represent the median number of tools and the HPD 94% interval as a function of *log-population*. In both panels, the size of the dots is proportional to the population of each island-society.

Notice how the correlations among Malekula, Tikopia, and Santa Cruz describe the fact that they have a rather low number of tools close to the median or lower than the expected number of tools for their populations. Something similar is happening with Trobriands and Manus; they are geographically close and have fewer tools than expected for their population sizes. Tonga has way more tools than expected for its population and a relative high correlation with Fiji. In a way, the model is telling us that Tonga has a positive effect on Lau Fiji, increasing the total number of tools and counteracting the effect of its close neighbors, Malekula, Tikopia, and Santa Cruz.

Gaussian process classification

Gaussian processes are not restricted to regression. We can also use them for classification. As we saw in [Chapter 4, Generalizing Linear Models](#), we turn a linear model into a suitable model to classify data by using a Bernoulli likelihood with a logistic inverse link function (and then applying a boundary decision rule to separate classes). We will try to recapitulate `model_0` from [Chapter 4, Generalizing Linear Models](#), for the iris dataset, this time using a GP instead of a linear model.

Let's invite the iris dataset to the stage one more time:

```
iris = pd.read_csv('~/data/iris.csv')
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

We are going to begin with the simplest possible classification problem: two classes, setosa and versicolor, and just one independent variable, the sepal length. As per usual, we are going to encode the categorical variables, setosa and versicolor, with the numbers 0 and 1:

```
df = iris.query("species == ('setosa', 'versicolor')")
y = pd.Categorical(df['species']).codes
x_1 = df['sepal_length'].values
X_1 = x_1[:, None]
```

For this model, instead of using the `pm.gp.Marginal` class to instantiate a GP prior, we are going to use the `pm.gp.Latent` class. While the latter is more general and can be used with any likelihood, the former is restricted to Gaussian likelihoods and has the advantage of being more efficient by taking advantage of the mathematical tractability of the combination of a GP prior with a Gaussian likelihood:

```
with pm.Model() as model_iris:
    ℓ = pm.Gamma('ℓ', 2, 0.5)
    cov = pm.gp.cov.ExpQuad(1, ℓ)
    gp = pm.gp.Latent(cov_func=cov)
    f = gp.prior("f", X=X_1)
    # logistic inverse link function and Bernoulli likelihood
    y_ = pm.Bernoulli("y", p=pm.math.sigmoid(f), observed=y)
    trace_iris = pm.sample(1000, chains=1,
                          compute_convergence_checks=False)
```

Now that we have found the values of ℓ , we may want to get samples from the GP posterior. As we did with the `marginal_gp_model`, we can also compute the *conditional* distribution evaluated over a set of new input locations with the help of the `gp.conditional` function, as shown in the following code:

```
X_new = np.linspace(np.floor(x_1.min()), np.ceil(x_1.max()), 200)[:, None]

with model_iris:
    f_pred = gp.conditional('f_pred', X_new)
    pred_samples = pm.sample_posterior_predictive(
        trace_iris, vars=[f_pred], samples=1000)
```

To show the results for this model, we are going to create a figure similar to *Figure 4.4*. Instead of obtaining the boundary decision analytically, we will compute it directly from `f_pred`, using the following convenience function:

```
def find_midpoint(array1, array2, value):
    """
    This should be a proper docstring :-)
    """
    array1 = np.asarray(array1)
    idx0 = np.argsort(np.abs(array1 - value))[0]
    idx1 = idx0 - 1 if array1[idx0] > value else idx0 + 1
    if idx1 == len(array1):
        idx1 -= 1
    return (array2[idx0] + array2[idx1]) / 2
```

The following code is very similar to that used in Chapter 4, *Generalizing Linear Models*, to generate *Figure 4.4*:

```
_ , ax = plt.subplots(figsize=(10, 6))

fp = logistic(pred_samples['f_pred'])
fp_mean = np.mean(fp, 0)

ax.plot(X_new[:, 0], fp_mean)
# plot the data (with some jitter) and the true latent function
ax.scatter(x_1, np.random.normal(y, 0.02),
           marker='.', color=[f'C{x}' for x in y])

az.plot_hpd(X_new[:, 0], fp, color='C2')

db = np.array([find_midpoint(f, X_new[:, 0], 0.5) for f in fp])
db_mean = db.mean()
db_hpd = az.hpd(db)
ax.vlines(db_mean, 0, 1, color='k')
ax.fill_betweenx([0, 1], db_hpd[0], db_hpd[1], color='k', alpha=0.5)
ax.set_xlabel('sepal_length')
ax.set_ylabel('θ', rotation=0)
plt.savefig('B11197_07_11.png')
```

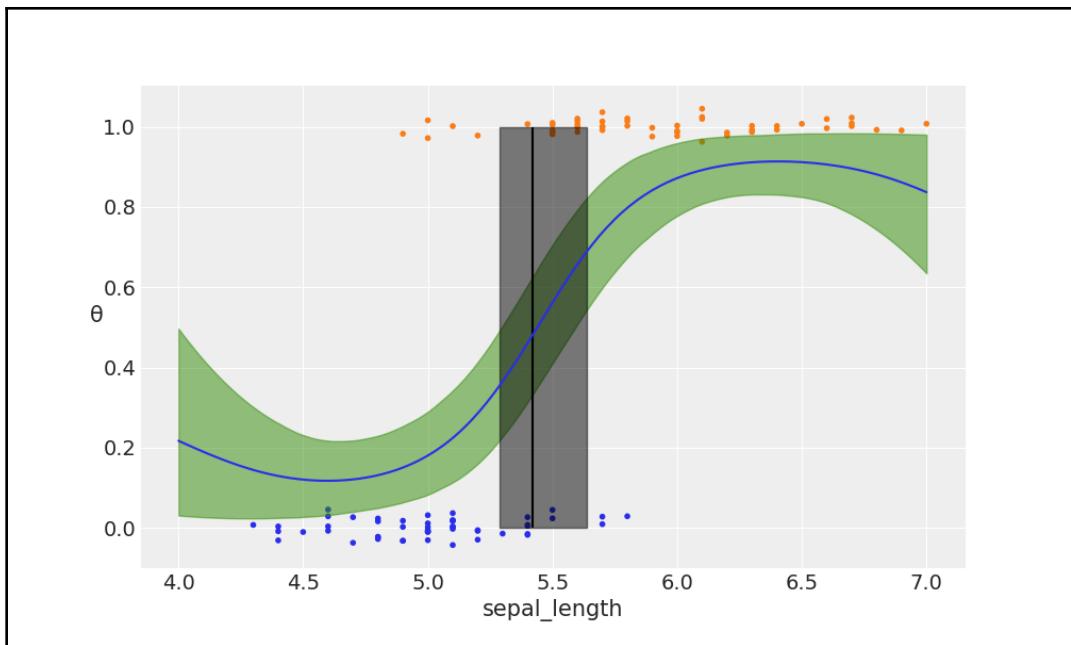


Figure 7.11

As we can see, *Figure 7.11* looks pretty similar to *Figure 4.4*. The f_{pred} looks like a sigmoid curve, except for the tails that go up at lower values of x_1 and down at higher values of x_1 . This is a consequence of the predicted function moving toward the prior when there is no data (or few data). If we only concern ourselves with the boundary decision, this should not be a real problem, but if we want to model the probabilities of belonging to setosa or versicolor for different values of `sepal_length`, then we should improve our model and do something to get a better model for the tails. One way to achieve this objective is to add more structure to the Gaussian process. A general way to get better Gaussian process models is by combining covariance functions in order to better capture details of the function we are trying to model.

The following model (`model_iris2`) is the same as `model_iris`, except for the covariance matrix, which we model as the combination of three kernels:

```
cov = K_{ExpQuad} + K_{Linear} + K_{whitenoise}(1E-5)
```

By adding the linear kernel, we fix the tail problem, as you can see in *Figure 7.12*. The white-noise kernel is just a computational trick to *stabilize* the computation of the covariance matrix. Kernels for Gaussian processes are restricted to guarantee the resulting covariance matrix is positive definite. Nevertheless, numerical errors can lead to violating this condition. One manifestation of this problem is that we get `nans` when computing posterior predictive samples of the fitted function. One way to mitigate this error is to stabilize the computation by adding a little bit of noise. As a matter of fact, PyMC3 already does this *under the hood*, but sometimes a little bit more noise is needed, as shown in the following code:

```
with pm.Model() as model_iris2:
    ℓ = pm.Gamma('ℓ', 2, 0.5)
    c = pm.Normal('c', x_1.min())
    τ = pm.HalfNormal('τ', 5)
    cov = (pm.gp.cov.ExpQuad(1, ℓ) +
           τ * pm.gp.cov.Linear(1, c) +
           pm.gp.cov.WhiteNoise(1E-5))
    gp = pm.gp.Latent(cov_func=cov)
    f = gp.prior("f", X=X_1)
    # logistic inverse link function and Bernoulli likelihood
    y_ = pm.Bernoulli("y", p=pm.math.sigmoid(f), observed=y)
    trace_iris2 = pm.sample(1000, chains=1,
                           compute_convergence_checks=False)
```

Now we generate posterior predictive samples for `model_iris2` for the values of `x_new` generated previously:

```
with model_iris2:
    f_pred = gp.conditional('f_pred', X_new)
    pred_samples = pm.sample_posterior_predictive(trace_iris2,
                                                   vars=[f_pred],
                                                   samples=1000)

    _, ax = plt.subplots(figsize=(10, 6))

    fp = logistic(pred_samples['f_pred'])
    fp_mean = np.mean(fp, 0)

    ax.scatter(x_1, np.random.normal(y, 0.02), marker='.',
               color=[f'C{ci}' for ci in y])

    db = np.array([find_midpoint(f, X_new[:, 0], 0.5) for f in fp])
    db_mean = db.mean()
    db_hpd = az.hpd(db)
    ax.vlines(db_mean, 0, 1, color='k')
    ax.fill_betweenx([0, 1], db_hpd[0], db_hpd[1], color='k', alpha=0.5)
```

```

ax.plot(X_new[:,0], fp_mean, 'C2', lw=3)
az.plot_hpd(X_new[:,0], fp, color='C2')

ax.set_xlabel('sepal_length')
ax.set_ylabel('θ', rotation=0)

```

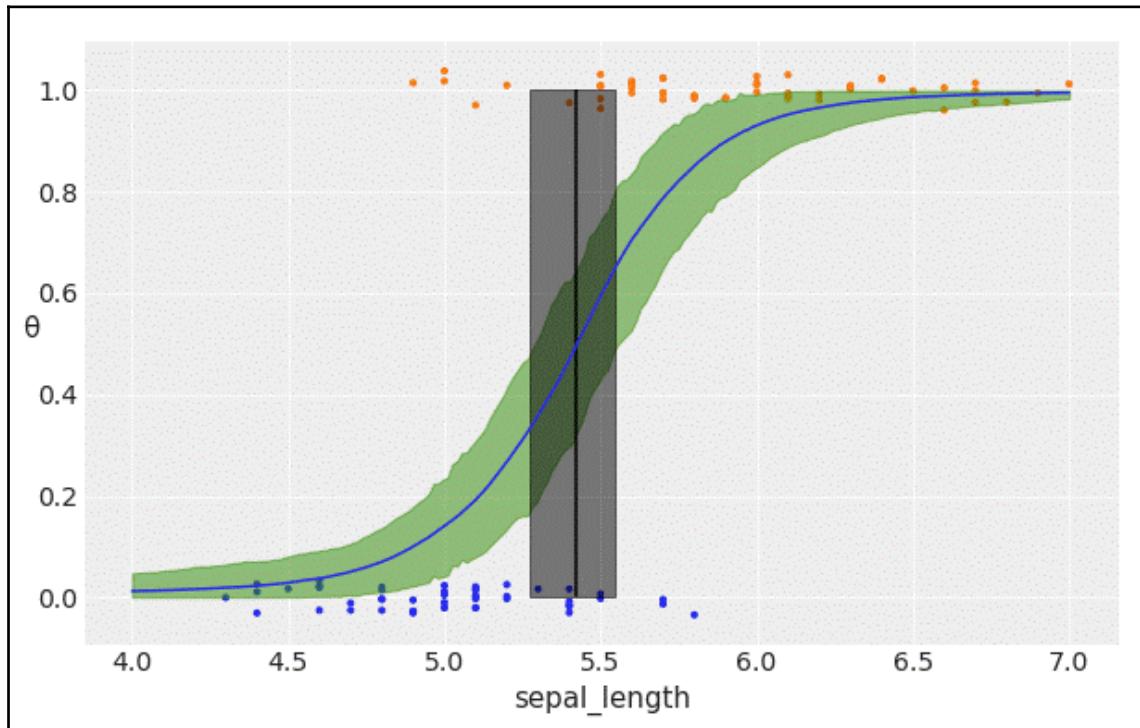


Figure 7.12

Now *Figure 7.12* looks much more similar to *Figure 4.4* than *Figure 7.11*. This example has two main aims:

- Showing how we can easily combine kernels to get a more expressive model
- Showing how we can recover a logistic regression using a Gaussian process

Regarding the second point, a logistic regression is indeed a special case of Gaussian processes, because a simple linear regression is just a particular case of a Gaussian process. In fact, many known models can be seen as special cases of GPs, or at least they are somehow connected to GPs. You can read Chapter 15 from Kevin Murphy's *Machine Learning: A Probabilistic Perspective* for details.

In practice, it does not make too much sense to use a GP to model a problem we can just solve with a logistic regression. Instead, we want to use a GP to model more complex data that is not well-captured with less flexible models. For example, suppose we want to model the probability of getting a disease as a function of age. It turns out that very young and very old people have a higher risk than people of middle age. The dataset `space_flu.csv` is a fake dataset inspired by the previous description. Let's load it:

```
df_sf = pd.read_csv('~/data/space_flu.csv')
age = df_sf.age.values[:, None]
space_flu = df_sf.space_flu

ax = df_sf.plot.scatter('age', 'space_flu', figsize=(8, 5))
ax.set_yticks([0, 1])
ax.set_yticklabels(['healthy', 'sick'])
```

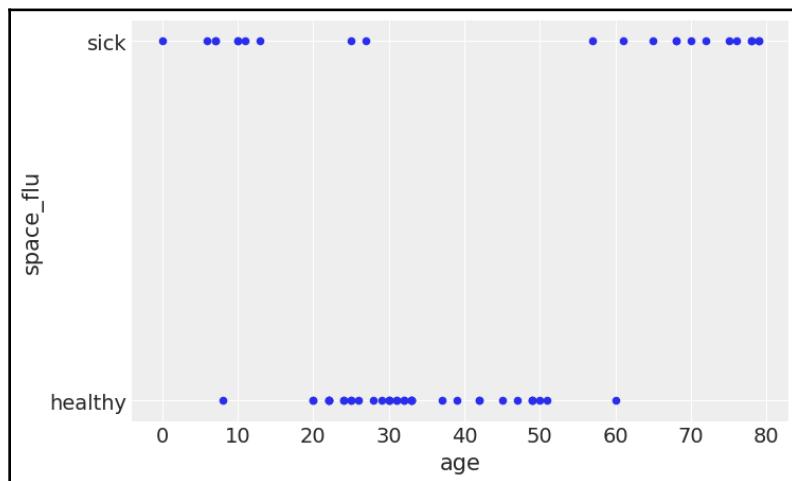


Figure 7.13

The following model is basically the same as `model_iris`:

```
with pm.Model() as model_space_flu:
    ℓ = pm.HalfCauchy('ℓ', 1)
    cov = pm.gp.cov.ExpQuad(1, ℓ) + pm.gp.cov.WhiteNoise(1E-5)
    gp = pm.gp.Latent(cov_func=cov)
    f = gp.prior('f', X=age)
    y_ = pm.Bernoulli('y', p=pm.math.sigmoid(f), observed=space_flu)
    trace_space_flu = pm.sample(
        1000, chains=1, compute_convergence_checks=False)
```

Now we generate posterior predictive samples for `model_space_flu` and then plot the results:

```
X_new = np.linspace(0, 80, 200)[:, None]

with model_space_flu:
    f_pred = gp.conditional('f_pred', X_new)
    pred_samples = pm.sample_posterior_predictive(trace_space_flu,
                                                   vars=[f_pred],
                                                   samples=1000)

_, ax = plt.subplots(figsize=(10, 6))

fp = logistic(pred_samples['f_pred'])
fp_mean = np.nanmean(fp, 0)

ax.scatter(age, np.random.normal(space_fiu, 0.02),
           marker='.', color=[f'C{ci}' for ci in space_fiu])

ax.plot(X_new[:, 0], fp_mean, 'C2', lw=3)

az.plot_hpd(X_new[:, 0], fp, color='C2')
ax.set_yticks([0, 1])
ax.set_yticklabels(['healthy', 'sick'])
ax.set_xlabel('age')
```

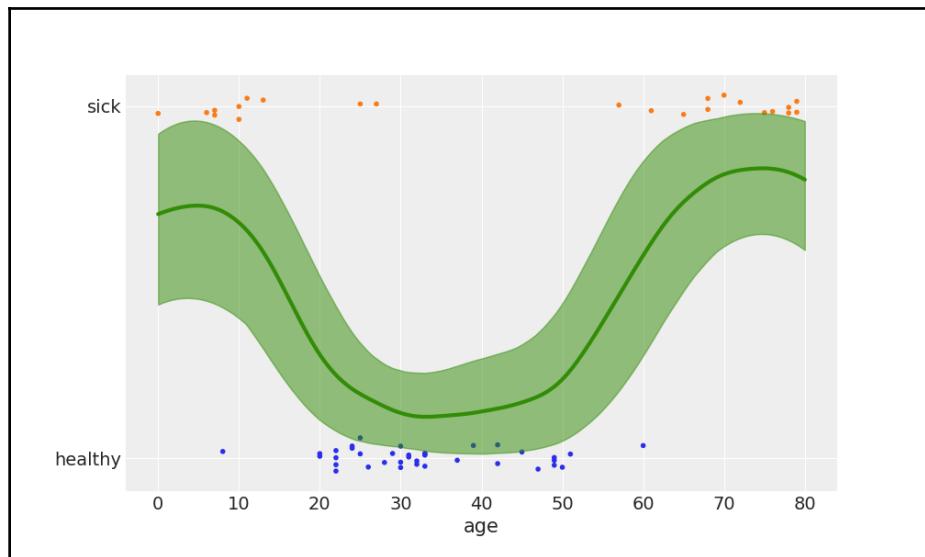


Figure 7.14

Notice, as illustrated in *Figure 7.14*, that the GP is able to fit this dataset very well, even when the data demands the function to be more complex than a logistic one. Fitting this data well will be impossible for a simple logistic regression, unless we introduce some *ad hoc* modifications to help it a little bit (see exercise 6 for a discussion of such modifications).

Cox processes

Let's now return to the example of modeling count data. We will see two examples; one with a time varying rate and one with a 2D-spatially varying rate. In order to do this, we will use a Poisson likelihood and the rate will be modeled using a Gaussian process. Because the rate of the Poisson distribution is limited to positive values, we will use an exponential as the inverse link function, as we did for the zero-inflated Poisson regression from Chapter 4, *Generalizing Linear Models*.

In the literature, the variable *rate* also appears with the name *intensity*; thus, this type of problem is known as **intensity estimation**. Also, this type of model is often referred to as a Cox model. A Cox model is a type of Poisson process, where the rate is itself a stochastic process. Just as a Gaussian process is a collection of random variables, where every finite collection of those random variables has a multivariate normal distribution, a Poisson process is a collection of random variables, where every finite collection of those random variables has a Poisson distribution. We can think of a Poisson process as a distribution over collections of points in a given space. When the rate of the Poisson process is itself a stochastic process, such as, for example, a Gaussian process, then we have what is known as a **Cox process**.

The coal-mining disasters

The first example is known as the coal-mining disaster example. This example consists of a record of coal-mining disasters in the UK from 1851 to 1962. The number of disasters is thought to have been affected by changes in safety regulations during this period. We want to model the rate of disasters as a function of time. Our dataset consists of a single column and each entry corresponds to the time a disaster happened.

Let's load the data and look at some of its values:

```
coal_df = pd.read_csv('../data/coal.csv', header=None)
coal_df.head()
```

	0
0	1851.2026
1	1851.6324
2	1851.9692
3	1851.9747
4	1852.3142

The model we will use to fit the data in the `coal_df` dataframe is:

$$f(x) \sim \mathcal{GP}(\mu_x, K(x, x')) \quad (7.13)$$

$$y \sim Poisson(f(x)) \quad (7.14)$$

As you can see, this a Poisson regression problem. You may be wondering, at this point, how are we going to perform a regression if we only have a single column with just the date of the disasters. The answer is to discretize the data, just as if we were building a histogram. We are going to use the centers of the bins as the variable x and the counts per bin as the variable y :

```
# discretize data
years = int(coal_df.max().values - coal_df.min().values)
bins = years // 4
hist, x_edges = np.histogram(coal_df, bins=bins)
# compute the location of the centers of the discretized data
x_centers = x_edges[:-1] + (x_edges[1] - x_edges[0]) / 2
# arrange xdata into proper shape for GP
x_data = x_centers[:, None]
# express data as the rate number of disaster per year
y_data = hist / 4
```

And now we define and solve the model with PyMC3:

```
with pm.Model() as model_coal:
    ℓ = pm.HalfNormal('ℓ', x_data.std())
    cov = pm.gp.cov.ExpQuad(1, ls=ℓ) + pm.gp.cov.WhiteNoise(1E-5)
    gp = pm.gp.Latent(cov_func=cov)
    f = gp.prior('f', X=x_data)

    y_pred = pm.Poisson('y_pred', mu=pm.math.exp(f), observed=y_data)
    trace_coal = pm.sample(1000, chains=1)
```

Now we plot the results:

```
_ , ax = plt.subplots(figsize=(10, 6))

f_trace = np.exp(trace_coal['f'])
rate_median = np.median(f_trace, axis=0)

ax.plot(x_centers, rate_median, 'w', lw=3)
az.plot_hpd(x_centers, f_trace)

az.plot_hpd(x_centers, f_trace, credible_interval=0.5,
            plot_kwarg={alpha: 0})

ax.plot(coal_df, np.zeros_like(coal_df)-0.5, 'k|')
ax.set_xlabel('years')
ax.set_ylabel('rate')
```

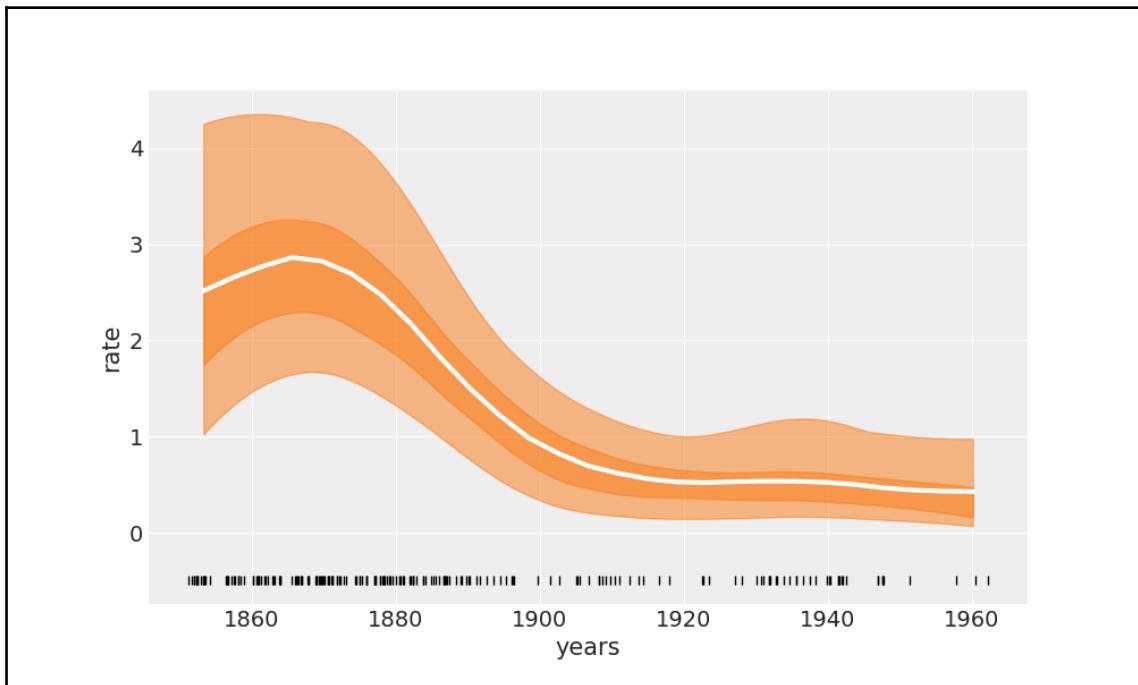


Figure 7.15

Figure 7.15 shows, using a white line, the median disaster rate as a function of time. The bands describe the 50% HPD interval (darker) and the 94% HPD interval (lighter). At the bottom, we have in black markers for each disaster (this is sometimes known as a rug plot). As we can see, the rate of accidents decrease with time, except for a brief initial increase. The PyMC3 documentation includes the coal mining disaster, but modeled from a different perspective. I strongly recommend that you check that example as it is very useful on its own and also it is useful to compare it with the approach we just implemented with the `model_coal` model.

Notice that even when we binned the data, we obtain, as a result, a smooth curve. In this sense, we can see `model_coal` (and, in general, this type of model) as building an histogram and then smoothing it.

The redwood dataset

Now we are going to focus our attention on applying the same type of model that we just did to a 2-D spacial problem, using the redwood data. I take this dataset (distributed with a GPL license) from the **GPstuff** package, a Gaussian process package for Matlab, Octave, and R. The dataset consists of the location of redwood trees over a given area. The motivation of the inference is to identify how the rate of trees is distributed in this area.

As usual, we load the data and plot it:

```
rw_df = pd.read_csv('../data/redwood.csv', header=None)
_, ax = plt.subplots(figsize=(8, 8))
ax.plot(rw_df[0], rw_df[1], 'C0.')
ax.set_xlabel('x1 coordinate')
ax.set_ylabel('x2 coordinate')
```

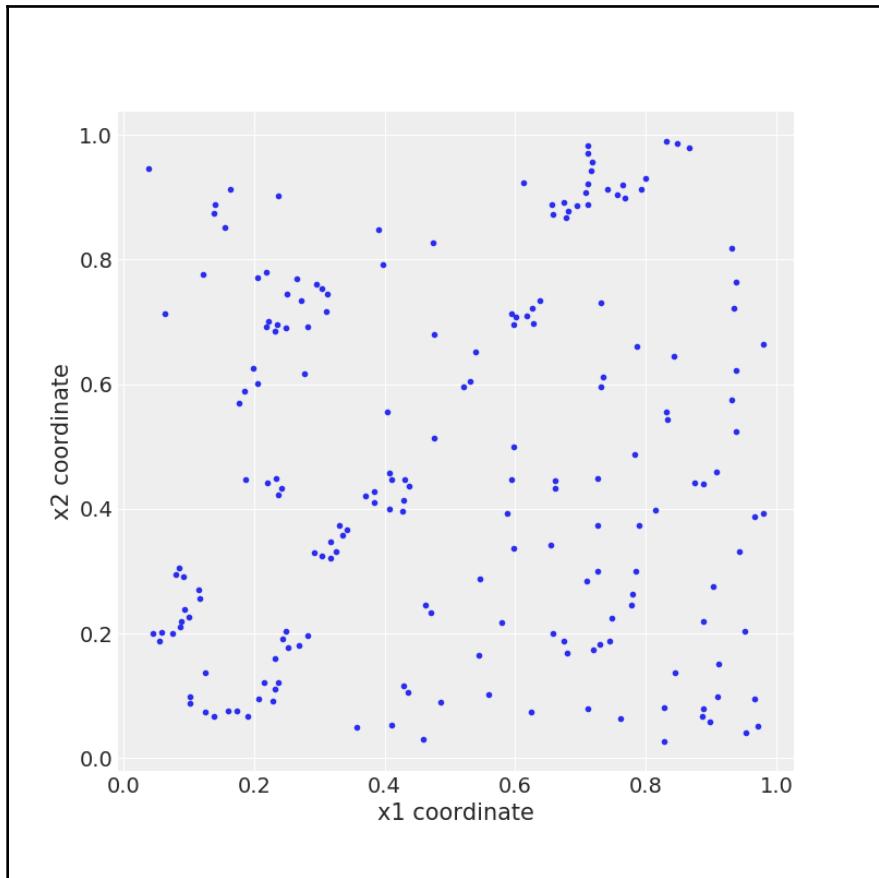


Figure 7.16

As with the coal-mining disaster example, we need to discretize the data:

```
# discretize spatial data
bins = 20
hist, x1_edges, x2_edges = np.histogram2d(
    rw_df[1].values, rw_df[0].values, bins=bins)
# compute the location of the centers of the discretized data
x1_centers = x1_edges[:-1] + (x1_edges[1] - x1_edges[0]) / 2
x2_centers = x2_edges[:-1] + (x2_edges[1] - x2_edges[0]) / 2
# arrange xdata into proper shape for GP
x_data = [x1_centers[:, None], x2_centers[:, None]]
# arrange ydata into proper shape for GP
y_data = hist.flatten()
```

Notice that instead of doing a mesh grid, we treat x_1 and x_2 data as being separated. This allows us to build a covariance matrix for each coordinate, effectively reducing the size of the matrix needed to compute the GP. We only need to take care when using the `LatentKron` class to define the GP. It is important to note that this is not a numerical trick, but a mathematical property of the structure of this type of matrix, so we are not introducing any approximation or error in our model. We are just expressing it in a way that faster computations are possible:

```
with pm.Model() as model_rw:
    ℓ = pm.HalfNormal('ℓ', rw_df.std().values, shape=2)
    cov_func1 = pm.gp.cov.ExpQuad(1, ls=ℓ[0])
    cov_func2 = pm.gp.cov.ExpQuad(1, ls=ℓ[1])

    gp = pm.gp.LatentKron(cov_funcs=[cov_func1, cov_func2])
    f = gp.prior('f', Xs=x_data)

    y = pm.Poisson('y', mu=pm.math.exp(f), observed=y_data)
    trace_rw = pm.sample(1000)
```

And, finally, we plot the results:

```
rate = np.exp(np.mean(trace_rw['f'], axis=0).reshape((bins, -1)))
fig, ax = plt.subplots(figsize=(6, 6))
ims = ax.imshow(rate, origin='lower')
ax.grid(False)
ticks_loc = np.linspace(0, bins-1, 6)
ticks_lab = np.linspace(0, 1, 6).round(1)
ax.set_xticks(ticks_loc)
ax.set_yticks(ticks_loc)
ax.set_xticklabels(ticks_lab)
ax.set_yticklabels(ticks_lab)
cbar = fig.colorbar(ims, fraction=0.046, pad=0.04)
```

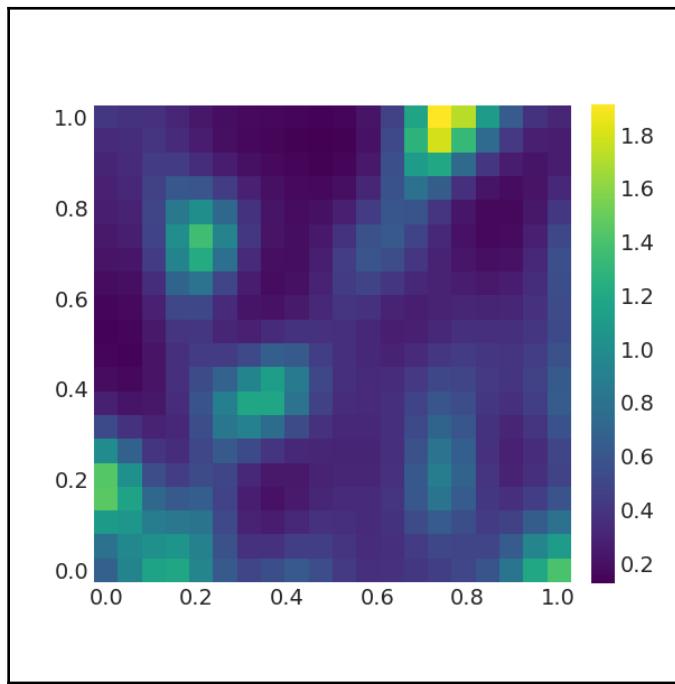


Figure 7.17

In *Figure 7.17*, the lighter color means a higher rate of trees than a darker color. We may imagine that we are interested in finding high growing rate zones, because we may be interested on how a forest, is recovering from a fire or maybe we are interested in the properties of the soil and we use the trees as a proxy.

Summary

A Gaussian process is a generalization of the multivariate Gaussian distribution to infinitely many dimensions and is fully specified by a mean function and a covariance function. Since we can conceptually think of functions as infinitely long vectors, we can use Gaussian processes as priors for functions. In practice, we do not work with infinite objects but with multivariate Gaussian distributions with as many dimensions as data points. To define their corresponding covariance function, we used properly parameterized kernels; and by learning about those hyperparameters, we ended up learning about arbitrary complex functions.

In this chapter, we have given a short introduction to GPs. We have covered regression, semi-parametric models (the islands example), combining two or more kernels to better describe the unknown function, and how a GP can be used for classification tasks. There are many other topics to discuss. Nevertheless, I hope this introduction to GPs has motivated you sufficiently to keep using, reading, and learning about Gaussian process and non-parametric models.

Exercises

1. For the example in the Covariance functions and kernels section make sure you understand the relationship between the input `data` and the generated covariance matrix. Try using other input such as `data = np.random.normal(size=4)`
2. Rerun the code generating *Figure 7.3* and increase the number of samples obtained from the GP-prior to around 200. In the original figure the number of samples is 2. Which is the range of the generated y values?
3. For the generated plot in the previous exercise. Compute the standard deviation for the values of y at each point. Do this in the following form:
 - Visually, just observing the plots
 - Directly from the values generated from `stats.multivariate_normal.rvs`
 - By inspecting the covariance matrix (if you have doubts go back to exercise 1)

Did the values you get from these 3 methods agree?

4. Re-run the model `model_reg` and get new plots but using as
`test_points x_new np.linspace(np.floor(x.min()), 20,
100)[:,None]`. What did you observed? How is this related to the specification
of the GP-prior?
5. Go back to exercise 1, but this time use a linear kernel (see the accompanying
code for a linear kernel)
6. Go and check the section <https://docs.pymc.io/notebooks/GP-MeansAndCovs.html> from PyMC3' documentation
7. Run a logistic regression model for the `space_flu` data. What do you see? Can
you explain the result?
8. Change the logistic regression model in order to fit the data. Tip, use an order
two polynomial.
9. Compare the model for the coal mining disaster with the one from the PyMC3
documentation (https://docs.pymc.io/notebooks/getting_started.html#Case-study-2:-Coal-mining-disasters). Describe the differences between
both models in terms of model-specification and results.

8

Inference Engines

"The first principle is that you must not fool yourself—and you are the easiest person to fool."

- Richard Feynman

So far, we have focused on model building, interpretation of results and criticism of models. We have relied on the *magic* of the `pm.sample` function to compute the posterior distributions for us. Now we will focus on learning some of the details of the inference engines behind this function. The whole purpose of probabilistic programming tools, such as PyMC3, is that the user should not care about how sampling is carried out, but understanding how we get samples from the posterior is important for a full understanding of the inference process, and could also help us to get an idea of when and how these methods fail and what to do about it. If you are not interested in understanding how the methods for approximating the posterior works, you can skip most of this chapter, but I strongly recommend you at least read the *Diagnosing samples* section, as this section provides a few guidelines that will help you to check whether your posterior samples are reliable.

There are many methods for computing the posterior distribution. In this chapter, we will discuss some general ideas and we will focus on the most important methods implemented in PyMC3.

In this chapter, we will learn about:

- Variational methods
- Metropolis-Hastings
- Hamiltonian Monte Carlo
- Sequential Monte Carlo
- Diagnosing samples

Inference engines

While conceptually simple, Bayesian methods can be mathematically and numerically challenging. The main reason is that the marginal likelihood, the denominator in Bayes' theorem (see equation 1.4), usually takes the form of an intractable or computationally-expensive integral to solve. For this reason, the posterior is usually estimated numerically using algorithms from the **Markov Chain Monte Carlo** (MCMC) family or, more recently, variational algorithms. These methods are sometimes called **inference engines**, because, at least in principle, they are capable of approximating the posterior distribution for any probabilistic model. Even when in practice inference does not always work that well, the existence of such methods has motivated the development of probabilistic programming languages such as PyMC3.

The goal of probabilistic programming languages is to separate the model-building process from the inference process to facilitate the iterative steps of model building, evaluation, and model modification/expansion (as discussed in [Chapter 1, Thinking Probabilistically](#), and [Chapter 2, Programming Probabilistically](#)). By treating the inference process (but not the model building process) as a black box, users of probabilistic programming languages such as PyMC3 are free to focus on their specific problems, leaving PyMC3 to handle the computational details for them. This is exactly what we have been doing up to this point. So, you may be biased to think that this is the obvious or *natural* approach. But it is important to notice that before probabilistic programming languages, people doing probabilistic models were also used to writing their own sampling methods, generally tailored to their models, or they were used to simplifying their models to make them suitable for certain mathematical approximations. In fact, this is still true in some academic circles. This *tailored* approach can be more elegant and can even provide a more efficient way of computing a posterior, but it is also error-prone and time-consuming, even for experts. Furthermore, the *tailored* approach is not suitable for most practitioners interested in solving problems with probabilistic models. Software like PyMC3 invites people from a very broad background to work with probabilistic models, lowering the mathematical and computational entry barrier. I personally think this is fantastic and also an invitation to learn more about good practices in statistical modeling so we try to avoid fooling ourselves. The previous chapters have been mostly about learning the basics of Bayesian modeling; now we are going to learn, at a conceptual level, how automatic inference is achieved, when and why it fails, and what to do when it fails.

There are several methods to numerically compute the posterior. I have ordered them into two broad groups:

- Non-Markovian methods
 - Grid computing
 - Quadratic approximation
 - Variational methods
- Markovian methods
 - Metropolis-Hastings
 - Hamiltonian Monte Carlo
 - Sequential Monte Carlo

Non-Markovian methods

Let's start our discussion of inference engines with the non-Markovian methods. Under some circumstances, these methods can provide a fast and accurate enough approximation to the posterior.

Grid computing

Grid computing is a simple brute-force approach. Even if you are not able to compute the whole posterior, you may be able to compute the prior and the likelihood point-wise; this is a pretty common scenario, if not the most common one. Let's assume we want to compute the posterior for a single parameter model, the grid approximation is as follows:

1. Define a reasonable interval for the parameter (the prior should give you a hint).
2. Place a grid of points (generally equidistant) on that interval.
3. For each point in the grid, multiply the likelihood and the prior.

Optionally, we may normalize the computed values, that is, to divide the result at each point by the sum of all points.

The following block of code implements the grid approach to compute the posterior for the coin-flipping model:

```
def posterior_grid(grid_points=50, heads=6, tails=9):  
    """  
    A grid implementation for the coin-flipping problem  
    """
```

```

grid = np.linspace(0, 1, grid_points)
prior = np.repeat(1/grid_points, grid_points) # uniform prior
likelihood = stats.binom.pmf(heads, heads+tails, grid)
posterior = likelihood * prior
posterior /= posterior.sum()
return grid, posterior

```

Assuming we flipped a coin 13 times and we observed three heads:

```

data = np.repeat([0, 1], (10, 3))
points = 10
h = data.sum()
t = len(data) - h
grid, posterior = posterior_grid(points, h, t)
plt.plot(grid, posterior, 'o-')

plt.title(f'heads = {h}, tails = {t}')
plt.yticks([])
plt.xlabel('θ');

```

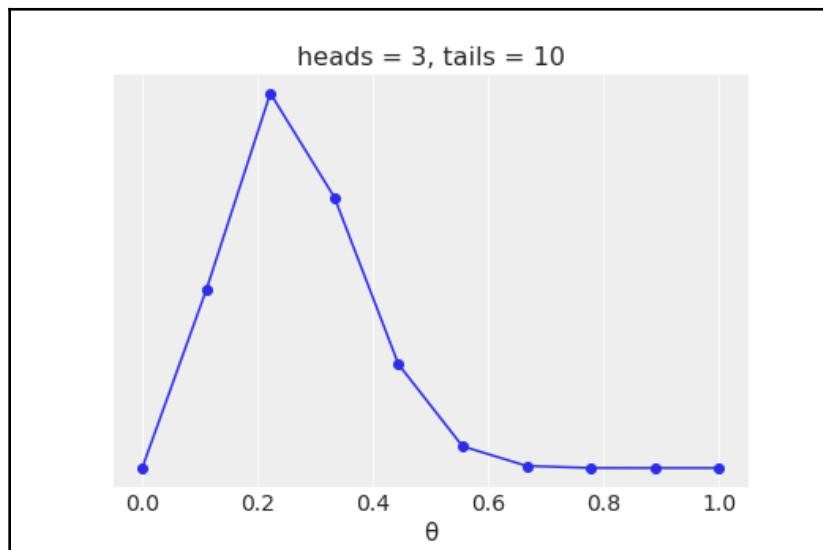


Figure 8.1

It is easy to notice that a larger number of points (or equivalently: a reduced size of the grid) results in a better approximation. As a matter of fact, in the limit of an infinite number of points, we will get the exact posterior at the cost of increasing the computational resources.

The biggest caveat of the grid approach is that this method scales poorly with the number of parameters, also referred to as dimensions. We can see this with a simple example.

Suppose we want to sample a unit interval (see *Figure 8.2*) like in the coin-flipping problem, and we use four equidistant points. This means a resolution of 0.25 units. Now suppose we have a 2D problem (the square in *Figure 8.2*) and we want to use a grid with the same resolution, we will need 16 points, and then for a 3D problem we will need 64 (see the cube in *Figure 8.1*). In this example, we need 16 times as many resources to sample from a cube of side 1 than for a line of length 1 with a resolution of 0.25. If we decide we instead need a resolution of 0.1 units, we will have to sample 10 points for the line and 1,000 for the cube:

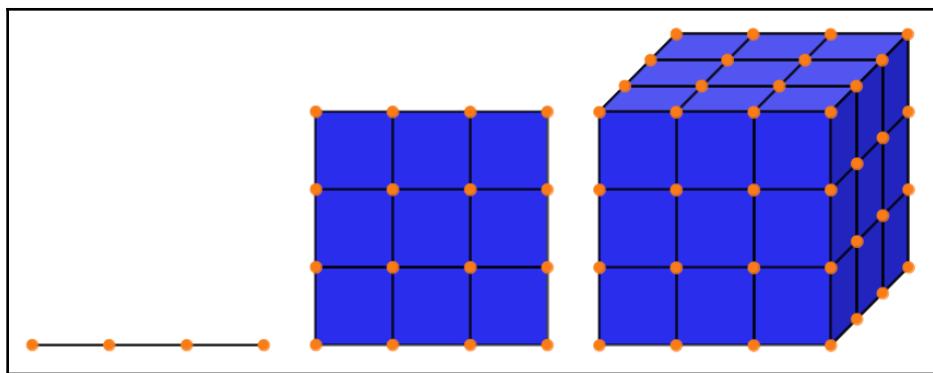


Figure 8.2

Besides *how the number of points increase*, there is also another phenomenon that is not a property of the grid method, or any other method for that matter, and instead is a property of high-dimensional spaces. As you increase the number of parameters, the region of the parameter-space where most of the posterior is concentrated gets smaller and smaller compared to the sampled volume. This is a pervasive phenomenon in statistics and machine learning and is usually known as the **curse of dimensionality**, or as mathematicians prefer to call it, the **concentration of measure**.

The curse of dimensionality is the name used to talk about various related phenomena that are absent in low-dimensional spaces but present in high-dimensional spaces. Here are some examples of these phenomena:

- As the number of dimensions increases, the euclidean distance between any pair of samples becomes closer and closer. That is, in high-dimensional spaces, most points are basically at the same distance from one another.
- For a hypercube, most of the volume is at its corners, *not in the middle*. For a hypersphere, most of the volume is at its surface and *not in the middle*.

- In high dimensions, most of the mass of a multivariate Gaussian distribution is not close to the mean (or mode), but in a *shell* around it that moves away from the mean to the tails as the dimensionality increases. This shell receives the name of *typical set*.

For code examples of some of these facts please check out <https://github.com/alocavodia/BAP>

For our current discussion, all these facts means that if we do not choose wisely where to evaluate the posterior, we will spend most of our time computing values with an almost null contribution to the posterior, and thus we will be wasting valuable resources. The grid method is not a very smart method to choose where to evaluate the posterior distribution, thus making it not very useful as a general method for high-dimensional problems.

Quadratic method

The quadratic approximation, also known as the *Laplace method* or the *normal approximation*, consists of approximating the posterior, $p(x)$, with a Gaussian distribution, $q(x)$.

This method consists of two steps:

1. Find the mode of the posterior distribution. This will be the mean of $q(x)$.
2. Compute the Hessian matrix. From this, we can compute the standard deviation of $q(x)$.

The first step can be done numerically using optimization methods, that is, methods to find the maximum or minimum of a function. There are many off-the-shelf methods for this purpose. Because for a Gaussian the mode and the mean are equal, we can use the mode as the mean of the approximating distribution, $q(x)$. The second step is not that *transparent*. We can approximately compute the standard deviation of $q(x)$ by evaluating the curvature at the mode/mean of $q(x)$. This can be done by computing the inverse of the square-root of the Hessian matrix. A Hessian matrix is the matrix of the second derivative of a function and its inverse provides the covariance matrix. Using PyMC3, we can do the following:

```
with pm.Model() as normal_approximation:
    p = pm.Beta('p', 1., 1.)
    w = pm.Binomial('w', n=1, p=p, observed=data)
    mean_q = pm.find_MAP()
    std_q = ((1/pm.find_hessian(mean_q, vars=[p]))**0.5)[0]
    mean_q['p'], std_q
```



If you try using the `pm.find_MAP` function in PyMC3, you will get a warning message. Because of the *curse of dimensionality*, using the **maximum a posteriori (MAP)** to represent the posterior or even to initialize a sampling method is not generally a good idea.

Let's see how the quadratic approximation looks for the beta-binomial model:

```
# analytic calculation
x = np.linspace(0, 1, 100)
plt.plot(x, stats.beta.pdf(x , h+1, t+1),
          label='True posterior')

# quadratic approximation
plt.plot(x, stats.norm.pdf(x, mean_q['p'], std_q),label='Quadratic
           approximation')
plt.legend(loc=0, fontsize=13)

plt.title(f'heads = {h}, tails = {t}')
plt.xlabel('θ', fontsize=14)
plt.yticks([]);
```

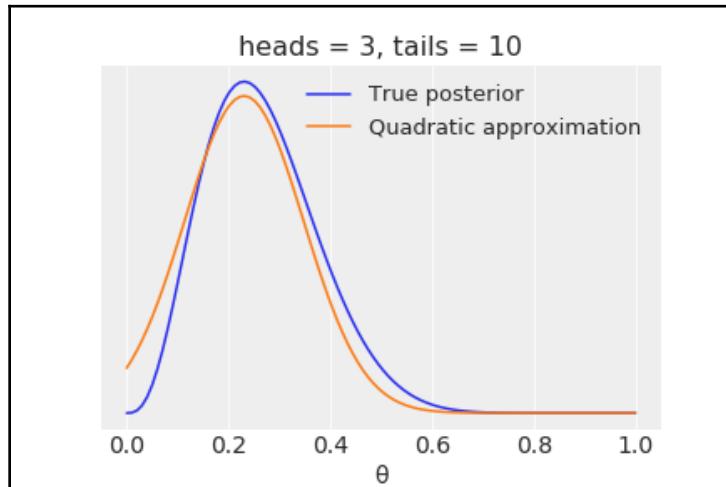


Figure 8.3

Figure 8.3 shows that the quadratic approximation is *not that bad*, at least for this example. Strictly speaking, we can only apply the Laplace method to unbounded variables, that is, variables living in \mathbb{R}^N . This is because the Gaussian is an unbounded distribution, so if we use it to model a bounded distribution (such as the beta distribution), we will end up estimating a positive density where in fact the density should be zero (outside the $[0, 1]$ interval for the beta distribution). Nevertheless, the Laplace method can be used if we first transform the bounded variable to make it unbounded. For example, we usually use a HalfNormal to model the standard deviation precisely because it's restricted to the $[0, \infty)$ interval, we can make a HalfNormal variable unbounded by taking the logarithm of it.

The Laplace method is limited, but can work well for some models and can be used to obtain analytical expressions to approximate the posterior. It's also one of the building blocks of a more advanced method known as **Integrated Nested Laplace Approximation (INLA)**.

In the next section, we will discuss variational methods that are somehow similar to the Laplace approximation but are more flexible and, powerful, and some of them can be applied automatically to a wide range of models.

Variational methods

Most of modern Bayesian statistics is done using Markovian methods (see the next section), but for some problems those methods can be too slow. Variational methods are an alternative that could be a better choice for large datasets (think big data) and/or for posteriors that are too expensive to compute.

The general idea of variational methods is to approximate the posterior distribution with a simpler distribution, in a similar fashion to the Laplace method but in a more elaborate way. We can find this simpler distribution by solving an optimization problem consisting of finding the closest possible distribution to the posterior under some way of measuring *closeness*. A common way of measuring *closeness* between distributions is by using the **Kullback-Leibler (KL)** divergence (as discussed in Chapter 5, *Model Comparison*). Using the KL divergence we can write:

$$D_{KL}(q(\theta) || p(\theta | y)) = \int q(\theta) \log \frac{q(\theta)}{p(\theta | y)} d(\theta) \quad (8.1)$$

Where q is the simpler distribution, we use to approximate the posterior, $p(\theta | y)$; q is often called the variational distribution, and by using an optimization method, we try to find out the parameters of q (often called the **variational parameters**) that makes q as close as possible, in terms of the KL divergence, to the posterior distribution. Notice that we wrote $D_{KL}(q(\theta) || p(\theta | y))$ and not $D_{KL}((\theta | y)) || q(\theta))$; we do so because this leads to a more convenient way of expressing the problem and a better solution, although I should make it clear that writing the KL divergence in the other direction can also be useful and in fact leads to another set of methods that we will not discuss here.

The problem with expression 8.1 is that we do not know the posterior so we can not directly use it. We need to find an alternative way to express our problem. The following steps show how to do that. If you do not care about the intermediate steps, please jump to equation 8.7.

First we replace the conditional distribution with its definition (see Chapter 1, *Thinking Probabilistically*, if you do not remember how to do this):

$$D_{KL}(q(\theta) || p(\theta | y)) = \int q(\theta) \log \frac{q(\theta)}{\frac{p(\theta, y)}{p(y)}} d(\theta) \quad (8.2)$$

Then we just reorder 8.2:

$$= \int q(\theta) \log \frac{q(\theta)}{p(\theta, y)} p(y) d(\theta) \quad (8.3)$$

By the properties of the logarithm, we have this equation:

$$= \int q(\theta) (\log \frac{q(\theta)}{p(\theta, y)} + \log p(y)) d(\theta) \quad (8.4)$$

Reordering:

$$= \int q(\theta) \log \frac{q(\theta)}{p(\theta, y)} d(\theta) + \int q(\theta) \log p(y) d(\theta) \quad (8.5)$$

The integral of $q(\theta)$ is 1 and we can move $\log p(y)$ out of the integral, then we get:

$$= \int q(\theta) \log \frac{q(\theta)}{p(\theta, y)} d(\theta) + \log p(y) \quad (8.6)$$

And using the properties of the logarithm:

$$D_{KL}(q(\theta) \parallel p(\theta \mid y)) = -\underbrace{\int q(\theta) \log \frac{p(\theta, y)}{q(\theta)} d(\theta)}_{\text{evidence lower bound (ELBO)}} + \log p(y) \quad (8.7)$$

Since $D_{KL} \geq 0$, then $\log p(y) \geq \text{ELBO}$, or in other words, the evidence (or marginal likelihood) is always equal or larger than the ELBO, and that is the reason for its name. Since $\log p(y)$ is a constant, we can just focus on the ELBO. Maximizing the value of the ELBO is equivalent to minimizing the KL divergence. Thus, maximizing the ELBO is a way to make $q(\theta)$ as close as possible to the posterior, $p(\theta \mid y)$.

Notice that, so far, we have not introduced any approximation, we just have been doing some algebra. The approximation is introduced the moment we choose $q(\cdot)$. In principle, $q(\cdot)$ can be anything we want, but in practice we should choose distributions that are easy to deal with. One solution is to assume that the high-dimensional posterior can be described by independent one-dimensional distributions; mathematically, this can be expressed as follows:

$$q(\theta) = \prod_j q_j(\theta_j) \quad (8.8)$$

This is known as the mean-field approximation. Mean-field approximations are common in physics, where it is used to model complex systems with many interacting parts as a collection of simpler subsystems not interacting at all, or in general, where the interactions have been taken into account only on average.

We could choose a different distribution, q_j , for each parameter, θ_j . Generally, the q_j distributions are taken from the exponential family because they are easy to deal with. The exponential family includes many of the distributions we have used in this book, such as Normal, exponential, beta, Dirichlet, gamma, Poisson, categorical, and Bernoulli.

With all these elements in place, we have effectively turned an inference problem into an optimization problem; thus, at least conceptually, all we need to solve it is to use some off-the-shelf optimizer methods and maximize the ELBO. In practice, things are a little bit more complex, but we have covered the general idea.

Automatic differentiation variational inference

The main drawback of the mean-field variational method we have just described is that we must come up with a specific algorithm for each model. We do not have a recipe for a universal inference engine, but instead the recipe to generate model-specific methods that require user intervention. Fortunately for us, many people have noticed this problem and have proposed solutions focused on the automatization of variational methods. A recently proposed method is the **Automatic Differentiation Variational Inference (ADVI)** (see <http://arxiv.org/abs/1603.00788>). At a conceptual level, the main steps that ADVI takes are:

- Transforms all bounded distributions to make them live on the real line, as we discussed for the Laplace method.
- Approximates the unbounded parameters with a Gaussian distribution (this is our q_j in equation 8.8); notice that a Gaussian on the transformed parameter space is non-Gaussian on the original parameter space.
- Uses automatic differentiation to maximize the ELBO.

The PyMC3 documentation (https://docs.pymc.io/nb_examples) provides many examples of how to use variational inference with PyMC3.

Markovian methods

There is a family of related methods, collectively known as *MCMC methods*. These stochastic methods allow us to get samples from the true posterior distribution as long as we are able to compute the likelihood and the prior point-wise. While this is the same condition that we need for the grid-approach, MCMC methods outperform the grid approximation. This is because MCMC methods are capable of taking more samples from higher-probability regions than lower ones. In fact, an MCMC method will *visit* each region of the parameter-space in accordance to their relative probabilities. If region **A** is twice as likely as region **B**, then we are going to get twice as many samples from **A** as from **B**. Hence, even if we are not capable of computing the whole posterior analytically, we could use MCMC methods to take samples from it.

At the most fundamental level, basically everything we care about in statistics is about computing expectations like:

$$\mathbb{E}[f] = \int_{\theta} p(\theta) f(\theta) d\theta \quad (8.9)$$

Here are some particular examples of this general expression:

- The posterior, equation 1.14
- The posterior predictive distribution, equation 1.17
- The marginal likelihood given a model, equation 5.13

With MCMC methods, we approximate equation 8.9 using finite samples:

$$\lim_{N \rightarrow \infty} \mathbb{E}_\pi[f] = \frac{1}{N} \sum_{n=1}^N f(\theta_n) \quad (8.10)$$

The big catch with equation 8.10 is that the equality only holds *asymptotically*, that is, for an infinite number of samples! In practice, we always have a finite number of samples, thus we want the MCMC methods to *converge* to the right answer as quickly as possible—with the least possible number samples (also known as **draws**).

In general, being sure that a particular sample from an MCMC has converged is not easy, to put it mildly. Thus, in practice, we must rely on empirical tests to make sure we have a reliable MCMC approximation. We will discuss such tests for MCMC samples in the *Diagnosing samples* section. It is important to keep in mind that other approximations (including the non-Markovian methods discussed in this chapter) also need empirical tests, but we will not discuss them as the focus of this book is MCMC methods.

Having a conceptual understanding of MCMC methods can help us to diagnose samples from them. So, let me ask, what's in a name? Well, sometimes not much, sometimes a lot. To understand what MCMC methods are, we are going to split the method into the *two MC parts*; the *Monte Carlo* part and the *Markov Chain* part.

Monte Carlo

The use of random numbers explains the Monte Carlo part of the name. Monte Carlo methods are a very broad family of algorithms that use random sampling to compute or simulate a given process. Monte Carlo is a ward in the Principality of Monaco where a very famous casino is located. One of the developers of the Monte Carlo method, Stanislaw Ulam, had an uncle who used to gamble there. The key idea *Stan* had was that while many problems are difficult to solve or even formulate in an exact way, they can be effectively studied by taking samples from them. In fact, as the story goes, the motivation was to answer questions about the probability of getting a particular hand in a game of Solitaire. One way to solve this problem is to follow the analytical combinatorial problem. Another way, *Stan* argued, is to play several games of Solitaire and count how many of the hands we play match the particular hand we are interested in! Maybe this sounds obvious to you, or at least pretty reasonable; you may even have used re-sampling methods to solve statistical problems. But, remember this, mental experiment was performed about 70 years ago, a time when the first practical computers began to be developed!

The first application of the Monte Carlo method was to solve a problem of nuclear physics, a hard-to-tackle problem using the tools at that time. Nowadays, even personal computers are powerful enough to solve many interesting problems using the Monte Carlo approach; hence, these methods are applied to a wide variety of problems in science, engineering, industry, and the arts. A classic pedagogical example of using a Monte Carlo method to compute a quantity of interest is the numerical estimation of the number π . In practice, there are better methods for this particular computation, but its pedagogical value still remains.

We can estimate the value of π with the following procedure:

1. Throw N points at random into a square of side $2R$.
2. Draw a circle of radius R inscribed in the square and count the number of points that are inside that circle.
3. Estimate $\hat{\pi}$ as the ratio, $4 \frac{\text{inside}}{N}$.

Here are a few notes:

- The area of the circle and square are proportional to the number of points inside the circle and the total N points, respectively.
- We know a point is inside a circle if the following relation holds, $\sqrt{(x^2 + y^2)} \leq R$.
- The area of the square is $(2R)^2$ and the area of the circle is πR^2 . Thus we know that the ratio of the area of the square to the area of the circle is π .

Using a few lines of Python, we can run this simple Monte Carlo simulation and compute π , and also the relative error of our estimate compared to the true value of π :

```
N = 10000

x, y = np.random.uniform(-1, 1, size=(2, N))
inside = (x**2 + y**2) <= 1
pi = inside.sum()*4/N
error = abs((pi - np.pi) / pi) * 100

outside = np.invert(inside)

plt.figure(figsize=(8, 8))
plt.plot(x[inside], y[inside], 'b.')
plt.plot(x[outside], y[outside], 'r.')
plt.plot(0, 0, label=f'\pi*= {pi:4.3f}\nerror = {error:4.3f}', alpha=0)
plt.axis('square')
plt.xticks([])
plt.yticks([])
plt.legend(loc=1, frameon=True, framealpha=0.9);
```

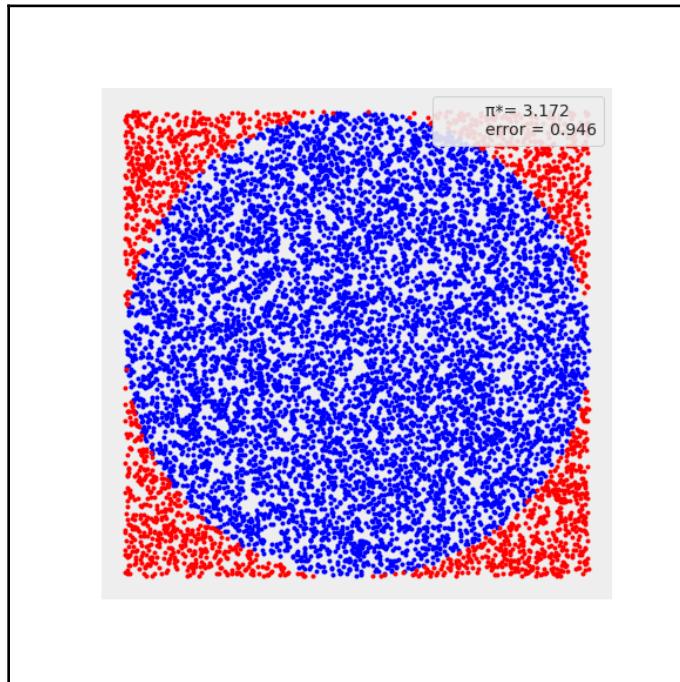


Figure 8.4

In the preceding code, we can see that the `outside` variable is only used to get the plot; we do not need it to compute $\hat{\pi}$. Also notice that because our computation is restricted to the unit circle; we can omit computing the square root when computing the `inside` variable.

Markov chain

A **Markov chain** is a mathematical object that consists of a sequence of states and a set of transition probabilities that describe how to move among the states. A chain is Markovian if the probability of moving to any other state depends only on the current state. Given such a chain, we can perform a *random walk* by choosing a starting point and moving to other states according to the transition probabilities. If we somehow find a Markov chain with transitions proportional to the distribution we want to sample from (the posterior distribution in Bayesian analysis), sampling simply becomes a matter of moving between states in this chain.

So, how do we find this chain if we do not know the posterior in the first place? Well, there is something known as a **detailed balance condition**. Intuitively, this condition says that we should move in a reversible way (a reversible process is a common approximation in physics). That is, the probability of being in state i and moving to state j should be the same as the probability of being in state j and moving toward state i . This condition is not really necessary, but it is sufficient and generally easier to prove, so it is generally used as a guide to design the majority of the most popular MCMC methods.

In summary, if we manage to create a Markov Chain that satisfies this detailed balance, we can sample from that chain with the guarantee that we will get samples from the correct distribution. This is a truly remarkable result! And the basic engine under the hood of software like PyMC3.

The most popular Markov Chain Monte Carlo method is probably the Metropolis-Hastings algorithm, and we will discuss it in the following section.

Metropolis-Hastings

For some distributions, such as the Gaussian, we have very efficient algorithms to get samples from, but for other distributions, this is not the case. Metropolis-Hastings enables us to obtain samples from any probability distribution, $p(x)$, given that we can compute at least a value proportional to it, thus ignoring the normalization factor. This is very useful since in a lot of problems, not just Bayesian statistics, the hard part is to compute the normalization factor.

To conceptually understand this method, we are going to use the following analogy. Suppose we are interested in finding the volume of water a lake contains and which part of the lake has the deepest point. The water is really muddy so we can't estimate the depth just by looking through the water to the bottom, and the lake is really big, so a grid approximation does not seem like a very good idea. To develop a sampling strategy, we seek help from two of our best friends; Markovia and Monty. After a fruitful discussion, they come up with the following algorithm, which requires a boat—nothing fancy, we can even use a wooden raft, and a very long stick. This is cheaper than SONAR and we have already spent all our money on the boat, anyway! Check out these steps:

1. Choose a random place in the lake, and move the boat there.
2. Use the stick to measure the depth of the lake.
3. Move the boat to another point and take a new measurement.
4. Compare the two measures in the following way:
 - If the new spot is deeper than the first one, write down in your notebook the depth of the new spot and repeat from *step 2*.
 - If the spot is shallower than the first one, we have two options: to accept or reject. Accepting means we write down the depth of the new spot and repeat from *step 2*. Rejecting means we go back to the first spot and write down (yes again!) the value for the depth of the first spot.

The rule to decide whether to accept or reject is known as the Metropolis-Hastings criterion, and it basically says that we must accept the new spot with a probability that is proportional to the ratio of the depth of the new and old spots.

If we follow this iterative procedure, we will get not only the total volume of the lake and the deepest point, we will also get an approximation of the entire curvature of the bottom of the lake. As you may have guessed, in this analogy, the curvature of the bottom of the lake is the posterior distribution and the deepest point is the mode. According to our friend Markovia, the larger the number of iterations, the better the approximation. Indeed, the theory guarantees that under certain general circumstances, we are going to get the exact answer if we get an infinite number of samples. Luckily for us, in practice, and for many, many problems, we can get a very accurate approximation using a finite and relatively small number of samples.

The preceding explanation is enough to get a conceptual-level understanding of Metropolis-Hastings. The next few paragraphs contain a more detailed and formal explanation in case you want to dig deeper.

The Metropolis-Hastings algorithm has the following steps:

1. Choose an initial value for a parameter, x_i . This can be done randomly or by making an educated guess.
2. Choose a new parameter value, x_{i+1} , by sampling from an easy-to-sample distribution, such as a Gaussian or uniform distribution, $q(x_{i+1} | x_i)$. We can think of this step as perturbing the x_i state somehow.
3. Compute the probability of accepting a new parameter value by using the Metropolis-Hastings criterion:

$$p_a(x_{i+1} | x_i) = \min \left(1, \frac{p(x_{i+1})q(x_i | x_{i+1})}{p(x_i)q(x_{i+1} | x_i)} \right) \quad (8.11)$$

4. If the probability computed on step 3 is larger than the value taken from a uniform distribution on the $[0, 1]$ interval, we accept the new state; otherwise, we stay in the old state.
5. We iterate from *step 2* until we have *enough* samples.

Here are a couple of notes to take into account:

- If the proposal distribution $q(x_{i+1} | x_i)$ is symmetric, we get the Metropolis criterion (notice we drop the Hastings part):

$$p_a(x_{i+1} | x_i) = \min \left(1, \frac{p(x_{i+1})}{p(x_i)} \right) \quad (8.12)$$

- *Step 3* and *step 4* imply that we will always accept moving to the most probable state. Less probable parameter values are accepted probabilistically, given the ratio between the probability of the new parameter value, x_{i+1} , and the old parameter value, x_i . This criterion for accepting proposed steps gives us a more efficient sampling approach compared to the grid approximation, while ensuring a correct sampling.
- The target distribution (the posterior distribution in Bayesian statistics) is approximated by a list of sampled parameter values. If we accept, we add to the list the new sampled value, x_{i+1} . If we reject, we add to the list the value of x_i (even if the value is repeated).

At the end of the process, we will have a list of values. If everything was done the right way, these samples will be an approximation of the posterior. The most frequent values in our trace will be the most probable values according to the posterior. An advantage of this procedure is that analyzing the posterior is as simple manipulating an array of values, as you have already experimented in all the previous chapters.

The following code illustrates a very basic implementation of the Metropolis algorithm. It is not meant to solve any real problem, only to show it is possible to sample from a probability distribution if we know how to compute its density point-wise. Notice also that the following implementation has nothing Bayesian in it; there is no prior and we do not even have data! Remember that MCMC methods are very general algorithms that can be applied to a broad array of problems.

The first argument of the metropolis function is a SciPy distribution; we are assuming we do not know how to directly get samples from this distribution:

```
def metropolis(func, draws=10000):
    """A very simple Metropolis implementation"""
    trace = np.zeros(draws)
    old_x = 0.5 # func.mean()
    old_prob = func.pdf(old_x)

    delta = np.random.normal(0, 0.5, draws)
    for i in range(draws):
        new_x = old_x + delta[i]
        new_prob = func.pdf(new_x)
        acceptance = new_prob / old_prob
        if acceptance >= np.random.random():
            trace[i] = new_x
            old_x = new_x
            old_prob = new_prob
        else:
            trace[i] = old_x
    return trace
```

In the next example, we have defined `func` as a beta function, simply because it's easy to change their parameters and get different shapes. We are plotting the samples obtained by `metropolis` as a histogram and also the true distribution as the continuous (orange) line:

```
np.random.seed(3)
func = stats.beta(2, 5)
trace = metropolis(func=func)
x = np.linspace(0.01, .99, 100)
y = func.pdf(x)
plt.xlim(0, 1)
plt.plot(x, y, 'C1-', lw=3, label='True distribution')
```

```
plt.hist(trace[trace > 0], bins=25, density=True, label='Estimated distribution')
plt.xlabel('x')
plt.ylabel('pdf(x)')
plt.yticks([])
plt.legend();
```

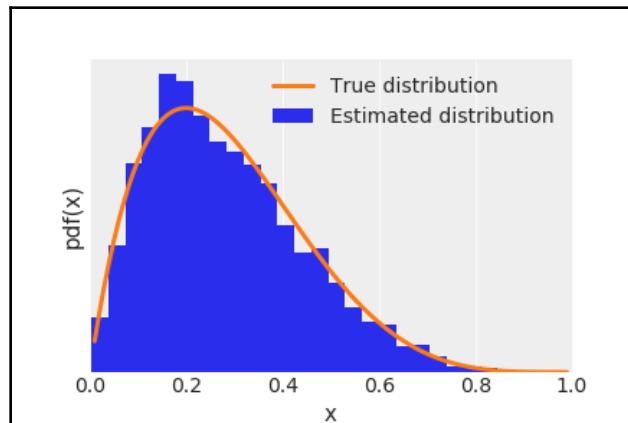


Figure 8.5

The efficiency of the algorithm depends heavily on the proposal distribution; if the proposed state is very far away from the current state, the chance of rejecting is very high, and if the proposed state is very close, we explore the parameter space very slowly. In both scenarios, we will need many more samples than for a less extreme situation. Usually, the proposal is a multivariate Gaussian distribution whose covariance matrix is determined during the tuning phase. PyMC3 tunes the covariance adaptively by following the rule of thumb that the ideal acceptance is between 50% for a unidimensional Gaussian and around 23% for an n-dimensional Gaussian target distribution.

MCMC methods often take some time before they start getting samples from the target distribution. So, in practice, people perform a burn-in step, which consists of eliminating the first portion of the samples. Doing a burn-in is a practical trick and not part of Markovian theory; in fact, it will not be necessary for an infinite sample. Thus, removing the first portion of the samples is just an *ad hoc* trick to get better results, given that we can only compute a finite sample. Having theoretical guarantees or guidance is better than not having them, but for any practical problem, it is important to understand the difference between theory and practice. Remember, we should not get confused by mixing mathematical objects with the approximation of those objects. Spheres, Gaussians, Markov chains, and all the mathematical objects live only in the platonic world of ideas not in our *imperfect*, real world.

At this point I hope you have a good conceptual grasp of the Metropolis-Hastings method. You may need to go back and read this section one or more times; that's totally fine. The main ideas are simple but also subtle.

Hamiltonian Monte Carlo

MCMC methods, including Metropolis-Hastings, come with the theoretical guarantee that if we take enough samples, we will get an accurate approximation of the correct distribution. However, in practice, it could take more time than we have to get enough samples. For that reason, alternatives to the general Metropolis-Hastings algorithm have been proposed.

Many of those alternative methods, such as the Metropolis-Hastings algorithm itself, were developed originally to solve problems in statistical mechanics, a branch of physics that studies properties of atomic and molecular systems, and thus can be interpreted in a very *natural* way using analogies of physical systems. One such modification is known as **Hamiltonian Monte Carlo**, or **Hybrid Monte Carlo (HMC)**. In simple terms, a Hamiltonian is a description of the total energy of a physical system. The name Hybrid is also used because it was originally conceived as a hybridization of molecular mechanics, a widely-used simulation technique for molecular systems, and Metropolis-Hastings.

The HMC method is essentially the same as Metropolis-Hastings except, and this is a very, very important, *except* the proposal of new *positions* is not random. To get a general conceptual understanding of HMC without going into mathematical details, let's use the lake and boat analogy again. Instead of moving the boat randomly, we do so by following the curvature of the bottom of the lake. To decide where to move the boat, we let a ball roll on to the bottom of the lake starting from our current position. Our ball is a very special one: not only is perfectly spherical, it also has no friction and thus is not slowed down by the water or mud. We throw the ball and let it roll for a short moment until we suddenly stop it. Then we accept or reject this proposed step using the Metropolis criterion, just as we did with the *vanilla* Metropolis-Hastings method, and the whole procedure is repeated *many* times. This modified procedure has a higher chance of accepting new positions, even if they are far away relative to the previous position.

Moving according to the curvature of the parameter space turns out to be a smarter way of moving because it avoids one of the main drawbacks of Metropolis-Hastings: an efficient exploration of the sample space requires rejecting most of the proposed steps. Instead, using HMC, it is possible to get a high acceptance rate even for faraway points in the parameter space, thus resulting in a very efficient sampling method.

Let's get out of our Gedanken experiment and back to the real world. We have to pay a price for this very clever Hamiltonian-based proposal. We need to compute gradients of our function. A **gradient** is the generalization of the concept of the derivative to more than one dimension; computing the derivative of a function at one point tells us in which direction the function increases and in which direction it decreases. We can use gradient information to simulate the ball moving in a curved space; in fact, we use the same motion laws and mathematics used in classical physics to simulate classical mechanical systems, such as balls rolling, the orbits in planetary systems, and the jiggling of molecules.

Computing gradients make us face a trade-off; each HMC step is more expensive to compute than a Metropolis-Hastings step, but the probability of accepting that step is much higher with HMC than with Metropolis. To balance this trade-off situation in favor of HMC, we need to tune a few parameters of the HMC model (in a similar fashion to how tune the *width* of the proposal distribution for an efficient Metropolis-Hastings sampler). When this tuning is done by hand, it takes some trial and error and also requires an experienced user, making this procedure a less universal inference engine than we may want. Luckily for us, PyMC3 comes with a relatively new sampler, known as **No-U-Turn Sampler (NUTS)**. This method has proven very useful in providing a very good efficiency for solving Bayesian models without requiring human intervention (or at least minimizing it). One caveat of NUTS is that only works for continuous distribution; the reason is that we cannot compute gradients for discrete distribution. PyMC3 solves this problem by assigning NUTS to continuous parameters and Metropolis to the discrete ones.

I strongly recommend you complement this section with this very cool animation by Chi Feng: <https://chi-feng.github.io/mcmc-demo/>.

Sequential Monte Carlo

One of the caveats of Metropolis-Hastings and also NUTS (and other Hamiltonian Monte Carlo variants) is that if the posterior has multiple peaks and these peaks are separated by regions of *very low* probability, these methods can get stuck in a single mode and miss the others!

Many of the methods developed to overcome this multiple minima problem are based on the idea of *tempering*. This idea, once again, is borrowed from statistical mechanics. The number of states a physical system can populate depends on the temperature of the system; at 0 Kelvin (the lowest possible temperature), every system is stuck in a single state. On the other extreme, for an infinite temperature all possible states are equally likely. Generally, we are interested in systems at some intermediate temperature. For Bayesian models, there is a very intuitive way to adapt this tempering idea by writing Bayes' theorem with a twist.

$$p(\theta | y)_\beta = p(y | \theta)^\beta p(\theta) \quad (8.13)$$

The only difference between expressions 1.4 and 8.13 is the specification of the β parameter, this is known as the *inverse temperature* or tempering parameter. Notice that for $\beta = 0$ we get $p(y | \theta)^\beta = 1$ and thus the *tempered* posterior, $p(\theta | y)_\beta$ is just the prior, $p(\theta)$, and when $\beta = 1$, the *tempered* posterior is the actual full posterior. As sampling from the prior is generally easier than sampling from the posterior (by increasing the value of β), we start sampling from an easier distribution and slowly *morph it* into the more complex distribution we really care about.

There are many methods that exploit this idea; one of them is known as **Sequential Monte Carlo (SMC)**. The SMC method, as implemented in PyMC3, can be summarized as follows:

1. Initialize β at zero.
2. Generate N samples S_β from the tempered posterior.
3. Increase β a *little bit*.
4. Compute a set of N weights W . The weights are computed according to the new tempered posterior.
5. Obtain S_w by resampling S_β according to W .
6. Run N Metropolis chains, starting each one from a different sample in S_w .
7. Repeat from step 3 until $\beta \geq 1$.

The resampling step works by removing samples with a low probability and replacing them with samples with a higher probability. The Metropolis step perturbs these samples helping, to explore the parameter-space.

The efficiency of the tempered method depends heavily on the intermediate values of β , what is usually referred as the *cooling* schedule. The smaller the difference between two successive values of β , the closer the two successive tempered posteriors will be, and thus the easier the transition from one stage to the next. But if the steps are too small, we will need many intermediate stages, and beyond some point this will translate into wasting a lot of computational resources without really improving the accuracy of the results.

Fortunately, SMC can automatically compute the intermediate values of β . The exact *cooling* schedule will be adapted to the difficulty of the problem; distributions that are more difficult to sample will require more stages than simpler ones.

SMC is summarized in *Figure 8.6*, the first subplot shows five samples (orange) dots at some particular stage. The second subplots show how these samples are re-weighted according to their tempered posterior density (blue) curve. The third subplot shows the result of running a certain number of Metropolis steps, starting from the re-weighted samples in the second subplot; notice how the two samples with the lower posterior density (the smaller circles at the most right and left) are discarded and not used to seed new Markov chains:

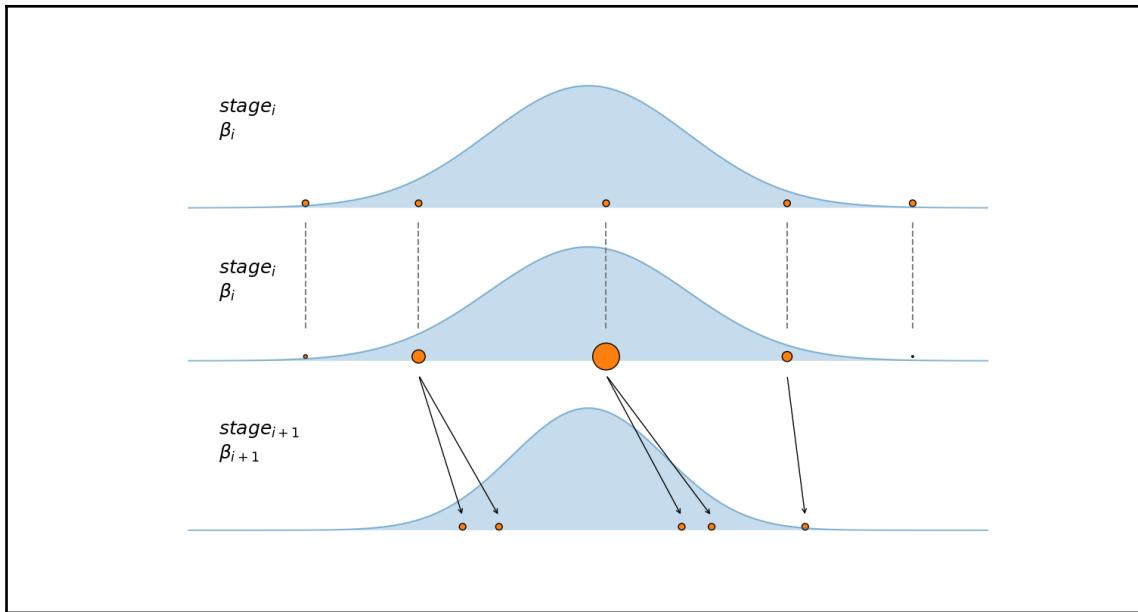


Figure 8.6

Besides the intermediate values of β , two more parameters are dynamically computed based on the acceptance rate of the previous stage: the number of steps of each Markov chain and the width of the proposal distribution.

For step 6 of the SMC algorithm, PyMC3 uses the Metropolis algorithm; this is not necessarily the only choice but is a very reasonable one and is motivated both by theoretical and practical arguments. It is important to note than even when the SMC method uses the Metropolis method, it has several advantages over it:

- It can sample from multimodal distributions.
- It does not have a burn-in period. This is due to the re-weighing step. The weights are computed in such a way that S_w approximates not $p(\theta | y)_{\beta_i}$ but $p(\theta | y)_{\beta_{i+1}}$, thus at each stage the MCMC chains start, approximately, from the correct tempered posterior distribution.
- It can generate samples with low autocorrelation.
- It can be used to approximate the marginal likelihood (see [Chapter 5, Model Comparison](#)), this is just a side-effect of the SMC method requiring almost no additional computations.

Diagnosing the samples

This section is focused on diagnostic samples for Metropolis and NUTS. Since we are approximating the posterior with a finite number of samples, is important to check whether we have a valid sample—otherwise any analysis from it will be totally flawed. There are several tests we can perform, some are visual and some are quantitative. These tests are designed to spot problems with our samples, but they are unable to prove we have the correct distribution; they can only provide evidence that the sample seems reasonable. If we find problems with the sample, there are many solutions to try:

- Increase the number of samples.
- Remove a number of samples from the beginning of the trace. This is known as **burn-in**. The PyMC3 tuning phase helps reduce the need for burn-in.
- Modify sampler parameters, such as increasing the length of the tuning phase, or increase the `target_accept` parameter for the NUTS sampler. Under certain circumstances, PyMC3 will provide suggestions on what to change.
- Re-parametrize the model, that is, express the model in a different but equivalent way.
- Transform the data. We already saw an example of this, in [Chapter 4, Generalizing Linear Models](#), and [Chapter 5, Model Comparison](#), we show that centering the data improves sampling from linear models.

To make the explanations concrete, we are going to use a *minimalist* hierarchical model, with two parameters: a *global* parameter, a , and a local parameter, b (the per group parameter). And that's all, we do not even have likelihood/data in this model! I am omitting the data here to emphasize that some of the properties we will discuss (especially in the section *Divergences*) are related to the structure of the model and not the data. We will discuss two alternative parameterizations of the same model:

```
with pm.Model() as centered_model:  
    a = pm.HalfNormal('a', 10)  
    b = pm.Normal('b', 0, a, shape=10)  
    trace_cm = pm.sample(2000, random_seed=7)  
  
with pm.Model() as non_centered_model:  
    a = pm.HalfNormal('a', 10)  
  
    b_shift = pm.Normal('b_offset', mu=0, sd=1, shape=10)  
    b = pm.Deterministic('b', 0 + b_shift * a)  
    trace_ncm = pm.sample(2000, random_seed=7)
```

The difference between the **centered** and **non-centered** models is that for the former we fit the group-level parameter directly, and for the later we model the group-level parameter as a shifted and scaled Gaussian. We will explore the differences using several plots and numerical summaries.

Convergence

An MCMC sampler, such as NUTS or Metropolis, could take some time before it converges; that is, it starts sampling from the correct distribution. As we previously explained, MCMC methods come with theoretical guarantees of convergence under very general conditions and an infinite number of samples. Unfortunately, in practice, we only can get a finite sample, so we must rely instead on empirical tests that provide, at best, hints or warnings that something bad could be happening when they fail but do not guarantee everything is OK when they do not fail.

One way to visually check for convergence is to run the ArviZ `plot_trace` function and inspect the result. To better understand what we should look for when *inspecting* these plots, let's compare the results for the two previously defined models (see *Figure 8.6* and *8.7*):

```
az.plot_trace(trace_cm, var_names=['a'], divergences='top')
```

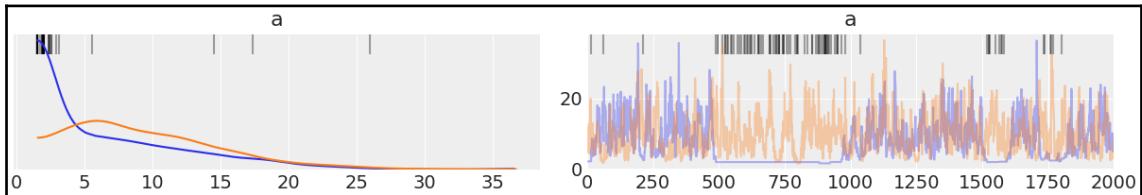


Figure 8.7

```
az.plot_trace(trace_ncm, var_names=['a'])
```

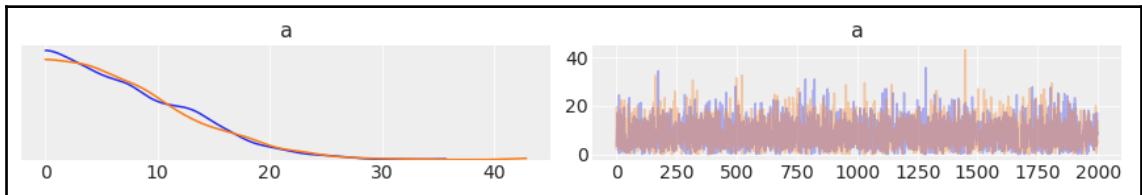


Figure 8.8

Notice how the KDE in *Figure 8.8* is smoother than in *8.7*; a smooth KDE is a good sign while an uneven KDE may indicate a problem, such as a need for more samples or a more serious problem. The *trace* itself (the plot on the right) should look like white noise, meaning we should not see any recognizable pattern; we want a curve *freely meandering around*, like the *trace* in *Figure 8.8*. When this happens, we say we have **good mixing**. Instead, *Figure 8.6* is an example of pathological behavior; if you carefully compare it to *Figure 8.8*, you will notice that the overlap of the two chains is larger for *8.8* than *8.7*, and you will also notice something fishy going on at several regions along the trace in *8.7*; the clearest one is in the region between 500-1000 draws: you will see that one of the chains (the blue one) got stuck (is basically a horizontal line).

This is really, really, really bad. The sampler is rejecting all new proposals except for those in the very close neighborhood; in other words, it is sampling very slowly and thus not very efficiently. For infinite samples, this will not be problematic, but for finite samples this introduce a bias in the result. A *simple fix* is to take more samples, but this will only help if the bias is somehow *small*, otherwise the number of samples required to compensate the bias will grow very very fast, making this *simple fix* useless.

Figure 8.9 has some additional examples of traces with good mixing (on the right) and bad mixing (on the left). If there is more than one region, such as for discrete variables or multimodal distributions, we expect the trace to not spend too much time in a value or region and then move to other regions, but to jump from one region to the other with ease:

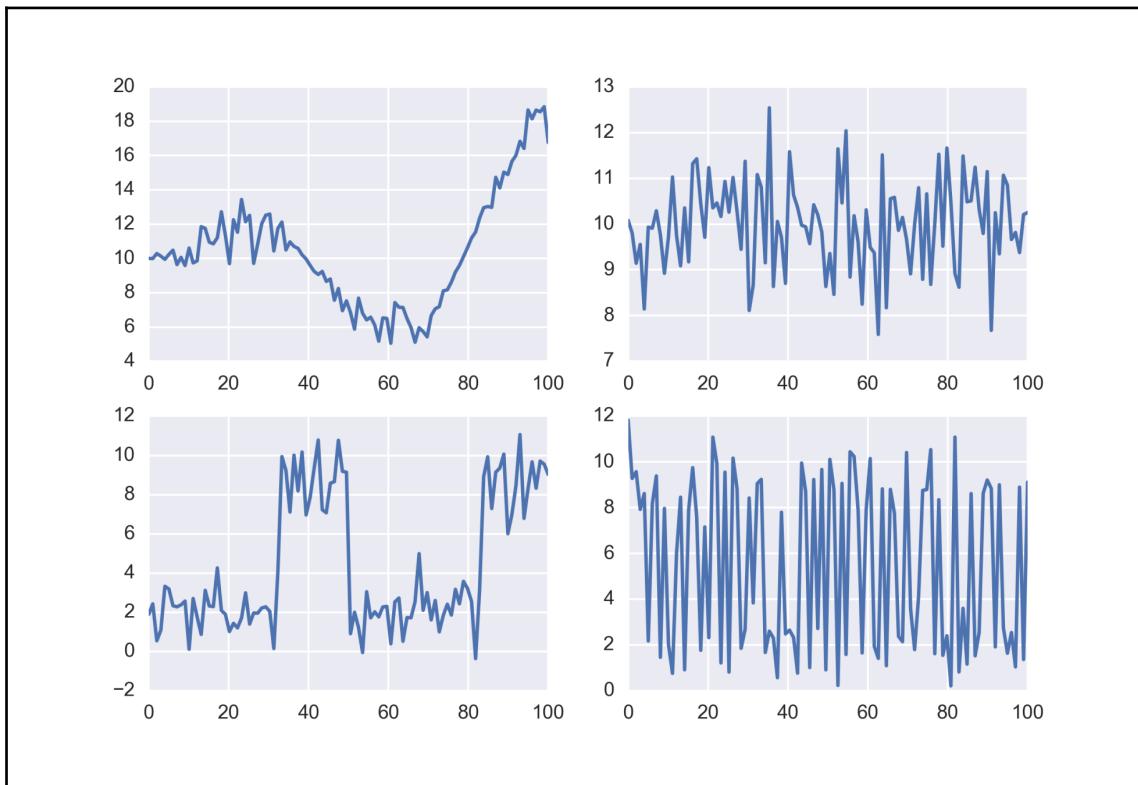


Figure 8.9

Another feature of a good MCMC sample is an auto-similar trace; for example, the first 10% (or so) should look similar to other portions in the trace, such as the last 50% or 10%. Once again, we do not want patterns; instead, we expect something noisy. This can also be seen using `az.plot_trace`. If the first part of the trace looks different than the others, this is an indication of the need for burn-in, or a higher number of samples. If we see a lack of auto-similarity in other parts or we see a pattern, this could mean we need more draws, but more often than not, we should try with a different parameterization. For difficult models, we may even need to apply a combination of all these strategies.

By default, PyMC3 will try to run independent chains in parallel (the exact number depends on the number of available processors). This is specified with the `chains` argument in the `pm.sample` function. We can use the `plot_trace` or `plot_forest` ArviZ functions to visually inspect whether the parallel chains are similar to each other. Then we can combine the parallel chains into a single one for inference, so notice that running chains in parallel is not a waste of resources.

A quantitative way of comparing independent chains is by using the Rhat statistic. The idea of this test is to compute the variance between chains with the variance within chains. Ideally, we should expect a value of 1. As an empirical rule, we will be OK with a value below 1.1; higher values signal a lack of convergence. We can compute it using the `az.r_hat` function; we just need to pass a PyMC3 trace object. The Rhat diagnostic is also computed by default with the `az.summary` function (as you may recall from previous chapters) and optionally with `az.plot_forest` (using the `r_hat=True` argument), as we can see in the following examples:

```
az.plot_forest(trace_cm, var_names=['a'], r_hat=True, eff_n=True)
```

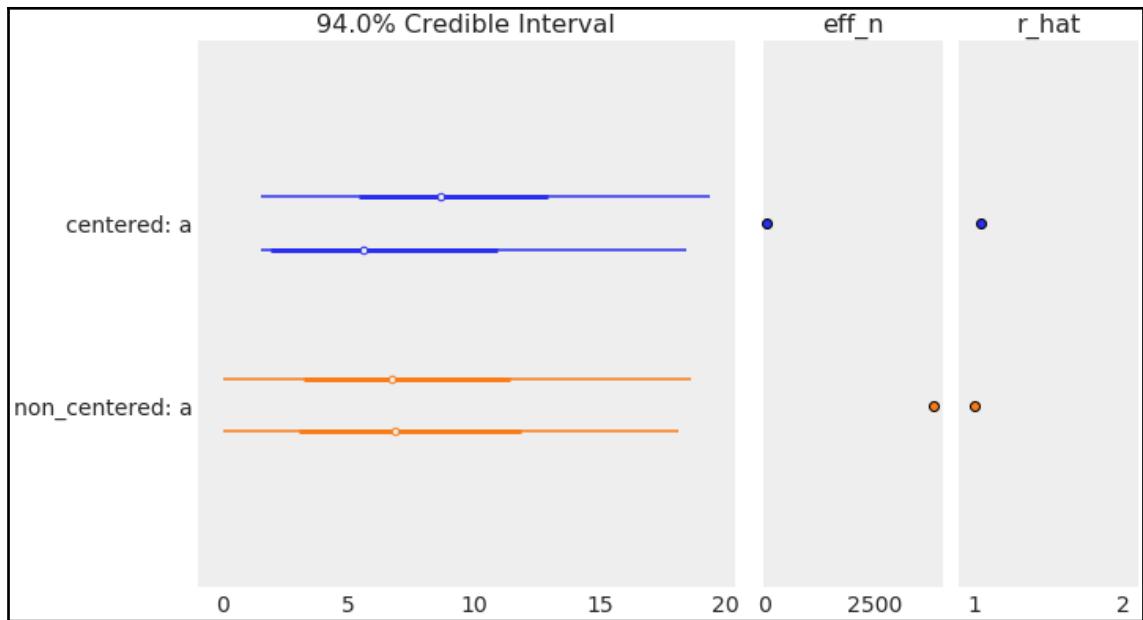


Figure 8.10

And also for `az.summary`:

```
summaries = pd.concat([az.summary(trace_cm, var_names=['a']),
                      az.summary(trace_ncm, var_names=['a'])])
summaries.index = ['centered', 'non_centered']
summaries
```

	mean	sd	mc error	hpdi 3%	hpdi 97%	eff_n	r_hat
centered	8.53	5.84	0.58	1.53	18.87	49.0	1.04
non_centered	7.92	6.01	0.04	0.01	18.48	3817.0	1.00

Monte Carlo error

One of the quantities returned by the summary is `mc_error`. This is an estimation of the error introduced by the sampling method. The estimation takes into account that the samples are not truly independent of each other. mc_{error} is the standard error of the mean's x of n blocks, and each block is just a portion of the trace:

$$mc_{error} = \frac{\sigma(x)}{\sqrt{n}} \quad (8.14)$$

This error should be below the precision we want in our results.

Autocorrelation

An ideal sample from a distribution, including the posterior, should have autocorrelation equal to zero (unless we expect a correlation like in time-series). A sample is autocorrelated when a value at a given iteration is not independent of the sampled values at other iterations. In practice, samples generated from MCMC methods will be autocorrelated, especially Metropolis-Hastings, and to a lesser extent NUTS and SMC. ArviZ comes with a convenient function to plot the autocorrelation:

```
az.plot_autocorr(trace_cm, var_names=['a'])
```

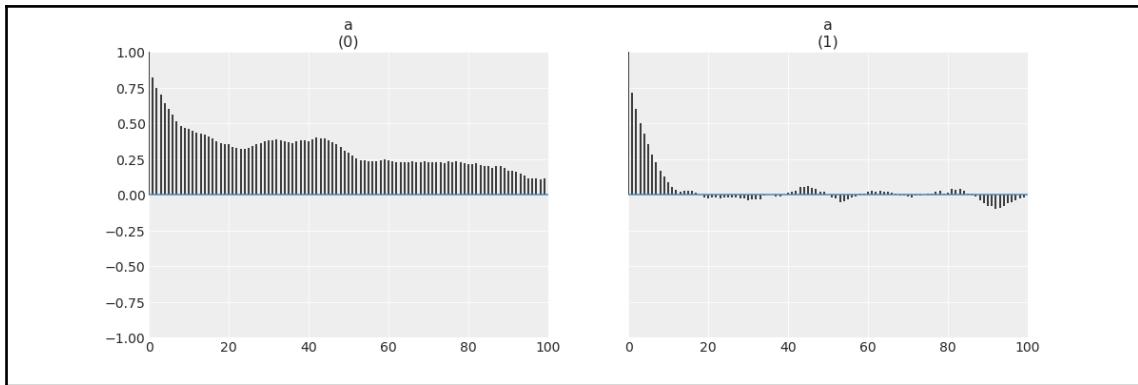


Figure 8.11

`az.plot_autocorr` shows the average correlation of sample values compared to successive points (up to 100 points). Ideally, we should see no autocorrelation. In practice, we want samples that quickly drop to low values of autocorrelation. Let's also plot the autocorrelation for the non-centered model:

```
az.plot_autocorr(trace_ncm, var_names=['a'])
```

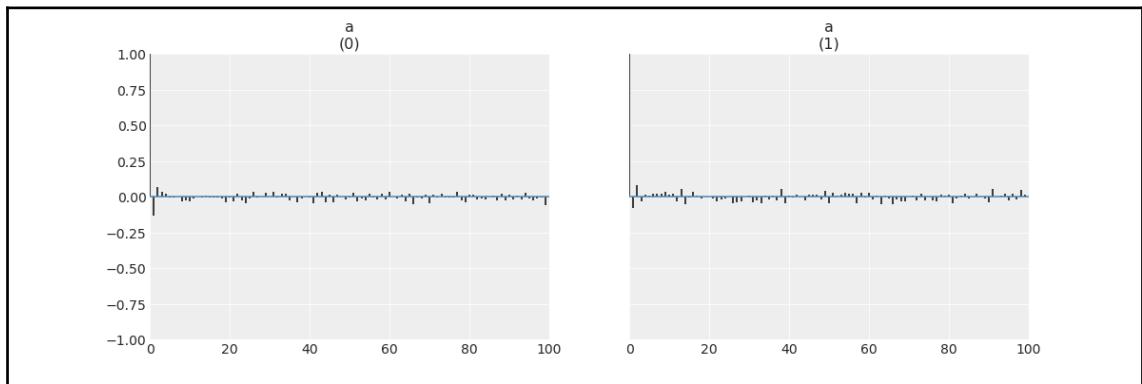


Figure 8.12

By comparing *Figures 8.11* and *8.12*, we can easily see that the samples from the non-centered model have almost no autocorrelation while the samples from the centered model show larger values of autocorrelation.

Effective sample sizes

A sample with autocorrelation has less information than a sample of the same size without autocorrelation. In fact, we can use the autocorrelation to estimate the size of a given sample with the equivalent information but without autocorrelation. This is called the **effective sample size**. The more autocorrelated a parameter is, the larger the number of samples we will need to obtain a given precision, in other words, autocorrelation has the detrimental effect of reducing the number of effective samples. We can compute the effective sample size with ArviZ using the `az.effective_n` function. The effective sample size is also computed by the `az.summary` (as we did a page ago and in previous chapters) and `az.plot_forest` functions, by passing the `eff_n=True` argument (see *Figure 8.9*).

Ideally, the effective sample size should be close to the actual sample size. One advantage of NUTS over Metropolis is that the effective sample size of NUTS is usually much higher than that of Metropolis, and thus in general if you use NUTS, you generally will need fewer samples than when using Metropolis.

PyMC3 will warn us if the effective sample size is lower than 200 for any parameter. As a general guide, 100 effective samples should provide a good estimate of the mean values of a distribution, but having more samples will provide estimates that change less every time we re-run a model, which is part of the reason you use a 200 cutoff for the effective sample size. For most problem values, 1,000 to 2,000 effective samples will be more than enough. If we want high precision on quantities that depend on the tails of distribution or events that are very rare, we will need a larger effective sample size.

Divergences

We will now explore tests that are exclusive of NUTS, as they are based on the inner-working of the method and not a property of the generated samples. These tests are based on something called **divergences** and are a powerful and sensitive method of diagnosing samples.

While I tried to set the models in this book to avoid divergences, you may have seen PyMC3 messages indicating that a divergence had occurred. Divergences may indicate that NUTS has encountered regions of high curvature in the posterior that it cannot explore properly; it is telling us that the sampler could be missing a region of the parameter space, and thus our results will be biased. Divergences are generally much more sensitive than the tests discussed here, and thus they can signal problems even when the rest of the tests pass. A nice feature of divergences is that they tend to appear close to the problematic parameter space regions, and thus we can use them to identify where the problem may be. One way to visualize divergences is by using `az.plot_pair` with the `divergences=True` argument:

```
_ , ax = plt.subplots(1, 2, sharey=True, figsize=(10, 5),
constrained_layout=True)

for idx, tr in enumerate([trace_cm, trace_ncm]):
    az.plot_pair(tr, var_names=['b', 'a'], coords={'b_dim_0': [0]}, kind='scatter',
                 divergences=True, contour=False,
                 divergences_kwarg={'color':'C1'},
                 ax=ax[idx])
    ax[idx].set_title(['centered', 'non-centered'][idx])
```

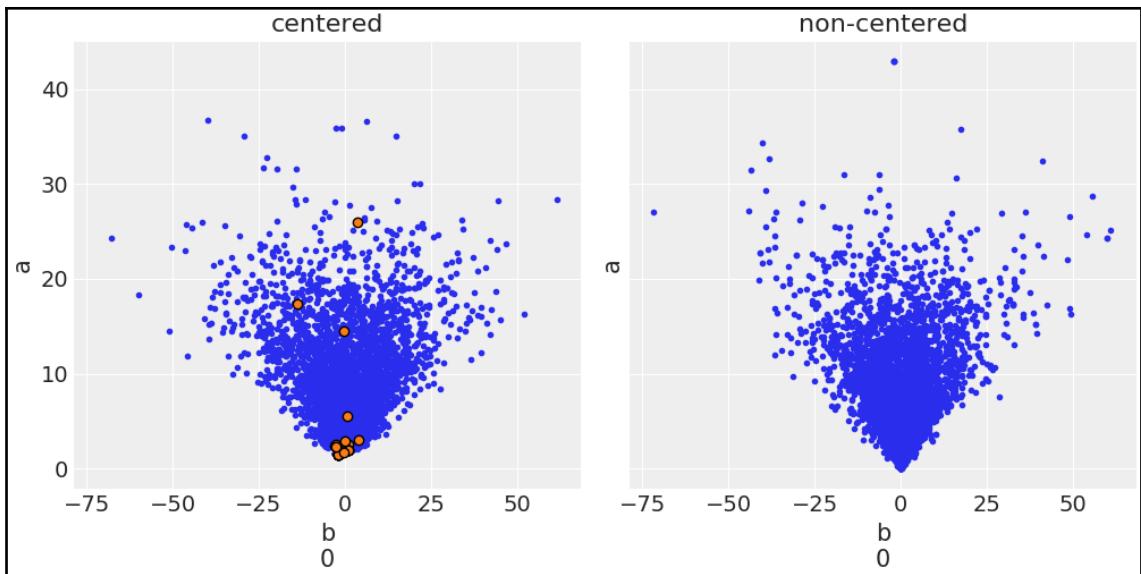


Figure 8.13

In *Figure 8.13*, the small (blue) dots are regular samples and the larger (black and orange) dots represent the divergences. We can see that the centered model has divergences that are mostly concentrated at the tip of the funnel. We can also see that the non-centered model has no divergences and a sharper tip. The sampler, through the divergences, is telling us that it is having a hard time sampling from the region close to the tip of the funnel. We can indeed check in *Figure 8.13* that the centered model does not have samples around the tip, close to where the divergences are concentrated. How neat is that?!

Divergences are also represented in ArviZ's trace plots using black "l" markers, see *Figure 8.7* for an example. Notice how divergences are concentrated around the pathological flat portions of the trace.

Another useful way to visualize divergences is with a parallel plot:

```
az.plot_parallel(trace_cm)
```

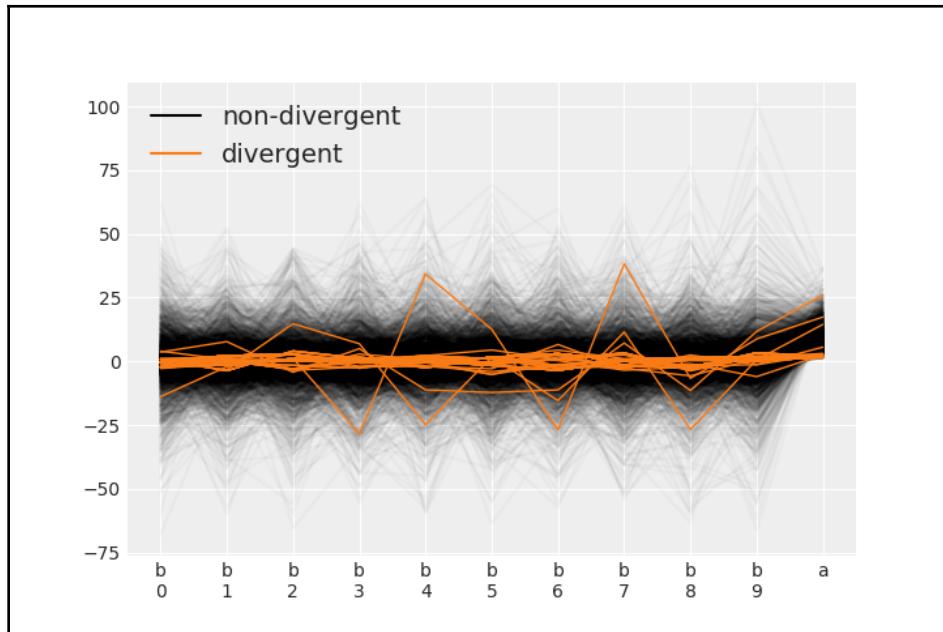


Figure 8.14

Here, we can see that divergences are concentrated around 0 for both b and a . Plots such as those in *Figure 8.13* and *8.14* are super important, because they let us know which part of the parameter space could be problematic, and also because they help us to spot false positives. Let me explain this last point. PyMC3 uses a heuristic to label divergences, sometimes, this heuristic could say we have a divergence when we do not. As a general rule, if divergences are scattered in the parameter space, we probably have false positives; if divergences are concentrated, then we likely have a problem. When getting divergences, there are three general ways to get rid of them, or at least reduce the number of them:

- Increase the number of tuning step, something like `pm.sample(tuning=1000)`.
- Increase the value of the `target_accept` parameter from its default value of 0.8. The maximum value is 1, so you can try with values such as 0.85 or 0.9.
- Re-parameterize the model. As we just saw, the non-centered model is a re-parameterization of the centered model, which leads to better samples and an absence of divergences.

Non-centered parameterization

I have presented the non-centered model as a magic trick that solves the sampling problem. Let's take a movement to undercover the trick and remove the magic.

From *Figure 8.13*, we can see that the a and b parameters are correlated. Since b is a vector of shape 10, we have chosen to plot $b(0)$, but any other element of b should show the same pattern, in fact this is rendered crystal clear in *Figure 8.14*. The correlation, and this particular funnel shape, are a consequence of the model definition and the ability of the model to partially pool data. As the values of a decrease, the individual values of b become closer to each other and closer to the global mean value. In other words, the shrinkage level gets higher and higher, and thus the data gets more and more pooled (up to the point of being completely pooled). The same structure that allows for partial pooling also introduces correlations that affect the performance of the sampler methods.

In Chapter 3, *Modeling with Linear Regression*, we saw that linear models also lead to correlation (of a different nature); for those models, an easy fix is to center the data. We may be tempted to do the same here, but unfortunately that will not help us to get rid of the sampling issues introduced by the funnel shape. The tricky feature of the funnel shape is that correlations vary with the position in parameter space, thus centering the data will not help to reduce that kind of correlation. As we saw, MCMC methods, such as Metropolis-Hastings, have problems exploring highly-correlated spaces; the only way these methods find to properly get samples is to propose a new step in the neighborhood of the previous step. As a result, the exploration becomes highly autocorrelated and painfully slow. The slow mixing can be so dramatic that simply increasing the number of samples (draws) is not a reasonable or practicable solution. Samplers such as NUTS are better at this job as they propose steps based on the curvature of the parameter space, but as we already discussed, the efficiency of the sampling process is highly dependent on the tuning phase. For some geometries of the posterior, such as those induced by hierarchical models, the tuning phase gets overly tuned to the local neighborhood where the chains started, making the exploration of other regions inefficient as the new proposals are more random, resembling the behavior of Metropolis-Hastings.

Summary

In this chapter, we have taken a conceptual walk through some of the most common methods used to compute the posterior distribution, including variational methods and Markov Chain Monte Carlo methods. We have put special emphasis on universal inference engines, methods that are designed to work on any given model (or at least a broad range of models). These methods are the core of any probabilistic programming language as they allow for automatic inference, letting users concentrate on iterative model design and interpretations of the results. We also discussed numerical and visual tests for diagnosing samples. Without good approximations to the posterior distribution, all the advantages and flexibility of the Bayesian framework vanish, so evaluating the quality of the inference process is crucial for us to be confident of the quality of the inference process itself.

Exercises

1. Use the grid approach with other priors; for example, try with `prior = (grid <= 0.5).astype(int)` or `prior = abs(grid - 0.5)`, or try defining your own crazy priors. Experiment with other data, such as increasing the total amount of data or making it more or less even in terms of the number of heads you observe.
2. In the code we use to estimate π , keep N fixed and re-run the code a couple of times. Notice that the results are different because we are using random numbers, but also check that the errors are more or less in the same order. Try changing the number of N points and re-run the code. Can you *guesstimate* how the number of N points and the error are related? For a better estimation, you may want to modify the code to compute the error as a function of N . You can also run the code a few times with the same N and compute the mean error and standard deviation of the error. You can plot these results using the `plt.errorbar()` function from matplotlib. Try using a set of N s, such as 100, 1,000, 10,000; that is a difference of one order of magnitude or so.
3. Modify the `func` argument you pass to the `metropolis` function; try using the values of the prior from Chapter 1, *Thinking Probabilistically*. Compare this code to the grid approach; which part should be modified to be able to use it to solve a Bayesian inference problem?

4. Compare your answer from the previous exercise to this code by Thomas Wiecki: <http://twiecki.github.io/blog/2015/11/10/mcmc-sampling/>
<http://twiecki.github.io/blog/2015/11/10/mcmc-sampling/>.
5. Revisit at least a few of the models from previous chapters and run all the diagnostic tools we saw in this chapter.
6. Revisit the code from all previous chapters, find those with divergences, and try to reduce the number of them.

9

Where To Go Next?

"Statistician is the technical term for a cynical data scientist."

- Jim Savage

I wrote this book to introduce the main concepts and practices of Bayesian statistics to those who are already familiar with Python and the Python data stack, but not very familiar with statistical analysis. Having read the previous eight chapters, you should have a reasonable practical understanding of many of the main topics of Bayesian statistics. Although you will not be an *expert-Bayesian-ninja-hacker* (whatever that could be), you should be able to create your own probabilistic models to solve your own data analysis problems. If you are really into Bayesian statistics, this book will not be enough – probably no single book will be enough. To become more fluent in Bayesian statistics, you will need practice, time, patience, enthusiasm, and more practice, and you will need to revisit ideas and concepts from a different perspective.

In the repository (<https://github.com/aloctavodia/BAP>), you will find examples complementing those that are discussed in this book. These are examples that did not fit this book, either due to space or time. In fact, at the time of writing this book, there are no extra examples yet, but I will add examples there from time to time. To gather extra material, you should also check the PyMC3 documentation at <https://docs.pymc.io>, especially the *examples* section, which is full of many examples of models that were covered in this book and many others that were not. As you already know, ArviZ is a really new library, but we are already writing an educational resource about exploratory analysis of Bayesian models. We hope this will be a useful reference, especially for newcomers to Bayesian modeling (https://github.com/arviz-devs/arviz_resources).

If you find mistakes in this book, either text or code, you can file an issue at <https://github.com/aloctavodia/BAP>. If you have general questions about Bayesian statistics, especially those related to PyMC3 or ArviZ, you can ask questions at <https://discourse.pymc.io/>.

In the next few paragraphs, I list some material that has definitely influenced my *Bayesian way of thinking*. This list is by no means exhaustive. I am confident that you will also find at least part of this material very useful and inspiring.

If you want to keep learning about Bayesian statistics in general, check out this list:

- I strongly recommend that you read *Statistical Rethinking* by Richard McElreath. This is a superb introductory book about Bayesian Analysis. The problem :-) is that the examples are in R/Stan. Hence, a group of volunteers have ported the examples in this book to Python/PyMC3. Check out the GitHub repository for more information: <https://github.com/pymc-devs/resources/tree/master/Rethinking>.
- Another book that's been ported to PyMC3 is *Doing Bayesian Data Analysis* by John K. Kruschke (also known as *the puppy book*). This is another nice introductory book about Bayesian analysis. Most of the examples from the first edition of this book have been ported to Python/PyMC3 in the following GitHub repository: https://github.com/aloctavodia/Doing_bayesian_data_analysis. You can find the second edition here: <https://github.com/JWarmenhoven/DBDA-python>. Unlike Statistical Rethinking, *the puppy book* is more focused on how to carry the Bayesian analog of many commonly frequentist statistical analysis. Depending on what you want, this can be a *pro* or a *con* of this book.
- Allen B. Downey has many great books, and Think Bayes (<http://greenteapress.com/wp/think-bayes/>) is not an exception. In this book, you will find several interesting examples and scenarios that will certainly challenge you and help you grasp the Bayesian approach to problem solving. This book does not use PyMC3, but a Python library that's constructed around Think Bayes. The second edition, which is still not written at the moment of publishing this book, will use PyMC3 and probably even ArviZ. You can check the repository for this second edition: <https://github.com/AllenDowney/ThinkBayes2>
- Another (optionally free/paid resource) is *Probabilistic Programming and Bayesian Methods for Hackers*, by Cameron Davidson-Pilon and several contributors. This book/notebook was originally written using PyMC2 and has now been ported to PyMC3: <https://github.com/quantopian/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers>.

- One book that is generally referred to as *The Bayesian* book is *Bayesian Data Analysis* by Andrew Gelman and others. This is definitely a great book, although it is not really an introductory book and probably works better as a reference book and not a textbook. If you are not familiar with statistics (Bayesian or otherwise), I recommend that you first pick *Statistical Rethinking* by Richard McElreath and then try *Bayesian Data Analysis*. You may also want to check out the book *Data Analysis Using Regression and Multilevel/Hierarchical Models*, by Andrew Gelman and Jennifer Hill.

If you want to keep learning about Gaussian processes, check out the following book:

- *Gaussian Processes for Machine Learning*, by Carl Edward Rasmussen and Christopher K. I. Williams, is *the* book for Gaussian processes. It was awarded with the 2009 DeGroot Prize of the International Society for Bayesian Analysis, and the only downside is that we all want a new edition!

A couple of machine learning books with a Bayesian twist are as follows:

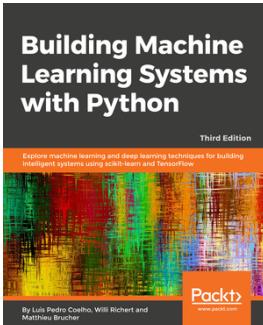
- *Machine Learning: A probabilistic Probabilistic Perspective*, by Kevin P. Murphy. This is a great book that tries to explain how many methods and models work by using a probabilistic approach. You may find this book a little dry or very concise and to the point, depending on your mathematical inclinations. Either way, this book is full of examples and was written with a very practical aim. Kevin Murphy has taken examples and ideas from many other sources and thus this book is a great summary of many other great resources. The first time I heard about deep learning was from this book, way before it became the new cool kid on the block.
- *Pattern Recognition and Machine Learning*, by Christopher Bishop, is a classical book in machine learning and has considerable overlap with *Machine Learning: A Probabilistic Perspective*, although probably with a little bit more of a Bayesian perspective. It is also maybe a little bit easier to read as a textbook than Murphy's, which is more of a reference book.

As a child, I dreamed of flying cars, clean unlimited energy, vacations on Mars or the Moon, a global government pursuing the well being of the entire human race.. yeah, I know... I used to be a dreamer! For many reasons, we have none of that. Instead, we have something that was completely unimagined, at least for me, just a couple of decades ago: the democratization of very powerful computer methods. One of the side effects of the computer revolution is that any person with a modest understanding of a programming language like Python now has access to a plethora of computational methods for data analysis, simulations, and other complex tasks. I think this is super-great, but also an invitation to be extra careful about these methods. The way I learned about statistics as an undergrad and how I had to memorize how to use *canned* methods was frustrating, useless, and completely unrelated to all of these changes. At a very personal level, this book is perhaps a response to that frustrating experience.

I tried to write a statistical book with an emphasis on a modeling approach and a judicious context-dependent analysis. I am unsure whether I had really succeeded in this front. One reason for this is probably that I still need to learn more about this (maybe we as a community need to learn more about this). Another reason is that a proper statistical analysis should be guided by the domain-knowledge and context, and providing context is generally difficult in a book with a very broad target audience. Nevertheless, I hope that I have provided a sane, skeptical perspective regarding statistical models, some useful examples, and enough momentum for you to keep learning.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

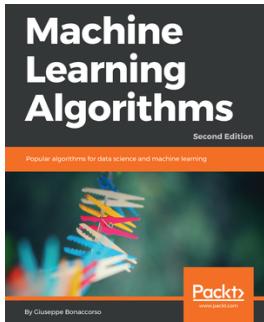


Building Machine Learning Systems with Python - Third Edition

Luis Pedro Coelho, Willi Richert, Matthieu Brucher

ISBN: 9781788623223

- Build a classification system that can be applied to text, images, and sound
- Employ Amazon Web Services (AWS) to run analysis on the cloud
- Solve problems related to regression using scikit-learn and TensorFlow
- Recommend products to users based on their past purchases
- Understand different ways to apply deep neural networks on structured data
- Address recent developments in the field of computer vision and reinforcement learning



Machine Learning Algorithms - Second Edition

Giuseppe Bonacorso

ISBN: 9781789347999

- Study feature selection and the feature engineering process
- Assess performance and error trade-offs for linear regression
- Build a data model and understand how it works by using different types of algorithm
- Learn to tune the parameters of Support Vector Machines (SVM)
- Explore the concept of natural language processing (NLP) and recommendation systems
- Create a machine learning architecture from scratch

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

autocorrelation 317, 318
automatic differentiation 42
Automatic Differentiation Variational Inference (ADVI) 298

B

Bayes factors
about 202, 203
and information criteria 209, 211, 212
computing 204, 205, 206, 207
computing, issues 207
computing, Sequential Monte Carlo used 208
remarks 203, 204
Bayes' theorem 18, 20
Bayesian analysis
communicating 32
model notation 32
posterior distribution, summarizing 33
Bayesian Information Criterion (BIC) 195
Bayesian modeling 9
Bayesian Modeling Averaging 199
Bayesian statistics 325, 326, 328
Boltzmann distribution 162

C

categorical distribution 225
centered model 312
central limit theorem (CLT) 52
chemical shift 52
clustering 237
coal-mining disasters 279, 282
Cohen's d 65, 66, 67
coin-flipping problem 21
coin-flipping problem, PyMC3 primer
inference button, pushing 44, 45

model specification 43, 44
coin-flipping problem, single-parameter inference
general model 22
influence of prior, selecting 30, 31
likelihood, selecting 22, 24
posterior, computing 27, 29
posterior, obtaining 26
posterior, plotting 27, 29
prior, selecting 24, 26
concentration of measure 292
concentration parameter 238
confounding variables 125, 128
continuous mixtures
about 246
beta-binomial and negative binomial 246, 247
students t-distribution 247
convergence 312, 313, 314
covariance functions 254, 256
Cox processes
about 279
coal-mining disasters 279, 282
redwood dataset 282, 285
curse of dimensionality 292

D

data
working with 8
determination coefficient 101
Deviance Information (DIC) 195
diagnostic samples, Metropolis
about 311, 312
autocorrelation 317, 318
convergence 312, 313, 314, 315, 316
divergences 319, 321
effective sample sizes 318, 319
Monte Carlo error 317
non-centered parameterization 322

Dirichlet distribution 226, 227, 228
Dirichlet process (DP) 238
discriminative classifier 164
divergences 319
draws 299

E

effect size
about 64
reporting, with probability of superiority 67
entropy 214
evidence 20
experimental design 8
Exploratory Data Analysis (EDA) 7

F

finite mixture models
about 223, 224, 225, 237
categorical distribution 225
Dirichlet distribution 226, 227, 228
K, selecting 232, 233, 235, 236
non-identifiability 230, 232

G

gamma function 24
Gaussian 52
Gaussian inferences 52, 53, 54, 55, 57, 58
Gaussian process classification 271, 274, 277, 279
Gaussian process regression 258, 260, 262, 263, 264
Gaussian processes (GPs) 257, 258
Generalized linear models 144, 145
generative classifier 164
GLM module 175, 176
gradient 308
Grid computing 290, 292
groups comparison
about 64, 65
Cohen's d 66, 67
tips dataset, using 68, 71, 72

H

Hamiltonian Monte Carlo (HMC) 307, 308
hierarchical linear regression

about 109, 111, 113, 115
causation 115, 116
correlation 115, 116
messiness 115, 116
hierarchical models
about 72, 75
example 80, 81, 82, 84
shrinkage 76, 79
Highest-Posterior Density (HPD) 33, 47, 98, 165
Hybrid Monte Carlo 307
hyper-priors 73
hyperparameters 73
hyperpriors 109

I

identically distributed variables 16, 17
independently and identically distributed (iid) 16
independently distributed variables 16, 17
Inference engines 289, 290
inferential statistics 7
information criteria
about 192, 195
akaike information criterion 193, 194
leave-one-out cross-validation 194
log-likelihood and deviance 192, 193
LOO computations 198
model averaging 198, 199, 200, 201
model, comparing with PyMC3 195, 196, 197, 198
WAIC reliability 198
widely applicable information criterion 194
Integrated Nested Laplace Approximation 295
intercept 89
interquartile range (IQR) 183
inverse link function 144
Iris dataset 147, 149

K

K-fold cross-validation 191
Kernel Density Estimation (KDE) 45, 222
kernels 254, 256
Kullback-Leibler (KL) divergence 216, 218, 295

L

label-switching problem 230
Laplace method 293
Large Hadron Collider (LHC) 8
Lasso regression 212
least squares fitting 89
leave-one-out cross-validation (LOOCV) 191
linear discriminant analysis (LDA) 165
linear models
 about 251
 data, modifying before running 95, 96
 high autocorrelation 94, 95
linear regression models 89, 90, 91, 93
logistic regression
 about 145, 146
 Iris dataset 147, 149
 logistic model 146, 147
 logistic model, applying to Iris dataset 150, 151, 152, 153
Lorentz distribution 59
loss functions 49, 51, 52

M

machine learning (ML) 8, 87, 88, 89
marginal likelihood 202
Markov chain 302
Markov Chain Monte Carlo (MCMC) 67, 289
Markovian methods
 about 298, 299
 Hamiltonian Monte Carlo (HMC) 307, 308
 Markov chain 302
 Metropolis-Hastings 302, 303, 304, 305, 307
 Monte Carlo 300, 301, 302
 Sequential Monte Carlo (SMC) 308, 309, 310
masking effect variables 133, 134
maximum a posteriori (MAP) 90, 176, 294
Metropolis-Hastings 302, 303, 305, 307
mixture models
 about 222
 non-finite mixture models 237
model-based clustering 237
modeling functions
 about 252, 253
covariance functions 254, 256

Gaussian processes (GPs) 257, 258
kernels 254, 256
multivariate Gaussians 254
models 7
Monte Carlo 300, 302
Monte Carlo error 317
multicollinearity 128, 131, 132
multilevel model 72
multiple linear regression
 about 88, 120, 121, 123, 124, 125
 confounding variables 125, 128
 interactions, adding 135, 136
 masking effect variables 133, 134
 multicollinearity 128, 131, 132
 redundant variables 125, 128
multiple logistic regression
 about 153
 boundary decision 153, 154
 coefficients, interpreting 155, 156, 157, 158
 correlated variables 158, 159, 160
 discriminative and generative models 164, 166
 model, implementing 154
 softmax regression 162, 163, 164
 unbalanced classes 160, 162
multivariate Gaussians 254

N

No-U-Turn Sampler (NUTS) 308
non-centered model 312
non-finite mixture model
 about 237
Dirichlet process 238, 239, 241, 242, 243, 244, 245
non-linear data 251
Non-Markovian methods
 about 290
 automatic differentiation variational inference 298
 Grid computing 290, 292
 Quadratic methods 293, 294, 295
 variational methods 295, 296, 297
normal approximation 293
Nuclear magnetic resonance (NMR) 52

O

Occam's razor
about 185, 186
cross-validation 191, 192
parameters overfitting 187, 188, 189
parameters underfitting 189
predictive accuracy measures 190, 191
simplicity and accuracy 190
simplicity and accuracy, balance 189
odds 156
over-dispersion 247
overfitting 120

P

parabolic model 181
parameter non-identifiability 230
parameters
interpreting, of polynomial regression 119
Pearson correlation coefficient
about 100, 101
from multivariate Gaussian 101, 102, 104
Poisson regression
about 167
and ZIP regression 171, 172
Poisson distribution 167, 168, 169
Zero-inflated Poisson model 169, 170
polynomial regression
about 116, 117, 118, 119, 120
parameters, interpreting 119
posterior distribution
summarizing 33
summarizing, to use highest-posterior density (HPD) 33, 34
posterior predictive checks 34, 35, 180, 181, 185
posterior-based decisions
about 47
loss functions 49, 50, 52
Region Of Practical Equivalence (ROPE) 47, 49
posterior
interpreting 97, 99
summarizing 45, 46
visualizing 97, 99
Principal Component Analysis (PCA) 132
priors

regularizing 212, 213
probabilistic models 9
probabilistic programming 41, 42
probabilistic programming languages (PPL) 41
probabilities
defining 11
probability distributions 9, 12, 14, 16
probability of superiority
about 65
used, for reporting effect size 67
probability theory
about 10
interpreting 10, 11
pseudo Bayesian Modeling Averaging 199
PyMC3 primer
about 42
coin-flipping problem 43
PyMC3
model, comparing with 195, 196, 197, 198

Q

quadratic linear discriminant (QDA) 166
quadratic method 293
quadratic model 181

R

redundant variables 125, 128
redwood dataset 282, 285
Region Of Practical Equivalence (ROPE) 47, 49
regression
with spatial autocorrelation 265, 267, 269, 271
regularizing priors 30, 132
ridge regression 212
robust inferences
about 58
Student's t-distribution 59, 61, 64
robust linear regression 104, 106, 108, 109
robust logistic regression 174, 175

S

Sequential Monte Carlo (SMC) 208, 308, 309, 310, 311
shrinkage 76, 77, 79
simple linear regression 88
single-parameter inference

about 21
coin-flipping problem 21
soft-clustering 237
softmax regression 162, 164
sopa seca 145
spatial autocorrelation
 used, for regression 265, 267, 269, 271
statistical methods
 Exploratory Data Analysis (EDA) 7
 inferential statistics 7
statistics 7
stick-breaking process 239
Student's t-distribution 59, 61, 64

T

Theano
 about 42
 reference 42
Tikhonov regularization 212

tips dataset
 using 68, 71, 72

V

variable variance 136, 139
variational parameters 296

W

WAIC
 about 213, 214
 entropy 214, 216
 Kullback-Leibler (KL) divergence 216, 218
weakly-informative priors 30
World Health Organization (WHO)
 about 73
 reference 136

Z

Zero-inflated Poisson model 169