# Bayesian Analysis with Python

Unleash the power and flexibility
of the Bayesian framework

**Osvaldo Martin**

Packt>

# Bayesian Analysis with Python



First published: November 2016

Production reference: 1211116

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78588-380-4

www.packtpub.com

# Contents

# Preface

Bayesian statistics has been around for more than 250 years now. During this time it has enjoyed as much recognition and appreciation as disdain and contempt. Through the last few decades it has gained more and more attention from people in statistics and almost all other sciences, engineering, and even outside the walls of the academic world. This revival has been possible due to theoretical and computational developments. Modern Bayesian statistics is mostly computational statistics. The necessity for flexible and transparent models and a more interpretation of statistical analysis has only contributed to the trend.

Here, we will adopt a pragmatic approach to Bayesian statistics and we will not care too much about other statistical paradigms and their relationship to Bayesian statistics. The aim of this book is to learn about Bayesian data analysis with the help of Python. Philosophical discussions are interesting but they have already been undertaken elsewhere in a richer way than we can discuss in these pages.

We will take a modeling approach to statistics, we will learn 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 great library for Bayesian statistics that hides most of the mathematical details and computations from the user. Bayesian methods are theoretically grounded in probability theory and hence 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 mathematical knowledge, as you will be able to verify by yourself throughout this book.

# What this book covers

*Chapter 1*, *Thinking Probabilistically – A Bayesian Inference Primer*, tells us about Bayes' theorem and its implications for data analysis. We then proceed to describe the Bayesian-way of thinking and how and why probabilities are used to deal with uncertainty. This chapter contains the foundational concepts used in the rest of the book.

*Chapter 2*, *Programming Probabilistically – A PyMC3 Primer*, revisits the concepts from the previous chapter, this time from a more computational perspective. The PyMC3 library is introduced and we learn how to use it to build probabilistic models, get results by sampling from the posterior, diagnose whether the sampling was done right, and analyze and interpret Bayesian results.

*Chapter 3*, *Juggling with Multi-Parametric and Hierarchical Models*, tells us about the very basis of Bayesian modeling and we start adding complexity to the mix. We learn how to build and analyze models with more than one parameter and how to put structure into models, taking advantages of hierarchical models.

*Chapter 4*, *Understanding and Predicting Data with Linear Regression Models*, tells us about how linear regression is a very widely used model per se and a building block of more complex models. In this chapter, we apply linear models to solve regression problems and how to adapt them to deal with outliers and multiple variables.

*Chapter 5*, *Classifying Outcomes with Logistic Regression*, generalizes the the linear model from previous chapter to solve classification problems including problems with multiple input and output variables.

*Chapter 6*, *Model Comparison*, discusses the difficulties associated with comparing models that are common in statistics and machine learning. We will also learn a bit of theory behind the information criteria and Bayes factors and how to use them to compare models, including some caveats of these methods.

*Chapter 7*, *Mixture Models*, discusses how to mix simpler models to build more complex ones. This leads us to new models and also to reinterpret models learned in previous chapters. Problems, such as data clustering and dealing with count data, are discussed.

*Chapter 8*, *Gaussian Processes*, closes the book by briefly discussing some more advanced concepts related to non-parametric statistics. What kernels are, how to use kernelized linear regression, and how to use Gaussian processes for regression are the central themes of this chapter.

# What you need for this book

This book is written for Python version >= 3.5, and it is recommended that you use the most recent version of Python 3 that is currently available, although most of the code examples may also run for older versions of Python, including Python 2.7 with minor adjustments.

Maybe the easiest way to install Python and Python libraries is using Anaconda, a scientific computing distribution. You can read more about Anaconda and download it from `https://www.continuum.io/downloads`. Once Anaconda is in our system, we can install new Python packages with this command: conda install NamePackage.

We will use the following python packages:

- `Ipython 5.0`
- `NumPy 1.11.1`
- `SciPy 0.18.1`
- `Pandas  0.18.1`
- `Matplotlib 1.5.3`
- `Seaborn 0.7.1`
- `PyMC3 3.0`

# Who this book is for

Undergraduate or graduate students, scientists, and data scientists who are not familiar with the Bayesian statistical paradigm and wish to learn how to do Bayesian data analysis. No previous knowledge of statistics is assumed, for either Bayesian or other paradigms. The required mathematical knowledge is kept to a minimum and all concepts are described and explained with code, figures, and text. Mathematical formulas are used only when we think it can help the reader to better understand the concepts. The book assumes you know how to program in Python. Familiarity with scientific libraries such as NumPy, matplotlib, or Pandas is helpful but not essential.

# Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "To compute the HPD in the correct way we will use the function `plot_post`."

A block of code is set as follows:

```
n_params = [1, 2, 4]
p_params = [0.25, 0.5, 0.75]
x = np.arange(0, max(n_params)+1)
f, ax = plt.subplots(len(n_params), len(p_params), sharex=True,
  sharey=True)
for i in range(3):
    for j in range(3):
        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='b', lw=5)
        ax[i,j].set_ylim(0, 1)
        ax[i,j].plot(0, 0, label="n = {:3.2f}\np =
         {:3.2f}".format(n, p), alpha=0)
        ax[i,j].legend(fontsize=12)
ax[2,1].set_xlabel('$\\theta$', fontsize=14)
ax[1,0].set_ylabel('$p(y|\\theta)$', fontsize=14)
ax[0,0].set_xticks(x)
```

Any command-line input or output is written as follows:

```
conda install NamePackage
```

# Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

# Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

# Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

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 bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Bayesian-Analysis-with-Python`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

# Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/BayesianAnalysiswithPython_ColorImages.pdf`.

# Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

# 1
# Thinking Probabilistically - A Bayesian Inference Primer

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

*-Pierre-Simon Laplace*

In this chapter, we will learn the core concepts of Bayesian statistics and some of the instruments in the Bayesian toolbox. We will use some Python code in this chapter, but this chapter will be mostly theoretical; most of the concepts in this chapter will be revisited many times through the rest of the book. This chapter, being intense on the theoretical side, may be a little anxiogenic for the coder in you, but I think it will ease the path to 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
- Installing all Python packages

# Statistics as a form of modeling

Statistics is about collecting, organizing, analyzing, and interpreting data, and hence statistical knowledge is essential for data analysis. Another 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 your data is clean and tidy, programming will still be very useful since modern Bayesian statistics is mostly computational statistics.

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 the statistical pantry, pick one can and open it, add data to taste and stir until obtaining a consistent p-value, preferably under *0.05* (if you don't know what a p-value is, don't worry; we will not use them in this book). The main goal in this type of course is to teach you how to pick the proper can. We will take a different approach: we will also learn some recipes, but this will be home-made rather than canned food; we will learn how to mix fresh ingredients that will suit different gastronomic occasions. But before we can cook, we must learn some statistical vocabulary and also some concepts.

# Exploratory data analysis

Data is an essential ingredient of statistics. Data comes from several sources, such as experiments, computer simulations, surveys, field observations, and so on. If we are the ones that will be 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 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 effort. In this book we will assume that we already have collected the data and also that the data is 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. If you want to learn how to use Python for cleaning and manipulating data and also a primer on machine learning, you should probably read the book *Python Data Science Handbook* by *Jake VanderPlas*.

OK, so let's assume we have our dataset; usually, a good idea is to explore and visualize it in order to get some intuition about what we have in our hands. This can be achieved through what is known as **Exploratory Data Analysis** (**EDA**), which basically consists of the following:

- Descriptive statistics
- Data visualization

The first one, descriptive statistics, is about how to use some measures (or statistics) to summarize or characterize the data in a quantitative manner. You probably already know that you can describe data using the mean, mode, standard deviation, interquartile ranges, and so forth. The second one, data visualization, is about visually inspecting the data; you probably are familiar with representations such as histograms, scatter plots, and others. While EDA was originally thought of as something you apply to data before doing any complex analysis or even as an alternative to complex model-based analysis, through the book we will learn that EDA is also applicable to understanding, interpreting, checking, summarizing, and communicating the results of Bayesian analysis.

# Inferential statistics

Sometimes, plotting our data and computing simple numbers, such as the average of our data, is all we need. Other times, we want to make a generalization based on our data. We may want to understand the underlying mechanism that could have generated the data, or maybe we want to make predictions for future (yet unobserved) data points, or we need to choose among several competing explanations for the same observations. That's the job of **inferential statistics**. To do inferential statistics we will rely on probabilistic models. There are many types of models and most of science, and I will add all of our understanding of the *real world*, is through models. The brain is just a machine that models reality (whatever reality might be) see this TED talk about the machine that builds the reality `http://www.tedxriodelaplata.org/videos/m%C3%A1quina-construye-realidad`.

What are models? Models are simplified descriptions of a given system (or process). Those descriptions are purposely designed to capture only the most relevant aspects of the system, and hence, most models do not pretend they are able to explain everything; on the contrary, if we have a simple and a complex model and both models explain the data more or less equally well, we will generally prefer the simpler one. This heuristic for simple models is known as Occam's razor, and we will discuss how it is related to Bayesian analysis in *Chapter 6*, *Model Comparison*.

Model building, no matter which type of model you are building, is an iterative process following more or less the same basic rules. 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 will build models. Most of the time, models will be crude approximations, but most of the time this is all we need.
2. Then we will use Bayes' theorem to add data to our models and derive the logical consequences of mixing the data and our assumptions. We say we are conditioning the model on our data.
3. Lastly, we will check that the model makes sense according to different criteria, including our data and our expertise on the subject we are studying.

In general, we will find ourselves performing these three steps in a non-linear iterative fashion. Sometimes we will retrace our steps at any given point: maybe we made a silly programming mistake, maybe we found a way to change the model and improve it, maybe we need to add more 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 the uncertainty in our data, so let's take a walk through the garden of forking paths.

# Probabilities and uncertainty

While Probability Theory is a mature and well-established branch of mathematics, there is more than one interpretation of what probabilities are. To a Bayesian, a probability is a measure that quantifies the uncertainty level of a statement. If we know nothing about coins and we do not have any data about coin tosses, it is reasonable to think that the probability of a coin landing heads could take any value between *0* and *1*; that is, in the absence of information, all values are equally likely, our uncertainty is maximum. If we know instead that coins tend to be balanced, then we may say that the probability of a coin landing is exactly *0.5* or may be around *0.5* if we admit that the balance is not perfect. If now, we collect data, we can update these prior assumptions and hopefully reduce the uncertainty about the bias of the coin. 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 x 10-31* kg, or the probability of the 9th of July of 1816 being a sunny day. Notice, for example, that the question of whether or not life exists on Mars has a binary outcome but what we are really asking is how likely is it to find life on Mars given our data and what we know about biology and the physical conditions on that planet? The statement is about our state of knowledge and not, directly, about a property of nature. We are using probabilities because we cannot be sure about the events, not because the events are necessarily random. Since this definition of probability is about our epistemic state of mind, sometimes it is referred to as the subjective definition of probability, explaining the slogan of subjective statistics often attached to the Bayesian paradigm. Nevertheless, this definition does not mean all statements should be treated as equally valid and so anything goes; this definition is about acknowledging that our understanding about the world is imperfect and conditioned on the data and models we have made. There is not such a thing as a model-free or theory-free understanding of the world; even if it were be possible to free ourselves from our social preconditioning, we will end up with a biological limitation: our brain, subject to the evolutionary process, has been wired with models of the world. We are doomed to think like humans and we will never think like bats or anything else! Moreover, the universe is an uncertain place and, in general the best we can do is to make probabilistic statements about it. Notice that it does not matter if the underlying reality of the world is deterministic or stochastic; we are using probability as a tool to quantify uncertainty.

Logic is about thinking without making mistakes. Under the Aristotelian or classical logic, we can only have statements taking the values 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 one has probability *0*. We would assign a probability of *1* about life on Mars 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 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. Interesting enough, 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. Now that we know the Bayesian interpretation of probability, let's see some of the mathematical properties of probabilities. For a more detailed study of probability theory, you can read *Introduction to probability* by *Joseph K Blitzstein & Jessica Hwang*.

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

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

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 expression *p(A, B)* represents the joint probability of A and *B*. The expression *p(A | B)* 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 a pavement is wet is different from the probability that the pavement is wet if we know (or given that) is raining. A conditional probability can be larger than, smaller than or equal to the unconditioned probability. If knowing *B* does not provides us with information about *A*, then *p(A | B)=p(A)*. That is *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.

Conditional probabilities are a key concept in statistics, and understanding them is crucial to understanding Bayes' theorem, as we will see soon. Let's try to understand them from a different perspective. If we reorder the equation for the product rule, we get the following:

$$p(A \mid B) = \frac{p(A, B)}{p(B)}$$

Notice that a conditional probability is always larger or equal than the joint probability. The reasons are that: we do not condition on zero-probability events, this is implied in the expression, and probabilities are restricted to be in the interval *[0, 1]*. Why do we divide by *p(B)?* Knowing *B* is equivalent to saying that we have restricted the space of possible events to B and thus, to find the conditional probability, we take the favorable cases and divide them by the total number of events. It is important to realize that all probabilities are indeed conditionals, there is not such a thing as an absolute probability floating in vacuum space. There is always some model, assumption, or condition, even if we don't notice or know them. The probability of rain is not the same if we are talking about Earth, Mars, or some other place in the Universe. In the same way, the probability of a coin landing heads or tails depends on our assumptions of the coin being biased in one way or another. Now that we are more familiar with the concept of probability, let's jump to the next topic, probability distributions.

# 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. A common and useful conceptualization in statistics is to think that data was generated from some probability distribution with unobserved parameters. Since the parameters are unobserved and we only have data, we will use Bayes' theorem to invert the relationship, that is, to go from the data to the parameters. Probability distributions are the building blocks of Bayesian models; by combining them in proper ways we can get useful complex models.

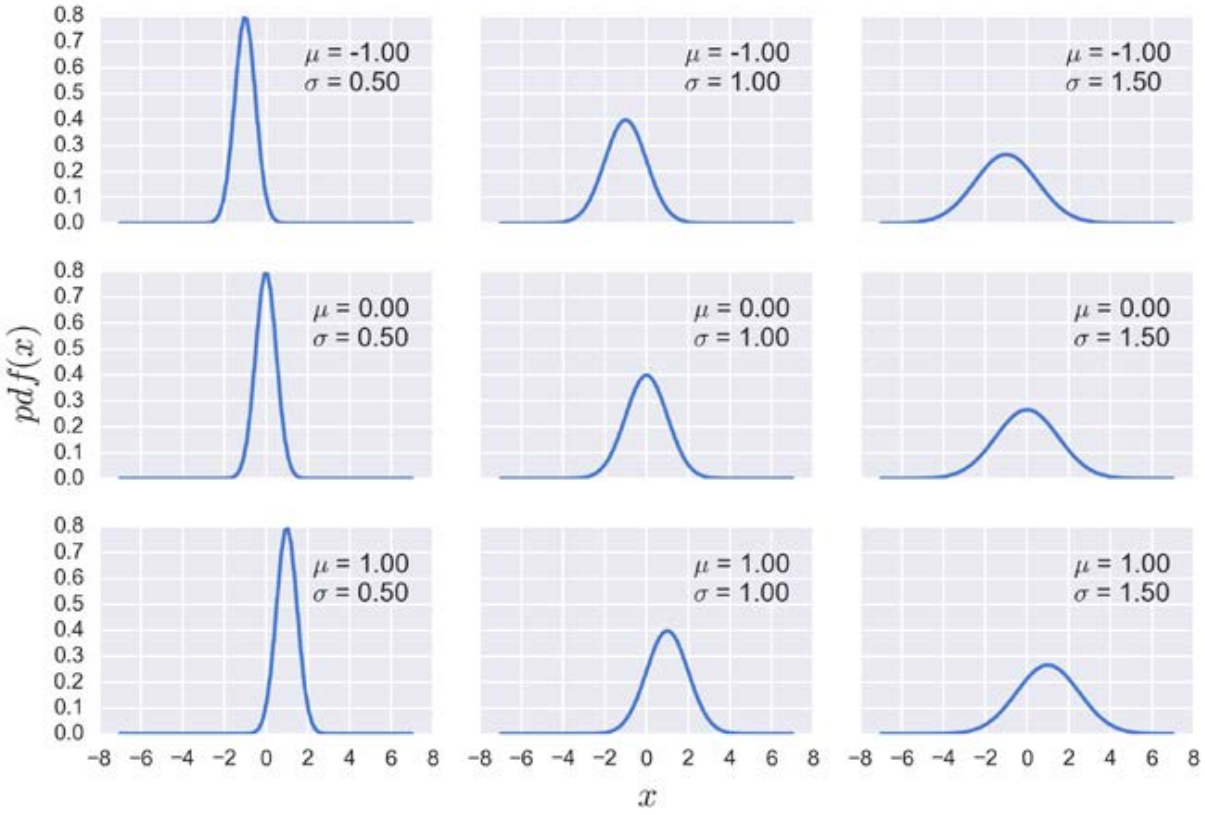
We will meet several probability distributions throughout the book; every time we discover one we will take a moment to try to understand it. Probably the most famous of all of them is the Gaussian or normal distribution. A variable *x* follows a Gaussian distribution if its values are dictated by the following formula:

$$pdf\left(x \mid \mu, \sigma\right) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

In the formula, $\mu$ and $\sigma$ are the parameters of the distributions. The first one 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). The second is the standard deviation, which can only be positive and dictates the spread of the distribution. Since there are an infinite number of possible combinations of $\mu$ and $\sigma$ values, 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 can be intimidating; a good way to break the ice is to use Python to explore them. Let's see what the Gaussian distribution family looks like:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sns

mu_params = [-1, 0, 1]
sd_params = [0.5, 1, 1.5]
x = np.linspace(-7, 7, 100)
f, ax = plt.subplots(len(mu_params), len(sd_params), sharex=True,
  sharey=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(0, 0,
        label="$\\mu$ = {:3.2f}\n$\\sigma$ = {:3.2f}".format
          (mu, sd), alpha=0)
        ax[i,j].legend(fontsize=12)
ax[2,1].set_xlabel('$x$', fontsize=16)
ax[1,0].set_ylabel('$pdf(x)$', fontsize=16)
plt.tight_layout()
```

The output of the preceding code is as follows:




A variable, such as $x$, that comes from a probability distribution is called a **random variable**. It is not that the variable can take any possible value. On the contrary, the values are strictly controlled by the probability distribution; the randomness arises from the fact that we could not predict which value the variable will take, but only the probability of observing those values. A common notation used to say that a variable is distributed as a Gaussian or normal distribution with parameters $\mu$ and $\sigma$ is as follows:

$$x \sim N(\mu, \sigma)$$

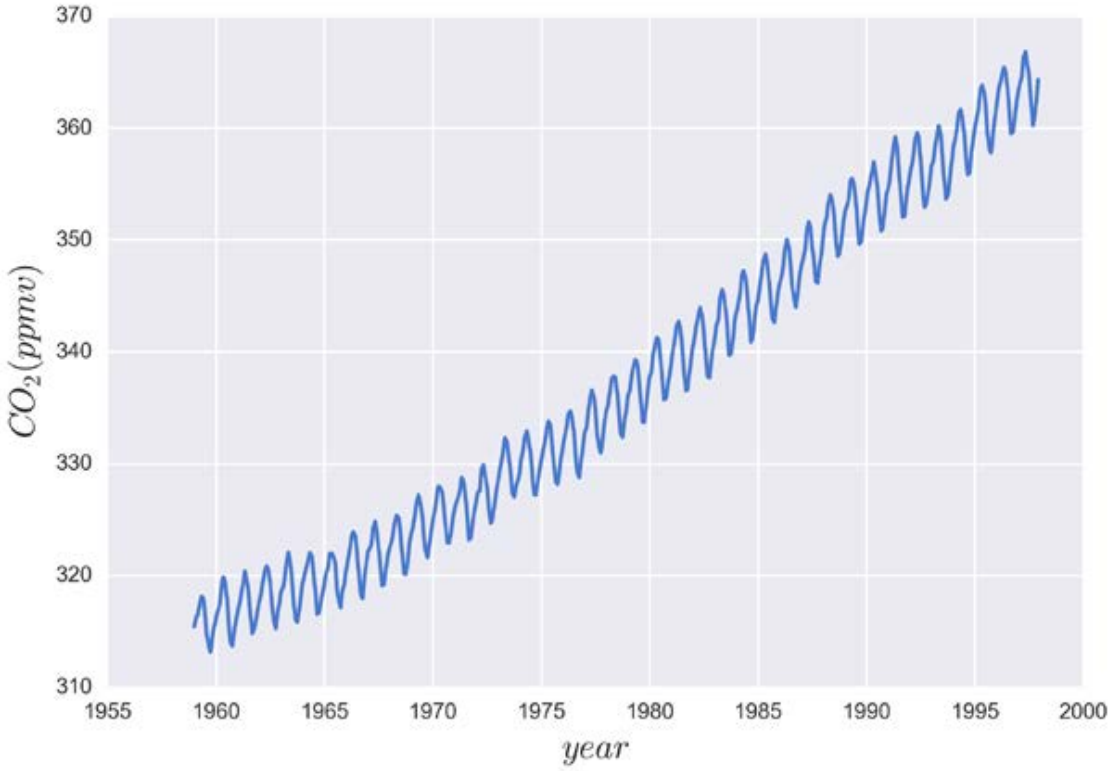
The symbol ~ (tilde) is read as *is distributed as*.

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

Many models assume that successive values of a 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, or *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 $CO_2$ measurements from 1959 to 1997. We are going to load the data (included with the accompanying code) and plot it.

```
data = np.genfromtxt('mauna_loa_CO2.csv', delimiter=',')
plt.plot(data[:,0], data[:,1])
plt.xlabel('$year$', fontsize=16)
plt.ylabel('$CO_2 (ppmv)$', fontsize=16)
```




Each point corresponds to the measured levels of atmospheric $CO_2$ per month. It is easy to see in this plot the temporal dependency of data points. 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 $CO_2$.

# Bayes' theorem and statistical inference

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

$$p(H \mid D) = \frac{p(D \mid H)\, p(H)}{p(D)}$$

Well, it is 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. In fact, we have already seen all the probability theory necessary to derive it:

- According to the product rule, we have the following:

$$p(H, D) = p(H \mid D)\, p(D)$$

- This can also be written as follows:

$$p(H, D) = p(D \mid H)\, p(H)$$

- Given than the terms on the left are equal, we can write the following:

$$p(D \mid H)\, p(H) = p(H \mid D)\, p(D)$$

- And if we reorder it, we get Bayes' theorem:

$$p(H \mid D) = \frac{p(D \mid H)\, p(H)}{p(D)}$$

Now, let's see what this formula implies and why it is important. First, it says that $p(D \mid H)$ is not necessarily the same as $p(D \mid H)$. This is a very important fact, one that's 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 necessary the same. The probability of having two legs given these someone is a human is not the same as the probability of being a human given that someone has two legs. Almost all humans have two legs, except for people that have suffered from accidents or birth problems, but a lot of non-human animals have two legs, such as birds.

If we replace $H$ with hypothesis and $D$ with data, Bayes' theorem tells us how to compute the probability of a hypothesis $H$ given the data $D$, 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 that using probability distributions so, in general, our $H$ will be a hypothesis in a very narrow sense. What we will be really doing is trying to find parameters of our models, that is, parameters of probability distributions. So maybe, instead of hypothesis, it is better to talk about models and avoid confusion. And by the way, don't try to set $H$ to statements such as "unicorns are real", unless you are willing to build a realistic probabilistic model of unicorn existence!

Since Bayes' theorem is central and we will use it over and over again, let's learn the names of its parts:

- $p(H)$: Prior
- $p(D \mid H)$: Likelihood
- $p(H \mid D)$: Posterior
- $p(D)$: Evidence

The **prior distribution** should reflect what we know about the value of some parameter before seeing the data $D$. If we know nothing, like Jon Snow, we will use flat priors that do not convey too much information. In general, we can do better, as we will learn through this book. The use of priors is why some people still think Bayesian statistics is 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.

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 parameters in our model and not a single value. This distribution is a balance of the prior and the likelihood. There is a joke that says: A Bayesian is one who, vaguely expecting a horse, and catching a glimpse of a donkey, strongly believes he has seen a mule. One 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 the joke 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 the data. In fact, the posterior of one analysis can be used as the prior of a new analysis after collecting new data. This makes Bayesian analysis particularly suitable for analyzing data that becomes available in sequential order. Some examples could be early warning systems for disasters that process *online* data coming from meteorological stations and satellites. For more details read about **online machine learning methods**.

The last term is the **evidence**, also known as marginal likelihood. Formally, the evidence is the probability of observing the data averaged over all the possible values the parameters can take. Anyway, for most of the parts of the book, we will not care about the evidence, and we will think of it as a simple normalization factor. This will not be problematic since we will only care about the relative values of the parameters and not their absolute ones. If we ignore the evidence, we can write Bayes' theorem as a proportionality:

$$p(H \mid D) \propto p(D \mid H)\, p(H)$$

Understanding the exact role of each term will take some time and will also require some examples, and that's what the rest of the book is for.

# Single parameter inference

In the last two sections, we have 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 the 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-flip problem 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 basic of Bayesian statistics; on the one hand, it is about tossing coins, something familiar to almost anyone; on the other, 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, safe or unsafe, healthy or unhealthy, and so on. 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 recorded the number of observed heads, so the data part is done. Getting the model will take a little bit more effort. Since this is our first model, we will do all the necessary math (don't be afraid, I promise it will be painless) and we will proceed step by step very slowly. In the next chapter we will revisit this problem by using PyMC3 to solve it numerically, that is, without us doing the math. Instead we will let PyMC3 and our computer do the math.

# 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 parameter $\theta$, and to represent the total number of heads for an $N$ number of tosses, we will use the variable $y$. According to Bayes' theorem we have the following formula:

$$p(\theta \mid y) \propto p(y \mid \theta)\, p(\theta)$$

Notice that we need to specify which prior $p(\theta)$ and likelihood $p(y \mid \theta)$ we will use. Let's start with the likelihood.

# Choosing the likelihood

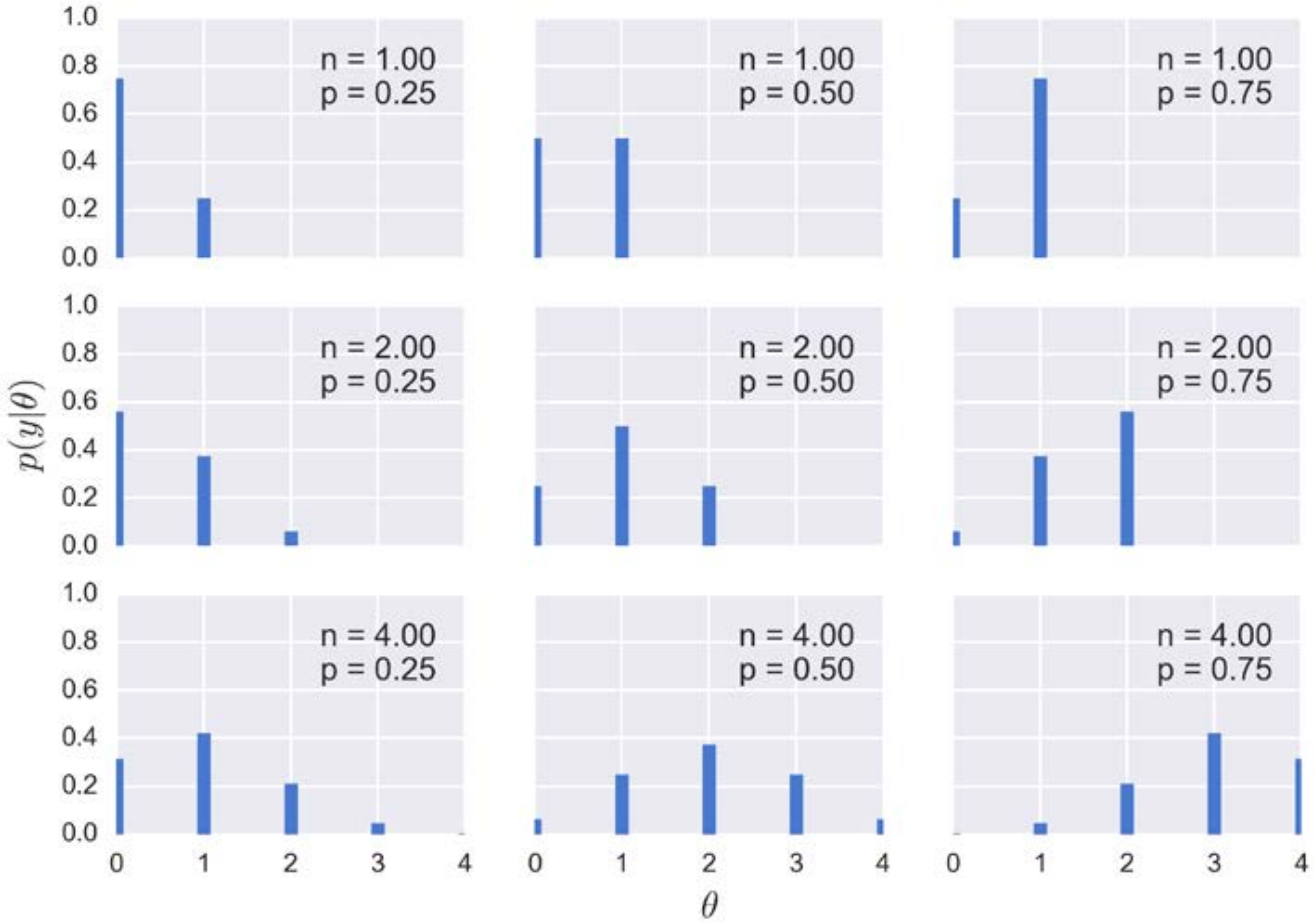
Let's assume that a coin toss does not affect other tosses, that is, we are assuming coin tosses are independent of each other. Let's also assume that only two outcomes are possible, heads or tails. I hope you agree these are very reasonable assumptions to make for our problem. Given these assumptions, a good candidate for the likelihood is the binomial distribution:

$$p(y \mid \theta) = \frac{N!}{N!(N-y)!}\theta^{y}1-\theta^{(N-y)}$$

This is a discrete distribution returning the probability of getting $y$ heads (or in general, success) out of $N$ coin tosses (or in general, trials or experiments) given a fixed value of $\theta$. The following code generates 9 binomial distributions; each subplot has its own legend indicating the corresponding parameters:

```
n_params = [1, 2, 4]
p_params = [0.25, 0.5, 0.75]
```

```
x = np.arange(0, max(n_params)+1)
f, ax = plt.subplots(len(n_params), len(p_params), sharex=True,
  sharey=True)
for i in range(3):
    for j in range(3):
        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='b', lw=5)
        ax[i,j].set_ylim(0, 1)
        ax[i,j].plot(0, 0, label="n = {:3.2f}\np =
         {:3.2f}".format(n, p), alpha=0)
        ax[i,j].legend(fontsize=12)
ax[2,1].set_xlabel('$\\theta$', fontsize=14)
ax[1,0].set_ylabel('$p(y|\\theta)$', fontsize=14)
ax[0,0].set_xticks(x)
```



The binomial distribution is also a reasonable choice for the likelihood. Intuitively, we can see that $\theta$ indicates how likely it is that we will obtain a head when tossing a coin, and we have observed that event *y* times. Following the same line of reasoning we get that $1-\theta$ is the chance of getting a tail, and that event has occurred *N-y* times.

OK, so if we know $\theta$, the binomial distribution will tell us the expected distribution of heads. The only problem is that we do not know $\theta$! 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.

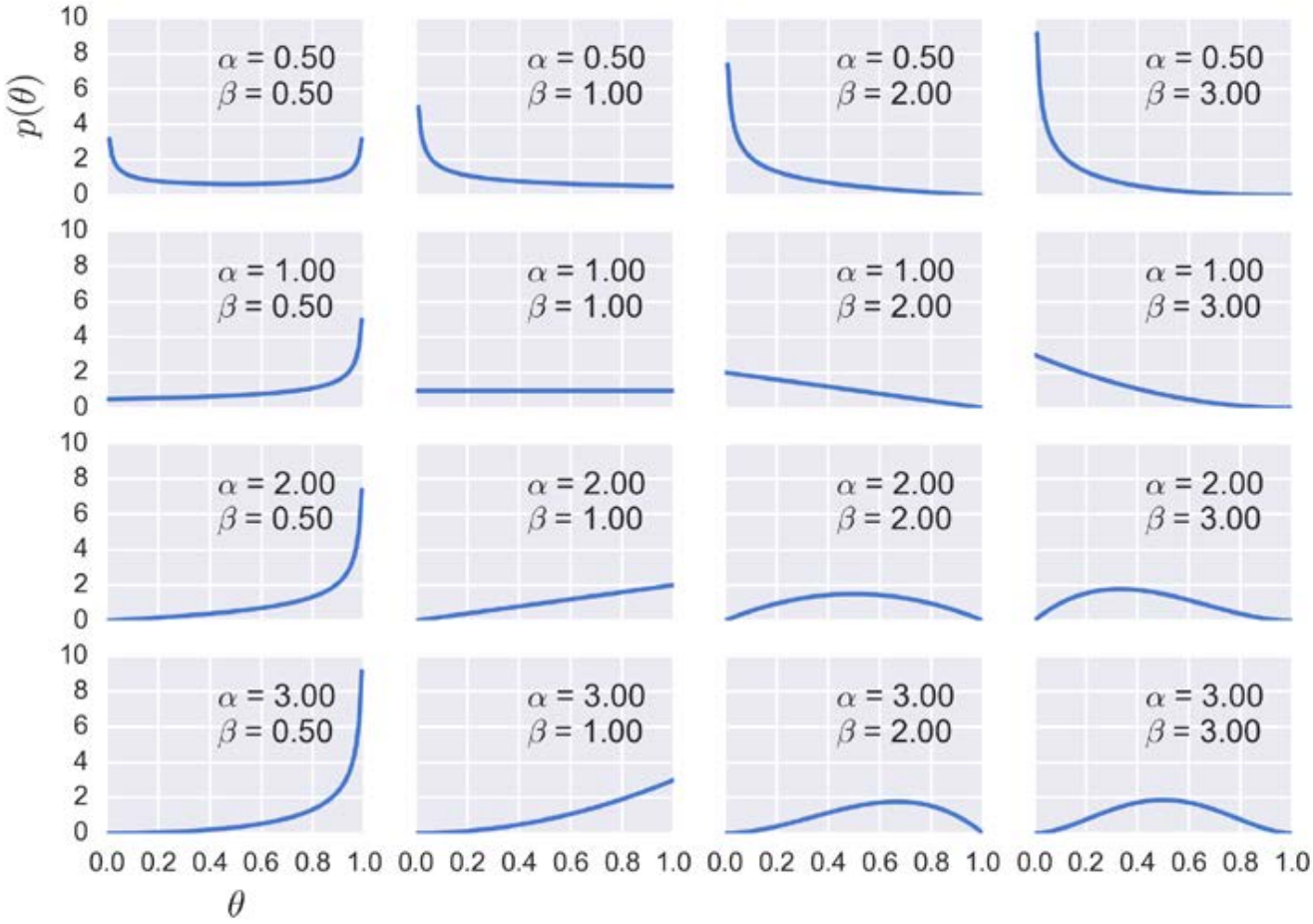
# Choosing the prior

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

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

If we look carefully we will see that the beta distribution looks similar to the binomial except for the term with the $\Gamma$. This is the Greek uppercase gamma letter and represents what is known as **gamma function**. All that we care about at this point is that the first term is a normalization constant that ensures the distribution integrates to *1* and that the beta distribution has two parameters, $\alpha$ and $\beta$, that control the distribution. 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)
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="$\\alpha$ = {:3.2f}\n$\\beta$ =
          {:3.2f}".format(a, b), alpha=0)
        ax[i,j].legend(fontsize=12)
ax[3,0].set_xlabel('$\\theta$', fontsize=14)
ax[0,0].set_ylabel('$p(\\theta)$', fontsize=14)
```




OK, the beta distribution is nice, 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 parameter $\theta$ is. Another reason is its versatility. As we can see in the preceding figure, the distribution adopts several *shapes*, including a uniform distribution, Gaussian-like distributions, U-like distributions, and so on. A third reason is that 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 the given likelihood, returns a posterior with the same functional form as the prior. Untwisting the tongue, every time we use a beta distribution as prior and a binomial distribution as likelihood, we will get a beta as a posterior. There are other pairs of conjugate priors, for example, the Gaussian distribution is the conjugate prior of itself. For a more detailed discussion read `https://en.wikipedia.org/wiki/Conjugate_prior`. 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 to end up with a posterior we cannot solve analytically. This was a deal breaker before the development of suitable computational methods to solve any possible posterior. From the next chapter on, 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 Bayes' theorem says that the posterior is proportional to the likelihood times the prior:

$$p(\theta \mid y) \propto p(y \mid \theta)\, p(\theta)$$

So for our problem, we have to multiply the binomial and the beta distributions:

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

Now let's simplify this expression. To our practical concerns we can drop all the terms that do not depend on $\theta$ and our results will still be valid. So we can write the following:

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

Reordering it, we get the following:

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

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

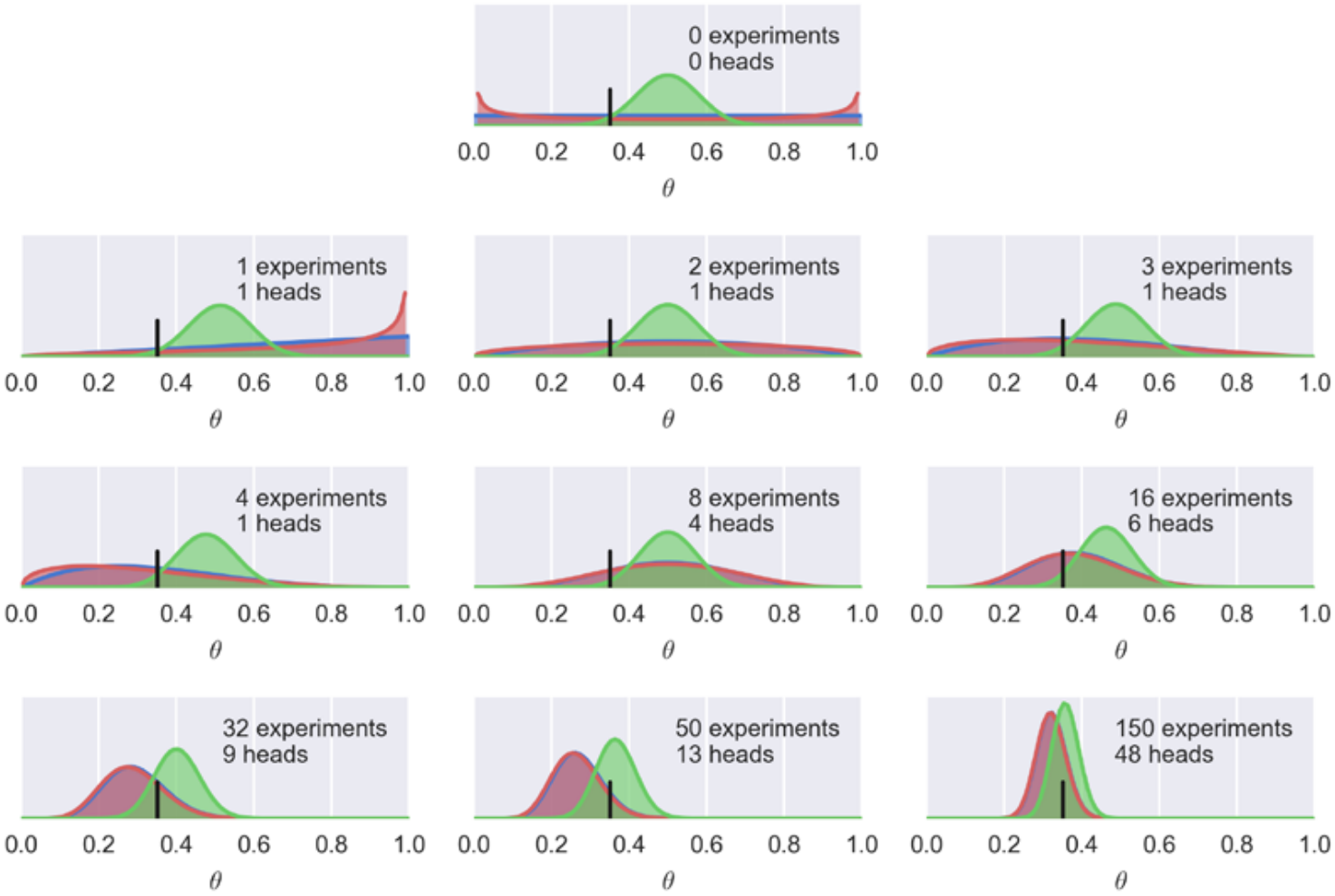
$$p(\theta \mid y) = Beta(\alpha_{prior} + y, \beta_{prior} + N - y)$$

# Computing and plotting the posterior

Now that we have the analytical expression for the posterior, let's use Python to compute it and plot the results. In the following code you will see there is actually one line that computes the results while the others are there just to plot them:

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

beta_params = [(1, 1), (0.5, 0.5), (20, 20)]
```

```
dist = stats.beta
x = np.linspace(0, 1, 100)

for idx, N in enumerate(trials):
    if idx == 0:
        plt.subplot(4,3, 2)
    else:
        plt.subplot(4,3, idx+3)
    y = data[idx]
    for (a_prior, b_prior), c in zip(beta_params, ('b', 'r', 'g')):
        p_theta_given_y = dist.pdf(x, a_prior + y, b_prior + N - y)
        plt.plot(x, p_theta_given_y, c)
        plt.fill_between(x, 0, p_theta_given_y, color=c, alpha=0.6)

    plt.axvline(theta_real, ymax=0.3, color='k')
    plt.plot(0, 0, label="{:d} experiments\n{:d} heads".format(N,
     y), alpha=0)
    plt.xlim(0,1)
    plt.ylim(0,12)
    plt.xlabel(r'$\theta$')
    plt.legend()
    plt.gca().axes.get_yaxis().set_visible(False)
plt.tight_layout()
```



On the first line we have **0 experiments** done, hence these curves are just our priors. We have three curves, one per prior:

- The blue one is a uniform prior. This is equivalent to saying that all the possible values for the bias are equally probable a priori.
- The red one is similar to the uniform. For the sake of this example we will just say that we are a little bit more confident that the bias is either *0* or *1* than the rest of the values.
- The green and last one 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 such a word is commonly used in Bayesian discussions we think is better to talk about models that are informed by data.

The rest of the subplots show posteriors $p(\theta \mid y)$ for successive experiments. Remember that we can think of posteriors as updated priors given the data. The number of experiments (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 $\theta$. 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 let's take a moment to understand it:

- The result of a Bayesian analysis is the 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 the distribution, the less certain we are.
- Even when $\frac{1}{2} = \frac{4}{8} = 0.5$ it is easy to see that the uncertainty we have in the first example is larger than in the second one because we have more data that supports our inference. This intuition is reflected in the posterior.
- 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, we will always get the same posterior. Remember that infinite is a limit and not a number, so from a practical point of view in some cases the infinite amount of data could be approximated with a really small number of data points.

- How fast posteriors converge to the same distribution depends on the data and the model. In the previous figure we can see that the blue and red posteriors look almost indistinguishable after only *8* experiments, while the red curve continues to be separated from the other two even after *150* experiments.
- 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, also leads to a natural way of updating our estimations when we get new data, a situation common in many data analysis problems.

# Influence of the prior and how to choose one

From the preceding example, it is clear that priors influence the result of the analysis. This is totally fine, priors are supposed to do this. Newcomers to Bayesian analysis (as well as detractors of this paradigm) are in general 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 okay, but we have to remember that data does not really speak; at best, data murmurs. Data only makes sense in the light 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. Check for example a recent *experiment* that appeared in the New York Times `http://www.nytimes.com/interactive/2016/09/20/upshot/the-error-the-polling-world-rarely-talks-about.html?_r=0`.

Some people fancy 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 if we expect the value to be close to zero or above/below some value. In such cases, we can use priors to put some weak information in our models without being afraid of being too pushy with our data. Because these priors work to keep the posterior distribution approximately within certain reasonable bounds, they are also know as regularizing priors.

Of course, it can also be possible to use informative priors. These 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 all the prior information they can get, in Bayesian and non-Bayesian ways, 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 if you have reliable prior information, there is no reason to discard that information, including the non-nonsensical argument that not using information we trust is objective. Imagine if every time an automotive engineer has to design a new car, she has to start from scratch and re-invent the combustion engine, the wheel, and for that matter, the whole concept of a car. That is not the way things work.

Now we know that there are different kind of priors, but this probably doesn't make us less nervous about choosing among them. Maybe it would be better to not have priors at all. That would make things easier. Well, every model, Bayesian or not has some kind of priors in some way or another, even if the prior does not appear explicitly. In fact many results from frequentist statistics can be seen as special cases of a Bayesian model under certain circumstances, such as flat priors. Let's pay attention to the previous figure one more time. We can see that the mode (the peak of the posterior) of the blue posterior agrees with the expected value for $\theta$ from a frequentist analysis:

$$\hat{\theta} = \frac{y}{N}$$

Notice that $\hat{\theta}$ is a point estimate (a number) and not a posterior distribution (or any other type of distribution for that matter). So notice that you can not really avoid priors, but if you include them in your analysis you will get 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 more easy 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 people you do not even know. Reproducibility matters and transparent assumptions in a model contributes to it.

We are free to use more than one prior (or likelihood) for a given analysis if we are not sure about any special one. Part of the modeling process is about questioning assumptions, and priors are just that. Different assumptions will lead to different models, using data and our domain knowledge of the problem we will be able to compare models. *Chapter 06*, *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

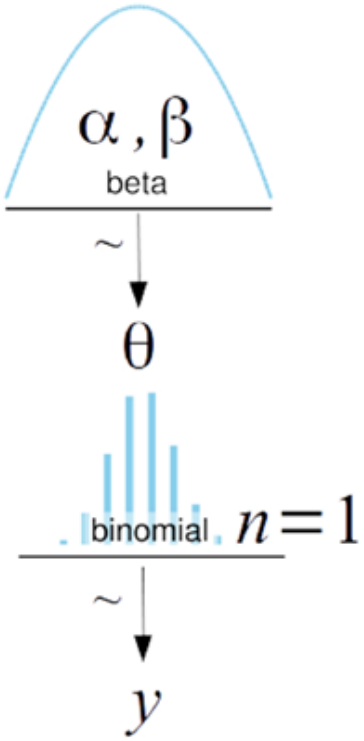
Now that we have the posterior, the analysis is finished and we can go home. Well, not yet! We probably need to communicate or summarize the results to others, or even record for later use by ourselves.

# Model notation and visualization

If you want to communicate the result, you may need, depending on your audience, to also communicate the model. A common notation to succinctly represent probabilistic models is as follows:

- $\theta \sim Beta(a, \beta)$
- $y \sim Bin(n = 1, p = \theta)$

This is the model we use for the coin-flip example. As we may remember, the symbol ~ indicates that the variable is a random variable distributed according to the distribution on the right, that is, $\theta$ is distributed as a beta distribution with parameters $\alpha$ and $\beta$, and $y$ is distributed as a binomial with parameter $n=1$ and $p = \theta$. The very same model can be represented graphically using Kruschke's diagrams:



On the first level, we have the prior that generates the values for $\theta$, then the likelihood and, on the last line, the data. Arrows indicate the relationship between variables, and the ~ 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/`). I would like to specially thanks him for making these templates available.

# Summarizing the posterior

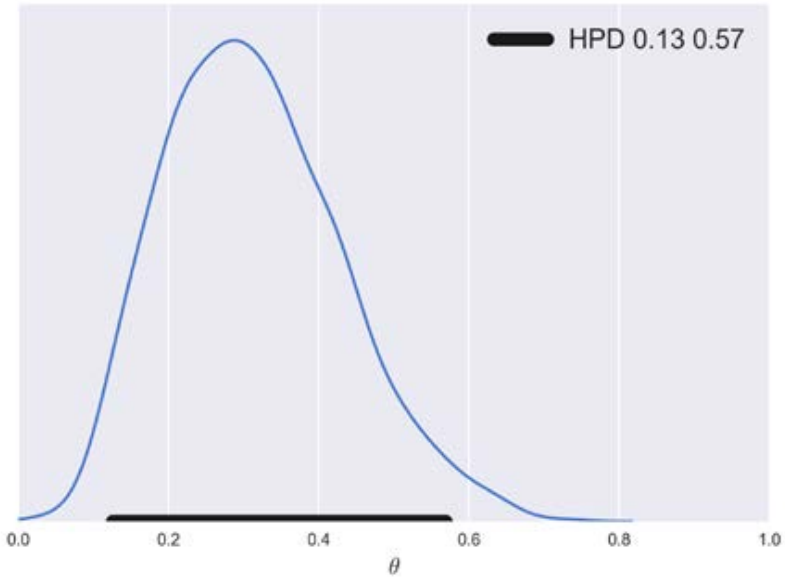
The result of a Bayesian analysis is the posterior distribution. This contains all the information about our parameters according to the data and the model. If possible, we can just show the posterior distribution to our audience. In general, it is also a good idea to report the mean (or mode or median) of the distribution 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 types of distributions, such as skewed ones. So intead, we could use the following approach.

## 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 or *98%* 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 *0.95* probability. This is a very intuitive interpretation, to the point that often people misinterpret frequentist confidence intervals as if they were Bayesian credible intervals. If you are familiar with the frequentist paradigm please note that both type of intervals have different interpretations. Performing a fully Bayesian analysis enables us to talk about the probability of a parameter having some value. While 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. Another word of caution before we continue: there is nothing special about choosing *95%* or *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, it's OK; just remember that this is just a default value and any justification of which value we should use will be always context-dependent and not automatic.

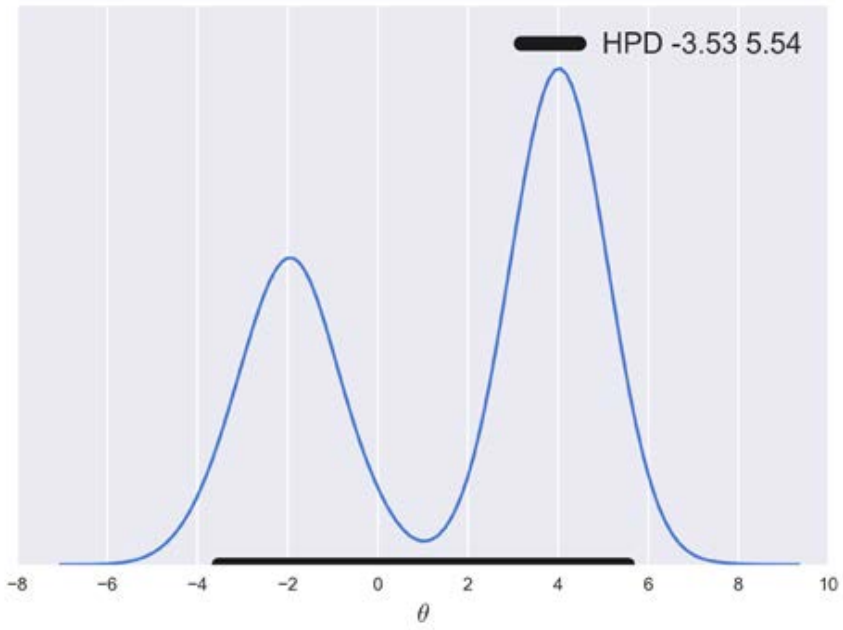
Computing the *95%* HPD for a unimodal distribution is easy, since it is defined by the percentiles `2.5` and `97.5`:

```
def naive_hpd(post):
    sns.kdeplot(post)
    HPD = np.percentile(post, [2.5, 97.5])
    plt.plot(HPD, [0, 0], label='HPD {:.2f} {:.2f}'.format(*HPD),
      linewidth=8, color='k')
    plt.legend(fontsize=16);
    plt.xlabel(r'$\theta$', fontsize=14)
    plt.gca().axes.get_yaxis().set_ticks([])


np.random.seed(1)
post = stats.beta.rvs(5, 11, size=1000)
naive_hpd(post)
plt.xlim(0, 1)
```



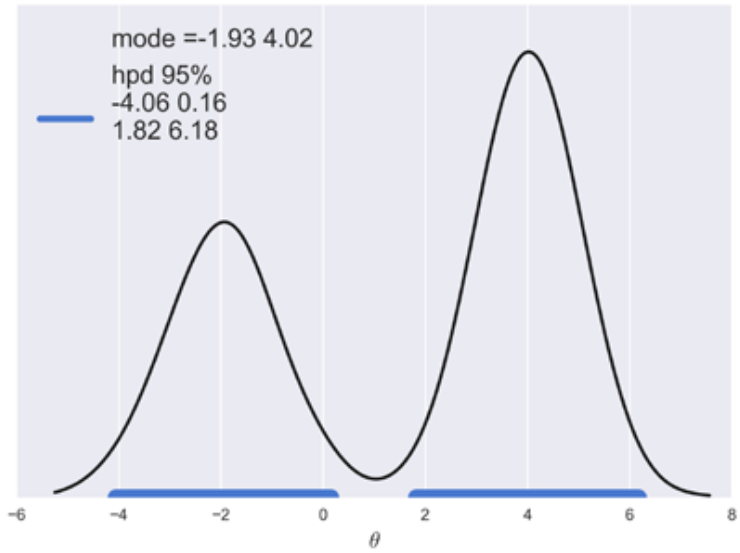
For a multi-modal distribution, the computation of the HPD is a little bit more complicated. If we apply our naive definition of the HPD to a mixture of Gaussians we will get the following:

```
np.random.seed(1)
gauss_a = stats.norm.rvs(loc=4, scale=0.9, size=3000)
gauss_b = stats.norm.rvs(loc=-2, scale=1, size=2000)
mix_norm = np.concatenate((gauss_a, gauss_b))

naive_hpd(mix_norm)
```




As we can see in the preceding figure, the HPD computed in the naive way includes values with a low probability, approximately between *[0, 2]*. To compute the HPD in the correct way we will use the function `plot_post`, which you can download from the accompanying code that comes with the book:

```
from plot_post import plot_post
plot_post(mix_norm, roundto=2, alpha=0.05)
plt.legend(loc=0, fontsize=16)
plt.xlabel(r"$\theta$", fontsize=14)
```




As you can see from the preceding figure, the *95% HPD* is composed of two intervals. `plot_post` also returns the values for the two modes.

# 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 to generate future data $y$, that is, predictions. Posterior predictive checks consist of comparing the observed data and the predicted data to spot differences between these two sets. 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 if we did not make any mistake, differences could arise. Trying to understand the mismatch could lead us to improve models or at least to understand their limitations. Knowing which part 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 well the mean behavior of our data 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 okay for us. The general aim will be not to declare that a model is false; instead we follow George Box's advice, all models are wrong, but some are useful. 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 models have a weaker epistemological status. Nevertheless, independently 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.

# Installing the necessary Python packages

The code in the book was written using Python version 3.5, and it is recommended you use the most recent version of Python 3 that is currently available, although most of the code examples may also run for older versions of Python, including Python 2.7, but code could need minor adjustments.

The recommended way to install Python and Python libraries is using Anaconda, a scientific computing distribution. You can read more about Anaconda and download it from `https://www.continuum.io/downloads`. Once Anaconda is in our system, we can install new Python packages with the following command:

```
conda install NamePackage
```

We will use the following Python packages:

- `Ipython 5.0`
- `NumPy 1.11.1`
- `SciPy 0.18.1`
- `Pandas 0.18.1`
- `Matplotlib 1.5.3`
- `Seaborn 0.7.1`
- `PyMC3 3.0`

To install the latest stable version of PyMC3, run the following command on a command-line terminal:

```
pip install pymc3
```

# Summary

We began our Bayesian journey with a very brief discussion about statistical modeling, probability theory and an introduction of the Bayes' theorem. We then use the coin-tossing 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 we must decide when doing data analysis, such as other parts of the model like the likelihood, or even more meta questions like why are we trying to solve a particular problem in the first place. We ended the chapter discussing the interpretation and communication of the results of a Bayesian analysis. In this chapter we have briefly summarized the main aspects of doing Bayesian data analysis. Throughout the rest of the 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 focus on computational techniques to build and analyze more complex models and we will introduce PyMC3 a Python library that we will use to implement and analyze all our Bayesian models.

# Exercises

We don't know if 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. Although you may disagree with this statement given our record as a species on wars, economic-systems that prioritize profit and not people's wellbeing, and other atrocities. Anyway, I strongly recommend you to do the proposed exercises at the end of each chapter:

1. Modify the code that generated *figure 3* 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.
2. Try reploting *figure 3* using other priors (`beta_params`) and other data (*trials* and *data*).
3. Read about Cromwell's rule at Wikipedia `https://en.wikipedia.org/wiki/Cromwell%27s_rule`.
4. Explore different parameters for the Gaussian, binomial and beta plots. Alternatively, you may want to plot a single distribution instead of a grid of distributions.
5. Read about probabilities and the Dutch book at Wikipedia `https://en.wikipedia.org/wiki/Dutch_book`.

# 2

# Programming Probabilistically – A PyMC3 Primer

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**. The main idea is that we are going to use code to describe our models and make inferences from them. It is not that we are too lazy to learn the mathematical way, nor are we elitist hardcore hackers—I-dream-in-code. One important reason behind this choice is that many models do not lead to a closed-form analytic posterior, that is, we can only compute those posteriors using numerical techniques. Another reason to learn probabilistic programing is that modern Bayesian statistics is done mainly 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 complex models and allows us to focus more on model design, evaluation, and interpretation, and less on mathematical or computational details.

In this chapter, we are going to learn about numerical methods used in Bayesian statistics and how to use PyMC3, a very flexible Python library for probabilistic programming. Knowing PyMC3 will also help us to learn more advanced Bayesian concepts in a more practical way.

In this chapter, we will cover the following topics:

- Probabilistic programming
- Inference engines
- PyMC3 primer
- The coin-flipping problem revisited
- Model checking and diagnoses

# Probabilistic programming

Bayesian statistics is conceptually very simple: we have some data that is fixed, in the sense that we cannot change what we have measured, and we have parameters whose values are of interest to us and hence we explore their plausible values. All the uncertainties we have are modeled using probabilities. In other statistical paradigms, there are different types of unknown quantities; in the Bayesian framework everything that is unknown is treated the same. If we do not know a quantity we assign a probability distribution to it. Then, Bayes' theorem is used to transform the prior probability distribution $p(\theta)$ (what we know about a given problem before observing the data), into a posterior distribution $p(\theta|D)$ (what we know after observing data). In other words, Bayesian statistics is a form of learning.

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 reasons that hindered the wide adoption of Bayesian methods. The arrival of the computational era and the development of numerical methods that can be applied to compute the posterior for almost any possible model has dramatically transformed Bayesian data analysis practices. We can think of these numerical methods as universal inference engines, because in principle, the inference part can be automated just by pushing the inference button, a term coined by Thomas Wiecki, one of the core developers of PyMC3.

The possibility of automating the inference part has led to the development of **probabilistic programming languages** (**PPL**) that allow 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 the programming languages can have in scientific computing is the introduction of the Fortran programming language more than 6 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, probabilistic programming hides from the user details on how probabilities are manipulated and how the inference is performed, allowing users to focus on model specification and analysis of results.

# Inference engines

There are several methods to compute the posterior even when it is not possible to solve it analytically. Some of the methods are:

- Non-Markovian methods:
    - Grid computing
    - Quadratic approximation
    - Variational methods
- Markovian methods:
    - Metropolis-Hastings
    - Hamiltonian Monte Carlo/No U-Turn Sampler

Nowadays, Bayesian analysis is performed mainly by using **Markov Chain Monte Carlo** (**MCMC**) methods, with variational methods gaining momentum for bigger datasets. We do not need to really understand these methods to perform Bayesian analysis, that's the whole point of probabilistic programming languages, but knowing at least how they work at a conceptual level is often very useful, for example for debugging our models.

# Non-Markovian methods

Let's start our discussion of inference engines with the non-Markovian methods. These methods are in general faster than Markovian ones. For some problems they are very useful on their own, and for other types of problems they only provide a crude approximation of the true posterior, but nevertheless they can be used to provide a reasonable starting point for Markovian methods. The meaning of Markovian will be explained later.

## Grid computing

Grid computing is a 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 for a given number of points. 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 we multiply the likelihood and the prior.

Optionally, we may normalize the computed values (divide the result at each point by the sum of all points).

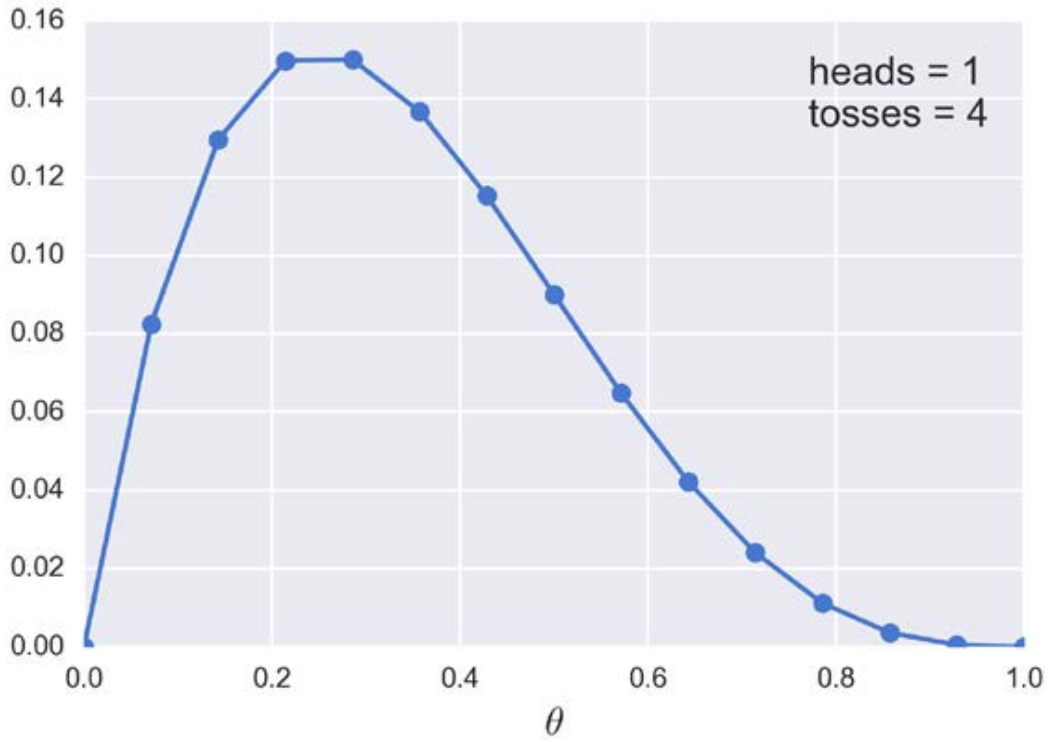
Is easy to see that a larger number of points (or equivalently a reduced size of the grid) will result in a better approximation. In fact, if we take an infinite number of points we will get the exact posterior. The grid approach does not scale well for many parameters (also referred as dimensions); as you increase the number of parameters the *volume* of the posterior gets relatively smaller compared with the sampled *volume*. In other words, we will spend most of the time computing values with an almost null contribution to the posterior, making this approach unfeasible for many statistical and data science problems.

The following code implements the grid approach to solve the coin-flipping problem we began to explore in *Chapter 1, Thinking Probabilistically - A Bayesian Inference Primer*:

```
def posterior_grid(grid_points=100, heads=6, tosses=9):
    """
    A grid implementation for the coin-flip problem
    """
    grid = np.linspace(0, 1, grid_points)
    prior = np.repeat(5, grid_points)
    likelihood = stats.binom.pmf(heads, tosses, grid)
    unstd_posterior = likelihood * prior
    posterior = unstd_posterior / unstd_posterior.sum()
    return grid, posterior
```

Assuming we made `4` tosses and we observe only `1` head we have the following:

```
points = 15
h, n = 1, 4
grid, posterior = posterior_grid_approx(points, h, n)
plt.plot(grid, posterior, 'o-', label='heads = {}\ntosses =
  {}'.format(h, n))
plt.xlabel(r'$\theta$')
plt.legend(loc=0)
```




## Quadratic method

The quadratic approximation, also known as the **Laplace method** or the normal approximation, consists of approximating the posterior with a Gaussian distribution. This method often works because in general the region close to the mode of the posterior distribution is more or less normal, and in fact in many cases is actually a Gaussian distribution. This method consists of two steps. First, find the mode of the posterior distribution. This can be done using optimization methods; that is, methods to find the maximum or minimum of a function, and there are many off-the-shelf methods for this purpose. This will be the mean of the approximating Gaussian. Then we can estimate the curvature of the function near the mode. Based on this curvature, the standard deviation of the approximating Gaussian can be computed. We are going to apply this method once we have introduced PyMC3.

## 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 and they do not necessarily parallelize well. The naive approach is to simply run $n$ chains in parallel and then combine the results, but for many problems this is not a really good solution. Finding effective ways of parallelizing them is an active research area.

Variational methods could be a better choice for large datasets (think big data) and/or for likelihoods that are too expensive to compute. In addition, these methods are useful for quick approximations to the posterior and as starting points for MCMC methods.

The general idea of variational methods is to approximate the posterior with a simpler distribution; this may sound similar to the Laplace approximation, but the similarities vanish when we check the details of the method. The main drawback of variational methods is that we must come up with a specific algorithm for each model, so it is not really a universal inference engine, but a model-specific one.

Of course, lots of people have tried to automatize variational methods. A recently proposed method is the **automatic differentiation variational inference** (**ADVI**) (see `http://arxiv.org/abs/1603.00788`). At the conceptual level, ADVI works in the following way:

1. Transform the parameters to make them live in the real line. For example, taking the logarithm of a parameter restricted to positive values we obtain an unbounded parameter on the interval $[-\infty, \infty]$.
2. Approximate the unbounded parameters with a Gaussian distribution. Notice that a Gaussian on the transformed parameter space is non-Gaussian on the original parameter space, hence this is not the same as the Laplace approximation.
3. Use an optimization method to make the Gaussian approximation as close as possible to the posterior. This is done by maximizing a quantity known as the **Evidence Lower Bound** (**ELBO**). How we measure the similarity of two distributions and what ELBO is exactly, at this point, is a mathematical detail.

ADVI is already implemented on PyMC3 and we will use it later in the book.

# Markovian methods

There is a family of related methods collectively know as MCMC methods. As with the grid computing approximation, we need to be able to compute the likelihood and prior for a given point and we want to approximate the whole posterior distribution. MCMC methods outperform the grid approximation because they are designed to spend more time in higher probability regions than in lower ones. In fact, a MCMC method will visit different regions of the parameter space in accordance with their relative probabilities. If region *A* is twice as likely as region *B*, then we are going to get twice the 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, and the larger the sample size the better the results.

What is 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 very famous casino located in the Principality of Monaco. 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, or by simulating them. In fact, as the story goes, the motivation was to answer questions about the probability of getting a particular hand in a solitary game. One way to solve this problem was to follow the analytical combinatorial problem. Another way, Stan argued, was to play several games of solitaire and just count how many of the hands we play match the particular hand we are interested in! Maybe this sounds obvious, or at least pretty reasonable; for example, you may have used re-sampling methods to solve your 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 method was to solve a problem of nuclear physics, a problem really hard to tackle using the conventional tools at that time. Nowadays, even personal computers are powerful enough to solve many interesting problems using the Monte Carlo approach and hence these methods are applied to a wide variety of problems in science, engineering, industry, arts, and so on.

A classic pedagogical example of using a Monte Carlo method to compute a quantity of interest is the numerical estimation of $\pi$. In practice there are better methods for this particular computation, but its pedagocial value still remains. We can estimate the value of $\pi$ 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 $\frac{4 \times inside}{N}$.

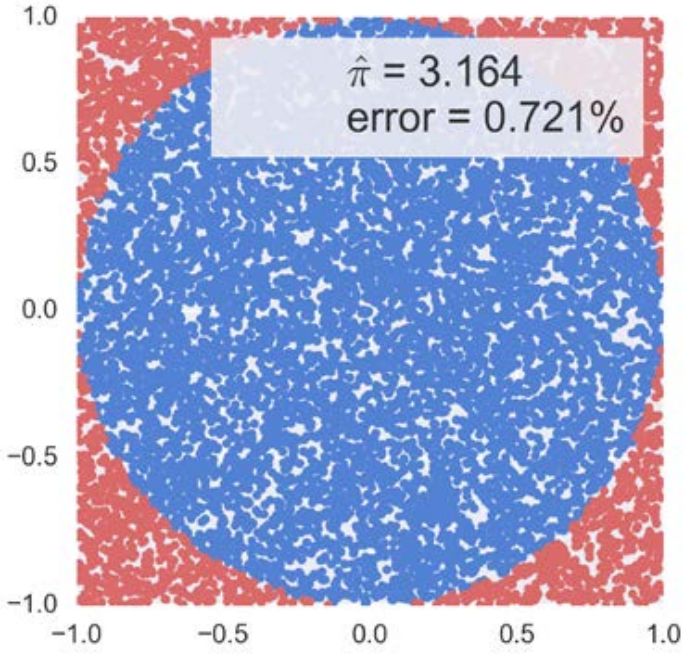
A couple of notes: We know a point is inside a circle if the following relation is true:

$$\sqrt{\left(x^2 + y^2\right)} \leq R$$

The area of the square is $(2R)^2$ and the area of the circle is $\pi R^2$. Thus we know that the ratio of the area of the square to the area of the circle is $\frac{4}{\pi}$, and the area of the circle and squares are proportional to the number of points inside the circle and the total $N$ points, respectively.

Using a few lines of Python we can run this simple Monte Carlo simulation and compute $\pi$ and also the relative error of our estimate compared to the true value of $\pi$:

```
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.plot(x[inside], y[inside], 'b.')
plt.plot(x[outside], y[outside], 'r.')
plt.plot(0, 0, label='$\hat \pi$ = {:4.3f}\nerror = {:4.3f}%'.
format(pi, error), alpha=0)
plt.axis('square')
plt.legend(frameon=True, framealpha=0.9, fontsize=16);
```



In the preceding code we can see that the `outside` variable is only used to get the plot; we do not need it for computing $\frac{4 \times inside}{N}$. Another clarification: because our computation is restricted to the unit circle we can omit computing the square root from the computation of the inside variable.

## Markov chain

A Markov chain is a mathematical object consisting of a sequence of states and a set of probabilities describing the transitions among those states. A chain is Markovian if the probability of moving to other states 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 following 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 becomes just 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 **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 towards state *i*.

In summary, all this means that if we manage to create a Markov Chain satisfying 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! The most popular method that guarantees detailed balance is the **Metropolis-Hasting algorithm**.

## Metropolis-Hastings

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 to the bottom, and the lake is really big, so a grid approximation does not seem like a very good idea. In order to develop a sampling strategy, we seek help from two of our best friends, Markovia and Monty. After some discussion they come up with the following algorithm that requires a boat; nothing fancy, we can even use a wooden raft, and a very long stick. This is cheaper than a sonar and we have already spent all our money on the boat, anyway!

1. Initialize the measuring by choosing 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 some other point and take a new measurement.

4. Compare the two measures in the following way:
    - If the new spot is deeper than the old one, write down in your notebook the depth of the new spot and repeat from 2.
    - If the spot is shallower than the old one, we have two options: to accept or reject. Accepting means to write down the depth of the new spot and repeat from 2. Rejecting means to go back to the old spot and write down (again) the value for the depth of the old spot.

How do we decide to accept or reject a new spot? Well, the trick is to apply the Metropolis-Hastings criteria. This means to 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, but we will also get an approximation of the entire curvature of the bottom of the lake. As you may have already 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, 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 relatively small number of samples.

Let's look at the method now in a little bit more formal way. For some distributions, like the Gaussian, we have very efficient algorithms to get samples from, but for some other distributions such as many of the posterior distributions, we are going to find this is not the case. Metropolis-Hastings enables us to obtain samples from any distribution with probability *p(x)* given that we can compute at least a value proportional to it. This is very useful since in a lot of problems like Bayesian statistics the hard part is to compute the normalization factor, the denominator of the Bayes' theorem. The Metropolis-Hastings algorithm has the following steps:

1. Choose an initial value for our parameter $x_i$. This can be done randomly or by using some educated guess.
2. We choose a new parameter value $x_{i+1}$, sampling from an easy-to-sample distribution such as a Gaussian or uniform distribution $Q(x_{i+1} \mid x_i)$. We can think of this step as perturbing the state $x_i$ somehow.
3. We compute the probability of accepting a new parameter value by using the Metropolis-Hastings criteria $p_a(x_{i+1} \mid x_i) = min\left(1, \frac{p(x_{i+1})}{p(x_i)} \frac{q(x_i \mid x_{i+1})}{q(x_{i+1} \mid x_i)}\right)$.

4. If the probability computed on 3 is larger than the value taken from a uniform distribution on the interval *[0, 1]* we accept the new state, otherwise we stay in the old state.
5. We iterate from 2 until we have enough samples. Later we will see what *enough* means.

A couple of things to take into account:

- If the proposal distribution $Q(x_{i+1} \mid x_i)$ is symmetric we get $p_a(x_{i+1} \mid x_i) = min\left(1, \frac{p(x_{i+1})}{p(x_i)}\right)$, often referred to as **Metropolis criteria** (we drop the **Hastings** part).
- Steps 3 and 4 imply that we will always accept or move to a most probable state, to a most probable parameter value. 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 criteria for accepting 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 saving the sampled (or visited) parameter values. We save a sampled value $x_{i+1}$ if we accept moving to a new state $x_{i+1}$. If we reject moving to $x_{i+1}$, we save the value of $x_i$.

At the end of the process we will have a list of values sometimes refereed to as a **sample chain** or **trace**. 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 simple. We have effectively transformed integrals (of the posterior) into just summing values in our vector of sampled values.

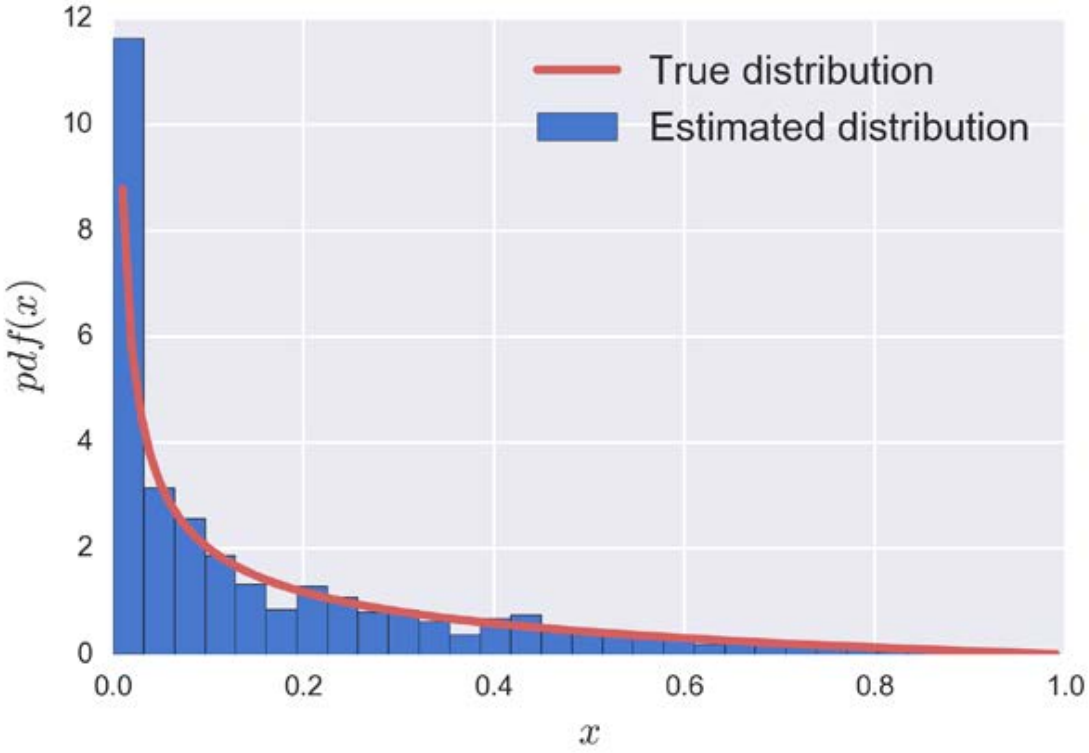
The following code illustrates a very basic implementation of the Metropolis algorithm. Is not meant to solve any real problem, only to show it is possible to sample from a function if we know how to compute its value at a given point. Notice also that the following implementation has nothing Bayesian in it; there is no prior and we do not even have data! Remember that the MCMC methods are very general algorithms that can be applied to a broad array of problems. For example, in a (non-Bayesian) molecular model, instead of `func.pdf(x)` we would have a function computing the energy of the system for the state *x*.

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, steps=10000):
    """A very simple Metropolis implementation"""
    samples = np.zeros(steps)
    old_x = func.mean()
    old_prob = func.pdf(old_x)

    for i in range(steps):
        new_x = old_x + np.random.normal(0, 0.5)
        new_prob = func.pdf(new_x)
        acceptance = new_prob/old_prob
        if acceptance >= np.random.random():
            samples[i] = new_x
            old_x = new_x
            old_prob = new_prob
        else:
            samples[i] = old_x
    return samples
```

In the next example we have defined `func` as a `beta` function, simply because is 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 a red line:

```
func = stats.beta(0.4, 2)
samples = metropolis(func=func)
x = np.linspace(0.01, .99, 100)
y = func.pdf(x)
plt.xlim(0, 1)
plt.plot(x, y, 'r-', lw=3, label='True distribution')
plt.hist(samples, bins=30, normed=True, label='Estimated
  distribution')
plt.xlabel('$x$', fontsize=14)
plt.ylabel('$pdf(x)$', fontsize=14)
plt.legend(fontsize=14)
```




At this point I hope you have a good conceptual grasp of the Metropolis-Hastings method. You may need to go back and read the previous pages, that's totally fine. I also strongly recommend that you check this post at `http://twiecki.github.io/blog/2015/11/10/mcmc-sampling/` one of the core developers of PyMC3. He has a simple example of the implementation of the `metropolis` method to compute a posterior distribution, including a very nice visualization of the sampling process and a brief a discussion on how the width of the proposal distribution affects the results.

# Hamiltonian Monte Carlo/NUTS

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. 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 is 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 that instead of proposing random displacements of our *boat* we do something more clever instead; we move the boat following the curvature of the lake's bottom. Why is this clever? Because in doing so we try to avoid one of the main problems of Metropolis-Hastings: the exploration is slow and samples tend to be autocorrelated, since most of the proposed moves are rejected.

So, how can we try to understand this method without going into mathematical details? Imagine we are in the lake with our boat. In order to decide where to move next we let a ball roll at the bottom of the lake, starting from our current position. Remember that this method was brought to us by the same people that treat horses as spheres, so our our ball is not only perfectly spherical, it also has no friction and thus is not slowed down by the water or mud. Well, we throw a ball and we let it roll for a short moment, and then we move the boat to where the ball is. Now we accept or reject this step using the Metropolis criteria just as we saw with the Metropolis-Hastings method. The whole procedure is repeated a number of times. This modified procedure has a higher chance of accepting new positions, even if they are far away relative to the previous position.

Out of our *Gedanken experiment* and back to the real world, the price we pay for this much cleverer Hamiltonian-based proposal is that we need to compute gradients of our function. A gradient is just a generalization of the concept of derivative to more than one dimension. We can use gradient information to simulate the ball moving in a curved space. So, we are faced with a trade-off; each HMC step is more expensive to compute than a Metropolis-Hastings one but the probability of accepting that step is much higher with HMC than with Metropolis. For many problems, this compromise turns in favor of the HMC method, especially for complex ones. Another drawback with HMC methods is that to have really good sampling we need to specify a couple of parameters. When done by hand it takes some trial and error and also requires experience from the user, making this procedure a less universal inference engine than we may want. Luckily for us, PyMC3 comes with a relatively new method known as **No-U-Turn Sampler** (**NUTS**). This method has proven very useful in providing the sampling efficiency of HMC methods, but without the need to manually adjust any knob.

## Other MCMC methods

There are plenty of MCMC methods out there and indeed people keep proposing new methods, so if you think you can improve sampling methods there is a wide range of persons that will be interested in your ideas. Mentioning all of them and their advantages and drawbacks is completely out of the scope of this book. Nevertheless, there are a few worth mentioning because you may hear people talk about them, so it is nice to at least have an idea of what are they talking about.

Another sampler that has been used extensively for molecular systems simulations is the **Replica Exchange** method, also known as **parallel tempering** or **Metropolis Coupled MCMC** (or $MC^3$; maybe that's too many *MCs*). The basic idea of this method is to simulate different replicas in parallel. Each replica follows the Metropolis-Hastings algorithm. The only difference between replicas is that the value of a parameter called temperature (physics influence once more time!) controls the probability of accepting less probable positions. From time to time, the method attempts a swap between replicas. The swapping is also accepted/rejected according to the Metropolis-Hastings criteria, but this time taking into account both replicas' temperatures. The swapping between chains can be attempted between random chains but it is generally preferable to do it for neighboring replicas; that is, replicas with similar temperatures and hence a higher probability-of-acceptance ratio. The intuition for this method is that as we increase the temperature the probability of accepting the new proposed position increases, and decreases with lower and lower temperatures. Replicas at higher temperatures explore the system more freely; for these replicas the surface becomes effectively flatter and thus easier to explore. For a replica with infinite temperature, all states are equally likely. The exchange between replicas avoids replicas at low temperatures getting trapped in local minima. This method is well suited for exploring systems with multiple minima.

# PyMC3 introduction

PyMC3 is a Python library for probabilistic programming. The last version at the moment of writing is 3.0.rc2 released on October 4th, 2016. 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 is written using Python, where the computationally demanding parts are written using NumPy and Theano. Theano is a Python library originally developed for deep learning that allows us to define, optimize, and evaluate mathematical expressions involving multidimensional arrays efficiently. The main reason PyMC3 uses Theano is because some of the sampling methods, like NUTS, need gradients to be computed and Theano knows how to do 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`.

# Coin-flipping, the computational approach

Let's revisit the coin-flipping problem, but this time using PyMC3. The first requirement is to get our data. We will use the same synthetic data as before. Since we are generating the data we know the value of $\theta$, `theta_real` in the following code. Of course, for a real dataset we would not have this knowledge and in fact this is what we want to estimate:

```
np.random.seed(123)
n_experiments = 4
theta_real = 0.35
data = stats.bernoulli.rvs(p=theta_real, size=n_experiments)
print(data)
array([1, 0, 0, 0])
```

# Model specification

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

$$\theta \sim Beta(\alpha, \beta)$$

$$y \sim Bin(n = 1, p = \theta)$$

This statistical model has an almost one-to-one translation to the PyMC3 syntax. The first line of the code creates a container for our first model, PyMC3 uses the `with` statement to indicate that everything inside the `with` block points to the same model. You can think of this as syntactic sugar to ease model specification. Imaginatively, our model is called `our_first_model`. The second line specifies the prior, and, as you can see, the syntax follows the mathematical notation closely. We call the random variable `theta`. Please note that this name matches the first argument of the PyMC3 `Beta` function; having both names the same is a good practice to avoid confusion. Then we will use the name of the variable to extract information from the sampled posterior. The variable `theta` is a stochastic variable; we can think of this variable as the rule to generate numbers from a given distribution (a beta distribution in this case) and not actual numbers. The third line specifies the likelihood following the same syntax as for the prior except that we pass the data using the `observed` argument. This is the way in which we tell PyMC3 that this is the likelihood. The data can be a Python list, a NumPy array or a Pandas DataFrame. That's all it takes to specify our model!

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

# Pushing the inference button

For this problem the posterior can be computed analytically and we can also take samples form the posterior using PyMC3 with just a few lines. At the first line we call `find_MAP`; this function calls optimization routines provided by SciPy and tries to return the Maximum a Posteriori (MAP). Calling `find_MAP` is optional; sometimes it works to provide a good starting point for the sampling method, sometimes it does not help too much, so often we can avoid it. Then, the next line is used to define the sampling method. Here we are using Metropolis-Hastings, simply called `Metropolis`. PyMC3 allows us to assign different samplers to different random variables; for now we have a model with only one parameter, but later we will have more variables. Alternatively, we can even omit this line and PyMC3 will assign samplers automatically to each variable based on properties of those variables. For example, NUTS works only for continuous variables and hence cannot be used with a discrete one, `Metropolis` can deal with discrete variables, and other types of variables have specially dedicated samplers. In general we should let PyMC3 choose the sampler for us. The last line performs the inference. The first argument is the number of samples we want, and the second and third arguments are the sampling method and the starting point. As we just saw, these arguments are optional:

```
start = pm.find_MAP()
step = pm.Metropolis()
trace = pm.sample(1000, step=step, start=start)
```

We have specified the model and done inference with just a few lines of code. Let's give a warm applause to the developers of PyMC3 for giving us this wonderful library!

# Diagnosing the sampling process

Since we are approximating the posterior with a finite number of samples, the first thing we need to do is to check if we have a reasonable approximation. There are several tests we can run, some are visual and some quantitative. These tests try to find problems with our samples but they cannot prove we have the correct distribution; they can only provide evidence that the sample seems reasonable. If we find problems with the sample, the solutions are:

- Increase the number of samples.

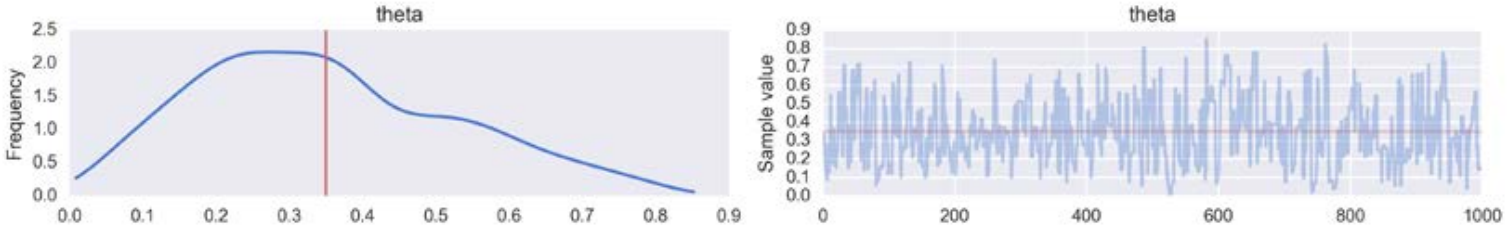
- Remove a number of samples from the beginning of the trace. This is know as **burn-in**. MCMC methods often take some time until we start getting samples from the target distribution. The burn-in will not be necessary for an infinite sample, as it is not part of the Markovian theory. Instead, removing the first samples is an ad hoc trick to get better results given that we are getting a finite sample. Remember we should not get confused by mixing mathematical objects with the approximation of those objects. Spheres, Gaussian, Markov chains, and all the mathematical objects live only in the platonic world of the ideas, not in our imperfect but real world.
- Re-parametrize the model, that is express the model in a different but equivalent way.
- Transform the data. This can help a lot to get a much more efficient sampling. When transforming the data we should take care to interpret the result in the transformed space or, alternatively, revert the transformation before interpreting the results.

We will explore all these options further throughout the book.

## Convergence

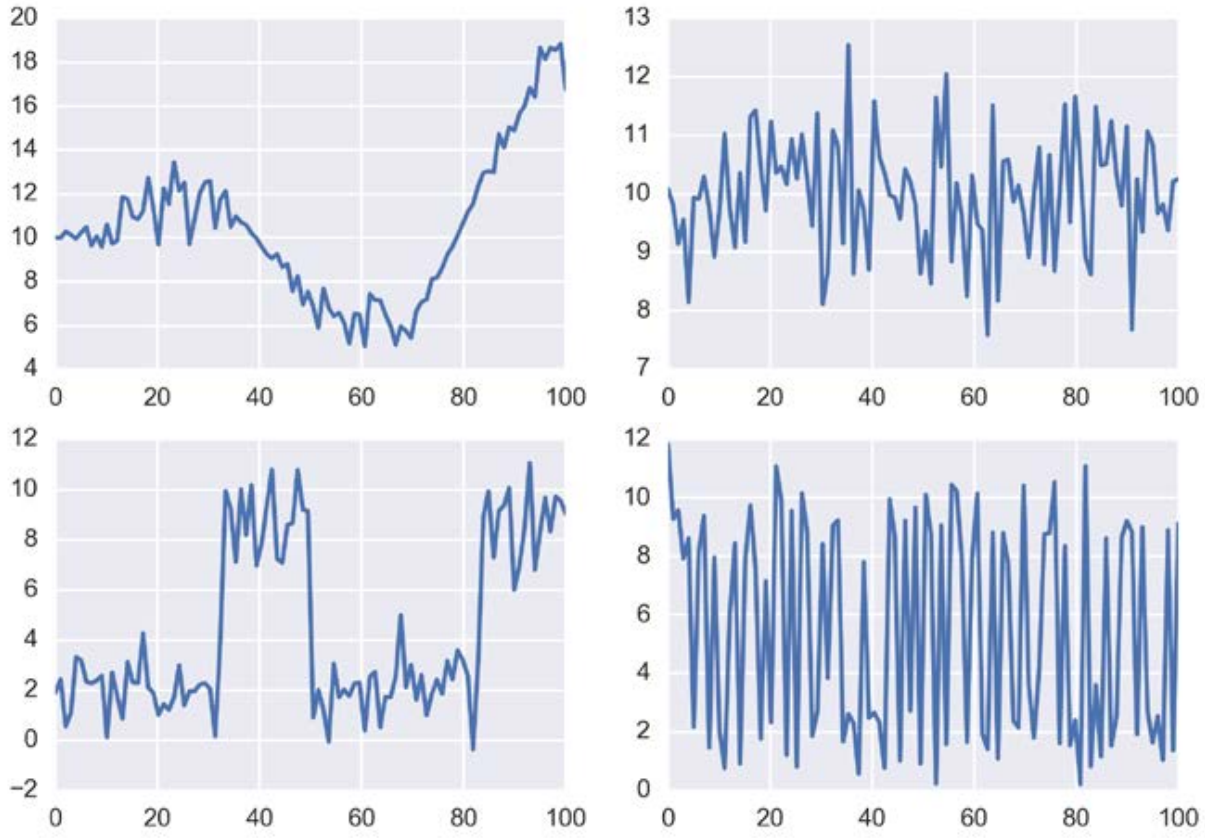
Generally, the first task we will perform is to check what the results look like. The `traceplot` function is ideally suited to this task:

```
burnin = 100
chain = trace[burnin:]
pm.traceplot(chain, lines={'theta':theta_real});
```




We get two plots for each unobserved variable. On the left, we get a **kernel density estimation** (**KDE**) plot; this is like the smoothed version of a histogram. On the right, we get the individual sampled values at each step during the sampling. Notice that the read line is indicating the value of the variable `theta_real`.

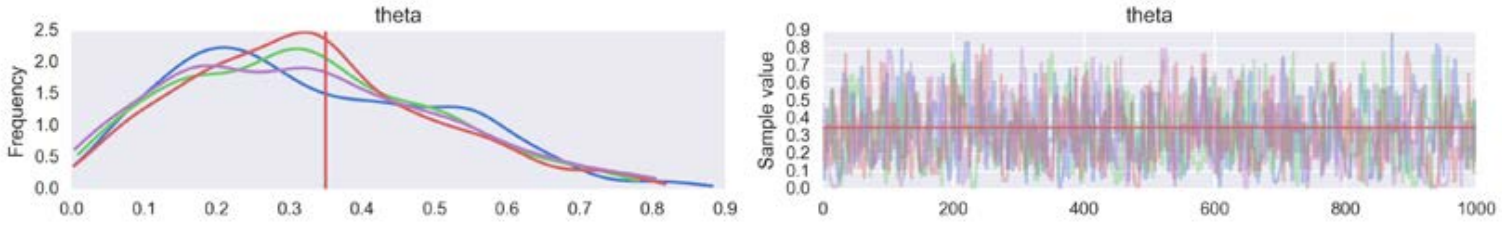
What do we need to look at when we see these plots? Well, KDE plots should look like smooth curves. Often, as the number of data increases, the distribution of each parameter will tend to become Gaussian-like; this is due to the law of the large numbers. Of course, this is not always true. The plot on the right should look like white noise; we are looking for good mixing. We should not see any recognizable pattern, we should not see a curve going up or down, instead we want a curve meandering around a single value. For multimodal distributions or discrete distributions we expect the curve to not spend too much time in a value or region before moving to other regions, we want to see sampled values moving freely among these regions. We should also expect to see a stable auto-similar trace, that is, a trace that at different points looks more or less the same; for example, the first *10%* (or so) should look similar to other portions in the trace like the last *50%* or *10%*. Once again, we do not want patterns; instead we expect something noisy. See the following figure for some examples of traces with good mixing (on the right) and bad mixing (on the left):




If the first part of the trace looks different than the others this is an indication of the need for burnin. If we see a lack of auto-similarity in other parts or we see some pattern this is an indication for more steps, or the need to use a different sampler or a different parametrization. For difficult models, we may apply a combination of all these strategies.

PyMC3 allows us to run a model several times in parallel and thus get a parallel chain for the same parameter. This is specified with the argument `njobs` in the `sample` function. Using `traceplot`, we plot all the chains for the same parameter in the same plot. Since each chain is independent of the others and each chain should be a good sample, all the chain should look similar to each other. Besides checking for convergence, these parallel chains can be used also for inference; instead of discarding the extra chains, we can combine them to increase the sample size:

```
with our_first_model:
    step = pm.Metropolis()
    multi_trace = pm.sample(1000, step=step, njobs=4)

burnin = 0
multi_chain = multi_trace[burnin:]
pm.traceplot(multi_chain, lines={'theta':theta_real});
```

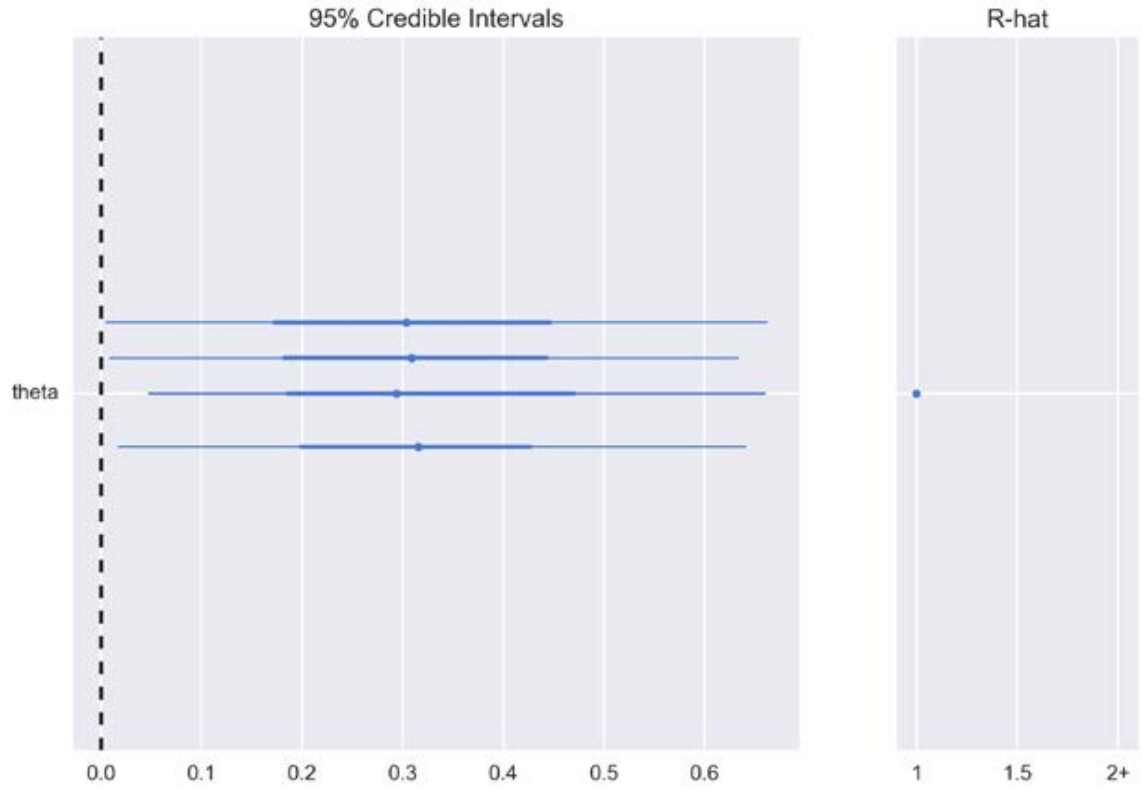



A quantitative way to check for convergence is by using the **Gelman-Rubin** test. The idea of this test is to compare the variance between chains with the variance within chains, so of course we need more than one chain for this test to work. Ideally, we should expect a value of $\hat{R} = 1$. As an empirical rule, we will be ok with a value below *1.1*; higher values are signaling a lack of convergence:

```
pm.gelman_rubin(multi_chain)
{'theta': 1.0074579751170656, 'theta_logodds': 1.009770031607315}
```

We can also visualize the $\hat{R}$ for every parameter together with the mean, *50%* HPD and *95%* HPD for each parameter distribution using the function `forestplot`:

```
pm.forestplot(multi_chain, varnames=['theta']);
```




The function `summary` provides a text-format summary of the posterior. We get the mean, standard deviation, and the HPD intervals:

```
pm.summary(multi_chain)
theta:
  Mean             SD               MC Error         95% HPD interval
  ------------------------------------------------------------------
  0.339            0.173            0.006             [0.037, 0.659]
  Posterior quantiles:
  2.5             25                                  50
75          97.5
  |--------------|==============|==============|--------------|
  0.063       0.206                           0.318
0.455       0.709
```

Alternatively, the function `df_summary`, returns a similar result but using a Pandas DataFrame:

```
pm.df_summary(multi_chain)
```

| | mean | sd | mc_error | hpd_2.5 | hpd_97.5 |
|---|---|---|---|---|---|
| theta | 0.33883 | 0.17305 | 0.00592 | 0.03681 | 0.65916 |

One of the quantities returned is the `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. The `mc_error` is the standard error of the means $\boldsymbol{x}$ of $n$ blocks, each block is just a portion of the trace:

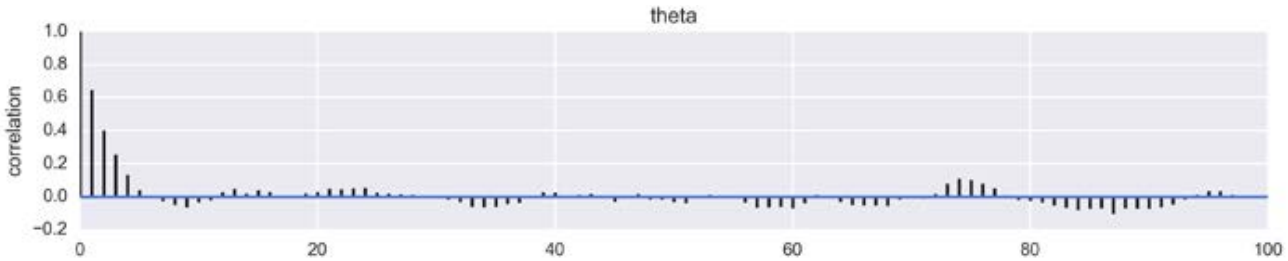
$$MC_{error} = \frac{\sigma(\boldsymbol{x})}{\sqrt{n}}$$

This error should be of course below the precision we want in our results. Since the sampling methods are stochastic, every time we re-run our models the values returned by `summary` or `df_summary` will be different; nevertheless, they should be similar for different runs. If they are not as similar as we want we may need more samples.

## Autocorrelation

An ideal sample will lack autocorrelation, that is, a value at one point should be independent of the values at other points. In practice, samples generated from MCMC methods, especially Metropolis-Hastings, can be autocorrelated. Some models will also lead to more autocorrelated samples due to correlations in the way one parameter depends on the others. PyMC3 comes with a convenient function to plot the autocorrelation:

```
pm.autocorrplot(chain)
```



The plot shows the average correlation of sample values compared to successive points (up to **100** points). Ideally we should see no autocorrelation, in practice; we seek samples that quickly drop to low values of autocorrelation. The more autocorrelated a parameter is, the larger the number of samples we will need to obtain a given precision; that is, autocorrelation has the detrimental effect of lowering the effective number of samples.

## Effective size

A sample with autocorrelation has less information than a sample of the same size without autocorrelation. Hence, given a sample of a certain size with a certain degree of autocorrelation we could try to estimate what will be the size of the sample with the same information without autocorrelation. That number will be the effective size of the sample. Ideally both quantities should be the same; the closer the two numbers the more efficient our sampling. The effective size of sample could serve us as a guide. If we want to estimate mean values of a distribution we will need an effective sample of at least *100* samples; if we want to estimate quantities that depend on the tails of distribution, such as the limits of credible intervals, we will need an effective size of *1000* to *10000* samples.

```
pm.effective_n(multi_chain)['theta']
667
```

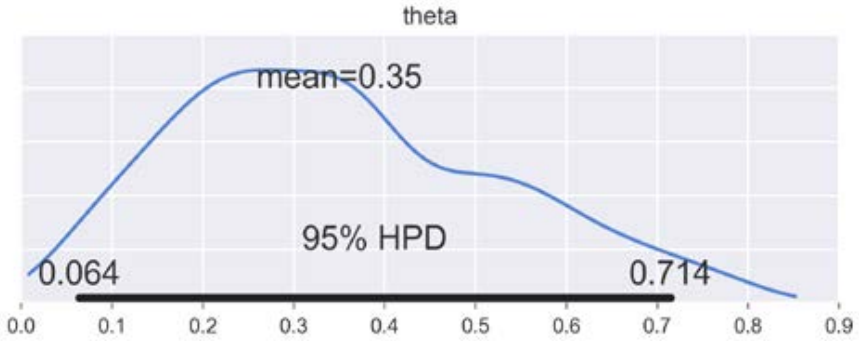
One way to have more efficient sampling is of course to use a better sampling method. An alternative is to transform the data or re-parametrize the model. Another commonly used option in the literature is to thin a chain. Thinning is just taking every k-esim observation. In Python, we would say taking slices of a chain. Thinning will indeed reduce the autocorrelation, but at the expense of reducing the number of samples. So in practice, it is generally a better idea to just increase the number of samples instead of doing thinning. Nonetheless, thinning can be useful, for example, to reduce storage requirements. When high autocorrelation cannot be avoided, we are obligated to compute long chains, and if our models contain many parameters storage can become problematic. Also we may need to do some post-processing of the posterior such as performing some expensive computation. In such cases having a smaller sample of minimally autocorrelated values could be important.

All the diagnostic tests we have seen have an empirical component and none of them is definitive. In practice, we run several tests and, if all of them look OK, then we proceed to further analyses. If we detect problems we have to go back and fix them; this is just part of the iterative process of modeling. It is also important to notice that having to run convergence tests is not really part of the Bayesian theory but is about the Bayesian practice, given that we are computing the posterior using numerical methods.

# Summarizing the posterior

As we have already seen, the result of a Bayesian analysis is a posterior distribution. This contains all the information about our parameters, according to the data and the model. One way to visually summarize the posterior is to use the `plot_posterior` function that comes with PyMC3. This function accepts a PyMC3 trace or a NumPy array as a main argument. By default, `plot_posterior` shows a histogram for the credible parameters together with the mean of the distribution and the *95%* HPD as a thick black line at the bottom of the plot. Different interval values can be set for the HPD with the argument `alpha_level`. We are going to refer to this type of plot as Kruschke's plot, since *John K. Kruschke* introduced this type of plot in his great book *Doing Bayesian Data Analysis*:

```
pm.plot_posterior(chain, kde_plot=True)
```




# 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 or no, contaminated or safe, and so on. Back to our problem, we may need to decide if the coin is fair or not fair. A fair coin is one with a $\theta$ value of exactly *0.5*. Strictly speaking, the chance of such a result is zero (think of an infinite number of trailing zeros), hence in practice we relax our definition of fairness and we will say a fair coin is one with a value of $\theta$ around *0.5*. What around exactly means 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.

Intuitively, one way to take such an informed decision is to compare the HPD to the value of interest, *0.5* in our case. In the preceding figure, we can see that the HPD goes from ~ *0.06* 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; maybe if we want a sharper decision we will need to collect more data to reduce the spread of the posterior or maybe we missed some important information that we could use to define a more informative prior.

# ROPE

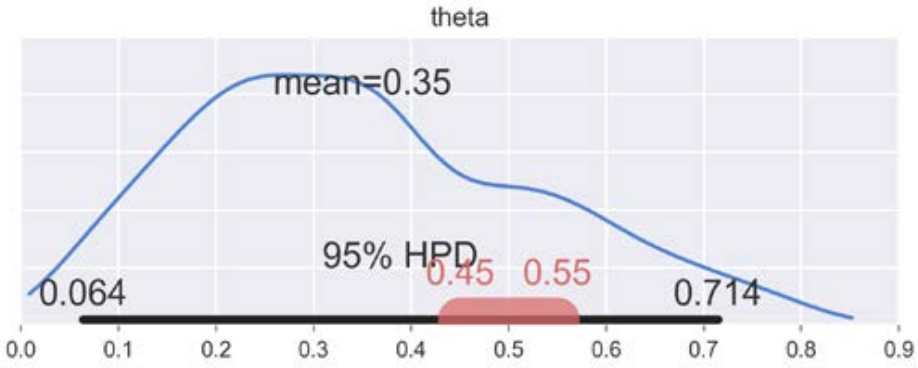
One possible option to take a posterior-based decision is to define a **Region Of Practical Equivalence** (**ROPE**). This is just an interval around the value of interest; for example, we could say that any value in the interval *[0.45, 0.55]* will be, for our purposes, practically equivalent to *0.5*. Once again the ROPE is context-dependent. So, now we are going to compare the ROPE to the HPD. We can define at least three scenarios:

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

Of course, if we choose a ROPE to cover the entire interval *[0, 1]*, we will always say we have a fair coin no matter what data we have but probably nobody is going to agree with our ROPE definition.

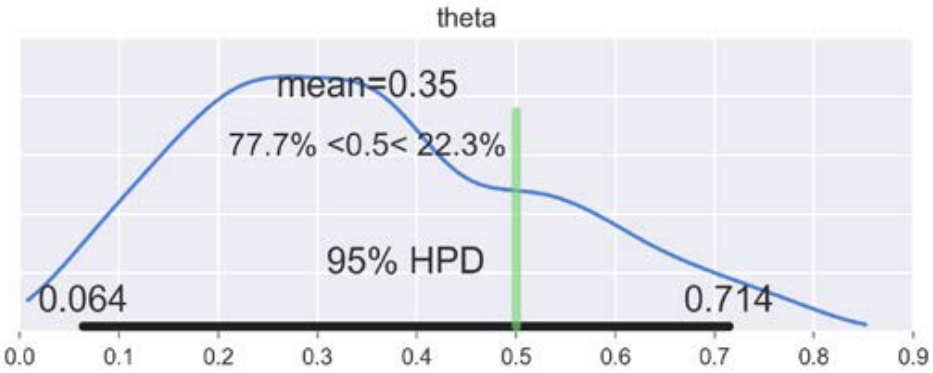
The `plot_posterior` function can be used to plot a ROPE. The ROPE appears as a semi-transparent red and very thick line, together with the overlayed limits of the ROPE:

```
pm.plot_posterior(chain, kde_plot=True, rope=[0.45, .55])
```



We can also pass to `plot_posterior` a reference value, for example `0.5`, that we want to compare with the posterior. We will get a green vertical line and the proportion of the posterior above and below our reference value:

```
pm.plot_posterior(chain, kde_plot=True, ref_val=0.5)
```




For a more detailed discussion of the use of the ROPE you could read chapter 12 of the great book *Doing Bayesian Data Analysis* by *John Kruschke* (I know I have already said this is a great book, but it is true!). 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 some parameter, but it is also important to take into account the cost of making a mistake. The benefit/cost trade-off can be mathematically formalized using cost functions, also know as loss functions. A loss function tries to capture the cost of predicting $X$ (the coin is fair) when $Y$ (the coin is not fair) turns out to be true. In many problems, the cost of making a decision is asymmetric. It is not the same to decide that it is safe not to administer a certain vaccine to children under five and being right, that being wrong. Making a bad decision could cost thousands of lives and produce a health crisis that could be avoided by administrator a relatively cheap and very safe vaccine. The subject of making good informed decisions has been studied for years and is known as **decision theory**.

# Summary

In this chapter, we learned about probabilistic programming and how inference engines leverage the power of Bayesian modeling. We discussed the main conceptual ideas behind MCMC methods and its central role in modern Bayesian data analysis. We encountered, for the first time, the powerful and easy-to-use PyMC3 library. We revisited the coin-flipping problem from the previous chapter, this time using PyMC3 to define it, solve it, and also perform model checks and diagnoses that are a very important part of the modeling process.

In the next chapter, we will keep building our Bayesian analytics skills by learning how to work with models having more than one parameter and how to make parameters talk to each other.

# Keep reading

- The PyMC3 documentation; be sure to check the examples section: `https://pymc-devs.github.io/pymc3/`.
- Probabilistic Programming and Bayesian Methods for Hackers by Cameron Davidson-Pilon and several contributors. This book/notebooks were originally written using PyMC2 and now have been ported to PyMC3: `https://github.com/quantopian/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers`.
- While My MCMC Gently Samples, a blog from Thomas Wiecki, core developer of PyMC3: `http://twiecki.github.io/`.
- *Statistical Rethinking* by *Richard McElreath* is a very nice introductory book about Bayesian Analysis; the problem is that the examples are in R/Stan. Hence I am porting the examples in the book to Python/PyMC3. Check the GitHub repository at `https://github.com/aloctavodia/Statistical-Rethinking-with-Python-and-PyMC3`.
- *Doing Bayesian Data Analysis* by *John K. Kruschke* is another nice introductory book about Bayesian Analysis, with a similar problem. Most of the examples from the first edition of the book are ported to Python/PyMC3 in the following GitHub repository: `https://github.com/aloctavodia/Doing_bayesian_data_analysis`.

# Exercises

We don't know if 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, although you may disagree with this statement given our record as a species on wars, economic-systems that prioritize profit and not people's wellbeing, and other atrocities. Anyway, I strongly recommend you do the following 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, like 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 $\pi$ keep the number *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 error is more or less in the same order. Try changing the number *N* of points and rerun 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 `errorbar()` function from `matplotlib`. Try using a set of *N*s like *100*, *1000*, *10000*; 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 - A Bayesian Inference Primer*. Compare this code with 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 the code in the following link: `http://twiecki.github.io/blog/2015/11/10/mcmc-sampling/` by Thomas Wiecki.
5. Using PyMC3, change the parameters of the prior beta distribution to match those of the previous chapter and 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 $beta(\alpha = 1, \beta = 1)$? 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? And what do you get if you don't use `find_MAP()` (remember to also remove the start argument from the `sample` function, especially if you are working with a Jupyter/IPython notebook!).

6. Modify the amount of burnin. Try it with values such as *0* and *500*. Try it also with and without the `find_MAP()` function. How different are the results? Hint: this is a very simple model to sample from; remember this exercise for future chapters when we work on more complex models.
7. Use your own data, try to recapitulate the chapter but using data that is interesting to you. This is a valid exercise for the rest of the book!
8. 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.

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. Try to change the model; if you think the task is beyond your actual skills, note down the problem and save it for a future time after reading other chapters of the 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 in Stack Overflow with the tag PyMC3.

# 3

# Juggling with Multi-Parametric and Hierarchical Models

In the previous two chapters, we learned the core ideas of the Bayesian approach and how to use PyMC3 to do Bayesian inference. If we want to build models of arbitrary complexity (and we certainly do), we must learn how to build multi-parametric models. Almost all interesting problems out there need to be modeled using more than one parameter. Moreover, in many real-world problems, some parameters depend on the values of other parameters; such relationships can be elegantly modeled using Bayesian hierarchical models. We will learn how to build these models and the advantages of using them. These are such important concepts that we will keep revisiting them over and over again throughout the rest of the book.

In this chapter, we will cover the following topics:

- Nuisance parameters and marginalized distributions
- The Gaussian model
- Robust estimation in the presence of outliers
- Comparing groups and measuring the effect size
- Hierarchical models and shrinkage

# Nuisance parameters and marginalized distributions

While almost any interesting model is multi-parametric, it is also true that not all the parameters we need in order to build a model are of direct interest to us. Sometimes we need to add a parameter just to build the model, even when we do not really care about this parameter. It may happen that we need to estimate the mean value of a Gaussian distribution to answer an important question we have. For such a model, and unless we know the value of the standard deviation, we should also estimate it even if we do not care about it. Parameters necessary to build a model but not interesting by themselves are known as nuisance parameters. Under the Bayesian paradigm, any unknown quantity is treated in the same way, so whether a parameter is or is not a nuisance parameter is more related to our questions than to the parameter itself, the model, or the inference process.

At this point, you may think that having to build a model with parameters that are of no interest is a burden more than an advantage. On the contrary, by including them, we allow the uncertainty we have about those parameters to propagate adequately to our results. In many problems, we have measurements that we need to convert to a quantity of interest; for example, in **magnetic resonance imaging** (**MRI**), we convert the radio frequency absorbed and emitted by certain nuclei (mostly hydrogen) into images of the interior of a person. Those conversions usually require some nuisance parameters and Bayesian statistics allow those values (and the uncertainty) to be estimated instead of fixing them into some pre-calibrated value or, as is often the case in many problems, using some educated guess or rule of thumb that more or less works well in practice.

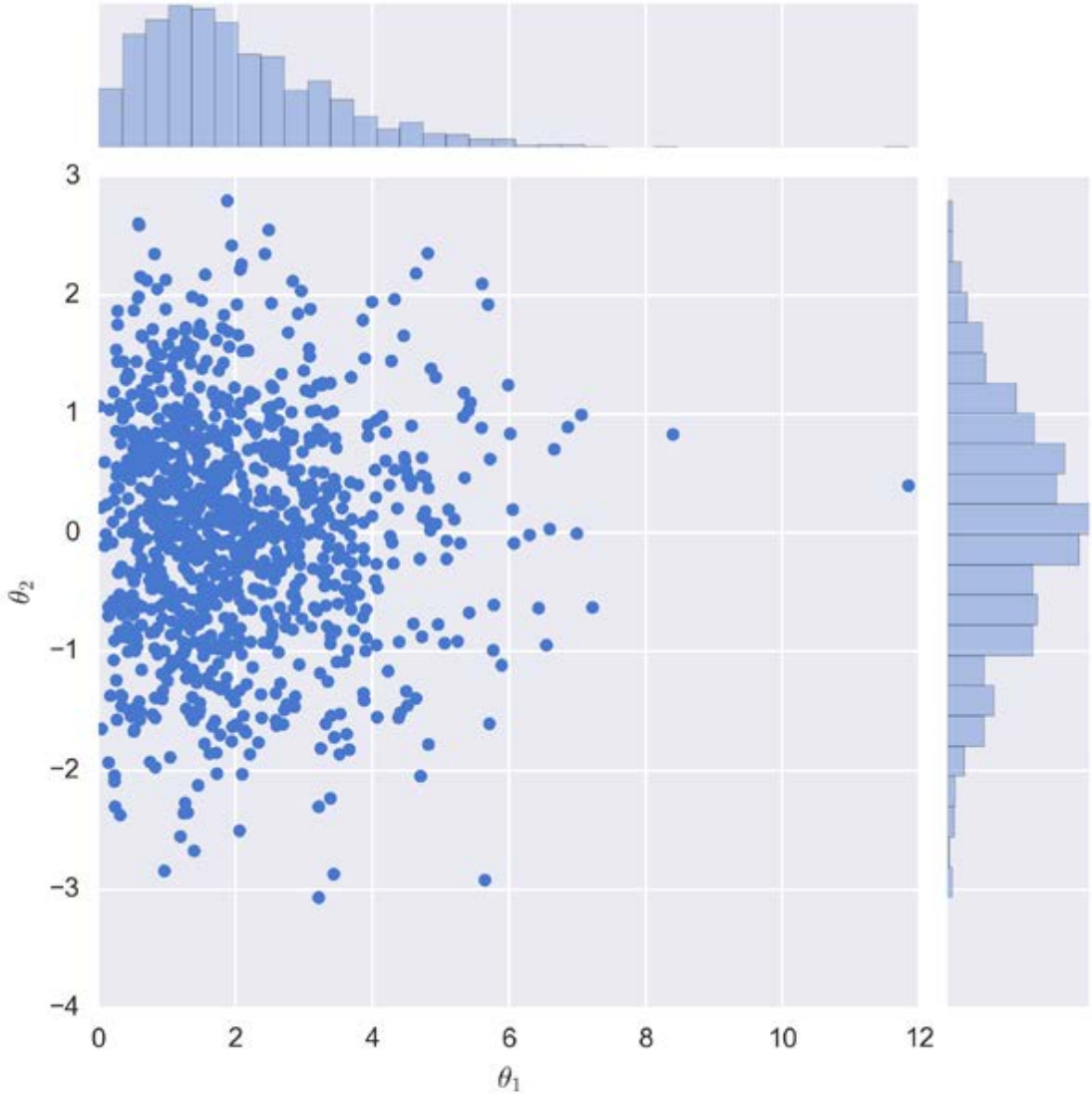
We can write Bayes' theorem for a two-parameter model as follows:

$$p(\theta_1, \theta_2 \mid y) \propto p(y \mid \theta_1, \theta_2)\, p(\theta_1, \theta_2)$$

This can be easily generalized to more than two parameters; just keep adding $\theta s$ . We are assuming the $\theta_1$ and $\theta_2$ are scalars (numbers) and not vectors. The first difference to what we saw in previous chapters is that now we have a bidimensional posterior representing the joint distribution of both $\theta_1$ and $\theta_2$ together. Now let's assume for a moment that $\theta_2$ is a nuisance parameter for our problem. How do we express the posterior in terms of $\theta_1$ only? We have to marginalize the posterior over $\theta_2$ :

$$p(\theta_1 \mid y) = \int p(\theta_1, \theta_2 \mid y)\, d\theta_2$$

That is, we integrate the posterior over all the possible values of $\theta_2$, effectively expressing the posterior in terms of $\theta_1$, but taking into account implicitly the uncertainty of $\theta_2$. For a discrete variable, the integral becomes a summation. The following figure shows the joint distribution of $\theta_1$ and $\theta_2$ in the center and the marginal distribution of $\theta_1$ and $\theta_2$ at the above and right margins, respectively:




Hence, every time we hear about the marginal distribution of parameter *x*, we must think of the average distribution of *x* taken over the entire distribution of the other parameters.

Marginalization is not only useful to get a unidimensional slice of a multidimensional posterior but also to simplify the mathematical and computational analysis. Sometimes it is possible to theoretically marginalize nuisance parameters before computing the posterior. We will see examples of this in *Chapter 7*, *Mixture Models*.

Once nice feature of getting the posterior by simulation, for example, using PyMC3, is that we will get a separate vector for each parameter in the model. That is, the parameters are already marginalized for us.

# Gaussians, Gaussians, Gaussians everywhere

We introduce the Bayesian ideas 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 persons. Well, indeed we get a bimodal one as the result of overlapping the distribution of heights of women and men. 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 like PyMC3.

# Gaussian inferences

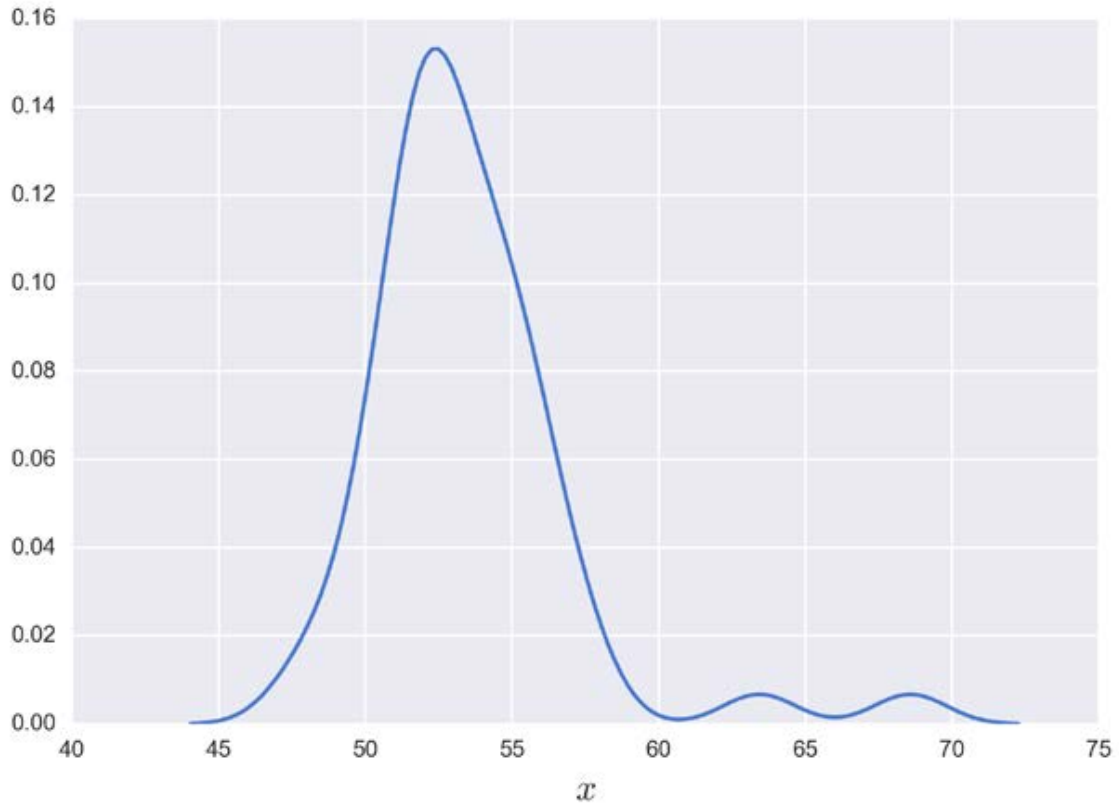
The following example corresponds to experimental measurements in nuclear magnetic resonance, a technique to study molecules and also living things (because, after all, they are just a bunch of molecules). For all we care for this example, we could have been measuring the height of a group of people, the average time to travel back home, the weight of oranges we buy at the supermarket, the number of sexual partners of the Tokay gecko, or in fact any measurement that we can approximate with a Gaussian distribution. In this example, we have *48* measurements:

```
data = np.array([51.06, 55.12, 53.73, 50.24, 52.05, 56.40, 48.45,
52.34, 55.65, 51.49, 51.86, 63.43, 53.00, 56.09, 51.93, 52.31, 52.33,
57.48, 57.44, 55.14, 53.93, 54.62, 56.09, 68.58, 51.36, 55.47, 50.73,
```

```
51.94, 54.95, 50.39, 52.91, 51.5, 52.68, 47.72, 49.73, 51.82, 54.99,
52.84, 53.19, 54.52, 51.46, 53.73, 51.61, 49.81, 52.42, 54.3, 53.84,
53.16])
```

A plot of this dataset shows a Gaussian-like distribution except for two data points far away from the mean:

```
sns.kdeplot(data)
```



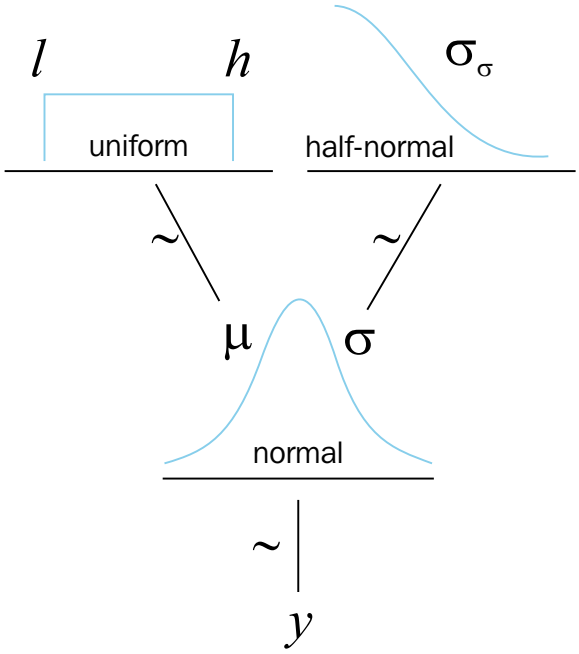

Let's for a moment forget about those two points and assume a Gaussian distribution. Since we do not know the mean or the standard deviation, we must set priors for both of them. Then, a reasonable model could be as follows:

$$\mu \sim Uniform(l, h)$$

$$\sigma \sim HalfNormal(\sigma_h)$$

$$y \sim Normal(\mu, \sigma)$$

Thus, $\mu$ comes from a uniform distribution with boundaries *l* and *h*, lower and upper respectively, and $\sigma$ comes from a half-normal distribution with standard deviation $\sigma_h$. A half-normal distribution is like the regular normal distribution but restricted to positive values; it is like we have folded the normal distribution by its mean. Finally, in our model, the data *y* comes from a normal distribution with the parameters $\mu$ and $\sigma$. Using Kruschke-style diagrams, we have the following:



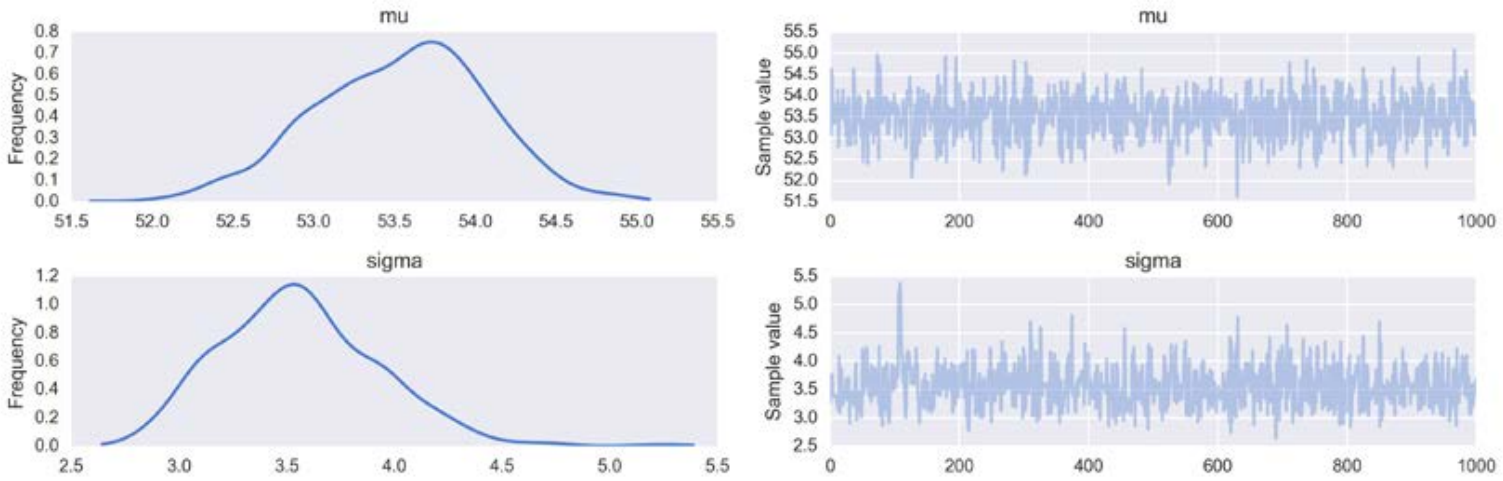

If we do not know the possible values of $\mu$ and $\sigma$, we can set priors reflecting our ignorance. One option is to set the boundaries of the uniform distribution to be *(l=40, h=75)*, which is a range larger than the range of the data. Alternatively, we could have chosen a wider range based, for example, on our previous knowledge. We may know that is not physically possible to have values below *0* or above *100*. So we could set the prior for the mean as a uniform, with parameters *(l=0, h=100)*. For the half normal, we will use a value of $\sigma_\sigma$ is equal to *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', 40, 75)
    sigma = pm.HalfNormal('sigma', sd=10)
    y = pm.Normal('y', mu=mu, sd=sigma, observed=data)

    trace_g = pm.sample(1100)
```

The `traceplot` looks OK and we can continue with the analysis; you could instead be a little more skeptical and run all the diagnostic tests we learned in *Chapter 2, Programming Probabilistically – A PyMC3 Primer*. As you may have noticed, this `traceplot` has two rows: one for each parameter. These are the marginalized distributions; remember that the posterior is really bi-dimensional:

```
chain_g = trace_g[100:]
pm.traceplot(chain_g)
```



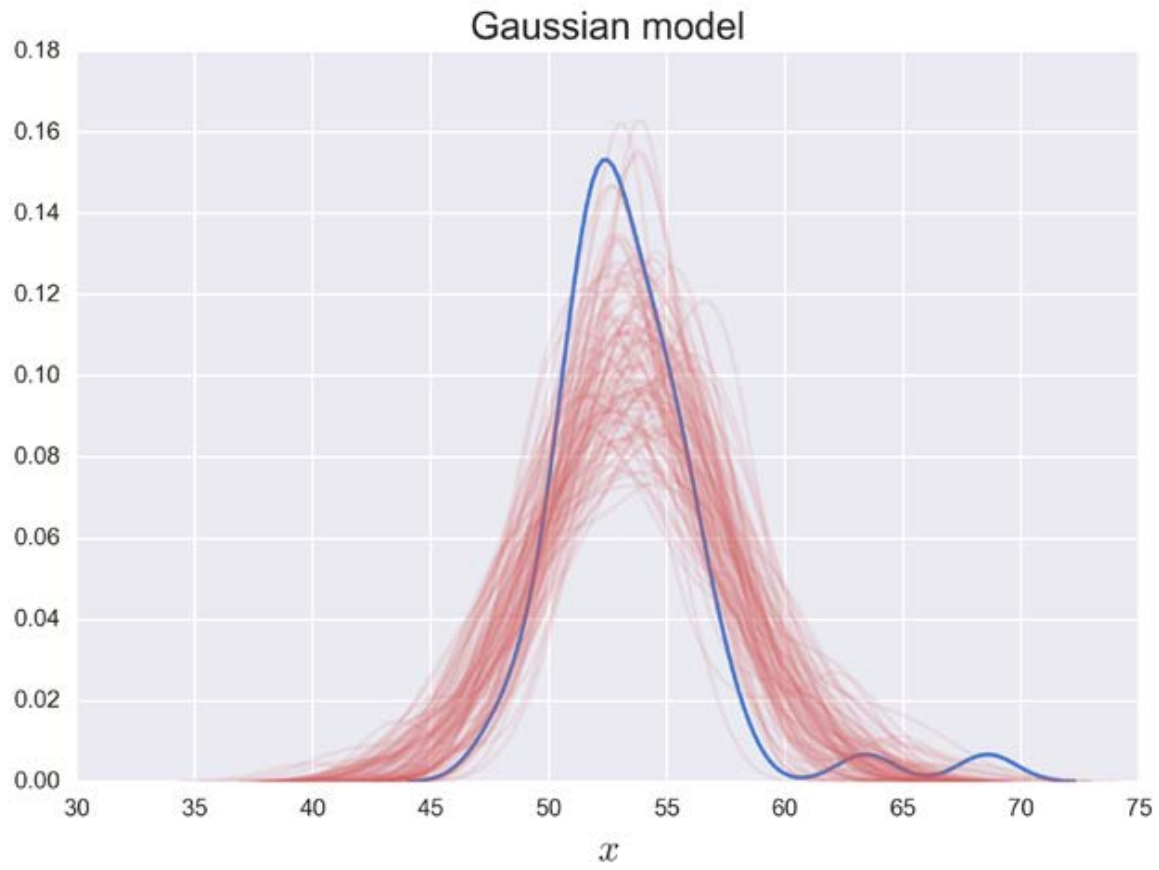

We are going to print the summary of our parameters for later use:

```
pm.df_summary(chain_g)
```

| | mean | sd | mc_error | hpd_2.5 | hpd_97.5 |
|---|---|---|---|---|---|
| mu | 53.51 | 0.53 | 0.02 | 52.56 | 54.54 |
| sigma | 3.55 | 0.38 | 0.01 | 2.86 | 4.32 |

Now that we have computed the posterior, we can use it to simulate data from it. Then we can check how consistent the simulated data is with the observed data. If you remember from *Chapter 1*, *Thinking Probabilistically – A Bayesian Primer*, we generically call these types of comparisons posterior predictive checks, because we are using the posterior to make predictions and those predictions to check the model. Using PyMC3, it is really easy to get posterior predictive samples using the function `sample_ppc()`. In 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_ppc()`, while the other arguments are optional:

```
y_pred = pm.sample_ppc(chain_g, 100, model_g, size=len(data))
sns.kdeplot(data, c='b')
for i in y_pred['y']:
    sns.kdeplot(i, c='r', alpha=0.1)
plt.xlim(35, 75)
plt.title('Gaussian model', fontsize=16)
plt.xlabel('$x$', fontsize=16)
```



In the previous figure, the blue line is a KDE of the data and the semi-transparent red lines are KDE computed from each one of the `100` posterior predictive samples. We can see that the mean of the samples seems to be slightly displaced to the right and that the variance seems to be greater for the samples than for the data. In the following section, we are going to improve the model and get a better match with the data.

# Robust inferences

One objection you may have with the previous model is that we are assuming a Gaussian distribution but we have two data points on the tails of the distribution. So it is not looking really Gaussian. Since the tails of the Gaussian distribution fall very quickly as we move away from the mean, the Gaussian (at least an anthropomorphized one) is *surprised* by seeing those two points and reacts by moving itself toward those points and also by increasing the standard deviation. We may argue that those points have an excessive weight in determining the parameters of the Gaussian. So what to do? One option is to declare those points 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 may even fix those data points, since they are just a result of a coding problem while cleaning the data. On many occasions, we may also want to automate the outlier elimination process by using one of the many *outlier rules*. Two of them are as follows:

- Outliers are all data points below *1.5* times the interquartile range from the lower quartile or *1.5* times the interquartile range above the upper quartile
- All data points below or above two times the standard deviation of our data should be declared outliers and banished from our data

# Student's t-distribution

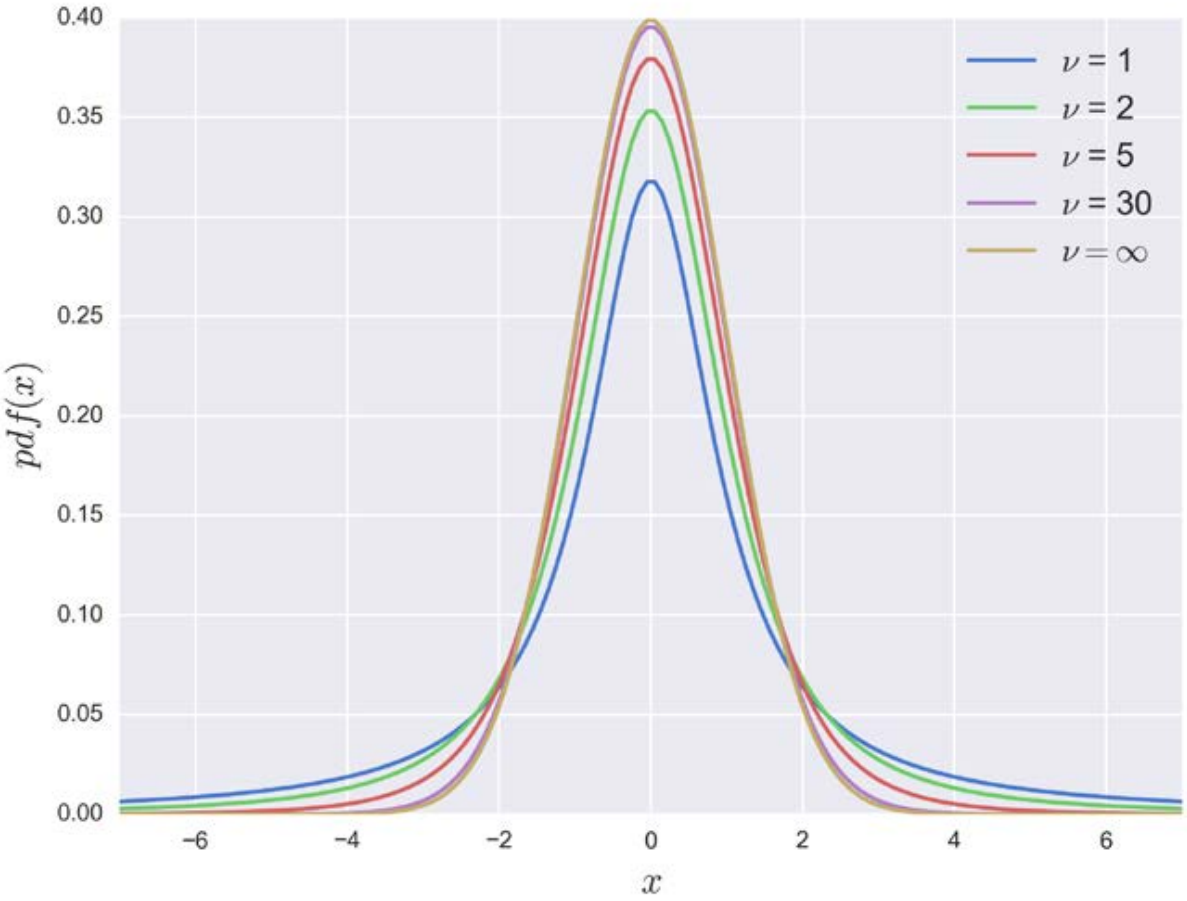
Instead of using one of these rules and changing the data, we can change the model. As a general rule, Bayesians prefer to encode assumptions directly in the model by using different priors or likelihoods than through ad-hoc heuristics like these outlier removal rules.

One very useful option when dealing with outliers is to replace the Gaussian likelihood with the Student's t-distribution. This distribution has three parameters: the mean, the scale (analogous to the standard deviation), and the degrees of freedom usually referred to as $\nu$, which can vary in the interval $[0, \infty]$. Following Kruschke's nomenclature, we are going to call $\nu$ the normality parameter, since it is in charge of controlling how normal-like the distribution is. For a value of $\nu = 1$, we get heavier tails than the Gaussian and the distribution is often called Cauchy or Lorentz, depending on the field. 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, they are not as concentrated around the mean as in a lighter tail distribution like the Gaussian. For example, *95%* of the Cauchy distribution points are found between *-12.7* and *12.7*. Instead, for a Gaussian (with standard deviation one), this occurs between *-1.96* and *1.96*. On the other side, we know that when $\nu$ 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, a sample from a Student's t-distribution is just a bunch of numbers from which it is always possible to compute an empirical mean, but the theoretical distribution itself does not have 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 you 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 $\nu <= 2$ the distribution has no defined variance and hence no defined standard deviation; both the scale and the standard deviation become closer and closer as $\nu$ approaches infinity:

```
x_values = np.linspace(-10, 10, 200)
for df in [1, 2, 5, 30]:
    distri = stats.t(df)
    x_pdf = distri.pdf(x_values)
    plt.plot(x_values, x_pdf, label=r'$\nu$ = {}'.format(df))

x_pdf = stats.norm.pdf(x_values)
plt.plot(x_values, x_pdf, label=r'$\nu = \infty$')
plt.xlabel('x')
plt.ylabel('p(x)', rotation=0)
plt.legend(loc=0, fontsize=14)
plt.xlim(-7, 7)
```




Using the Student's t-distribution, our model can be written as follows:

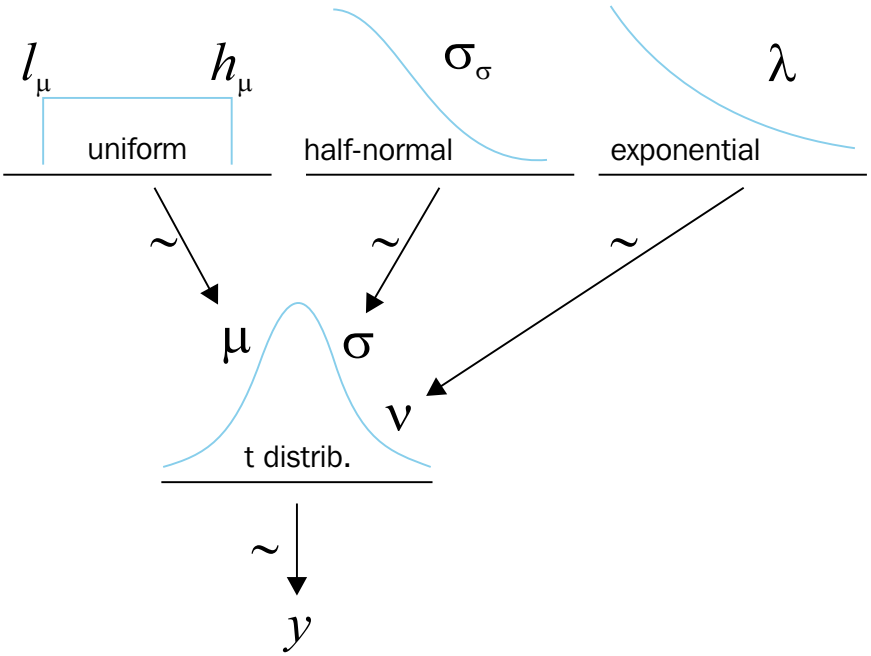
$$\mu \sim Uniform(l, h)$$

$$\sigma \sim HalfNormal(\sigma_h)$$

$$v \sim Exponential(\lambda)$$

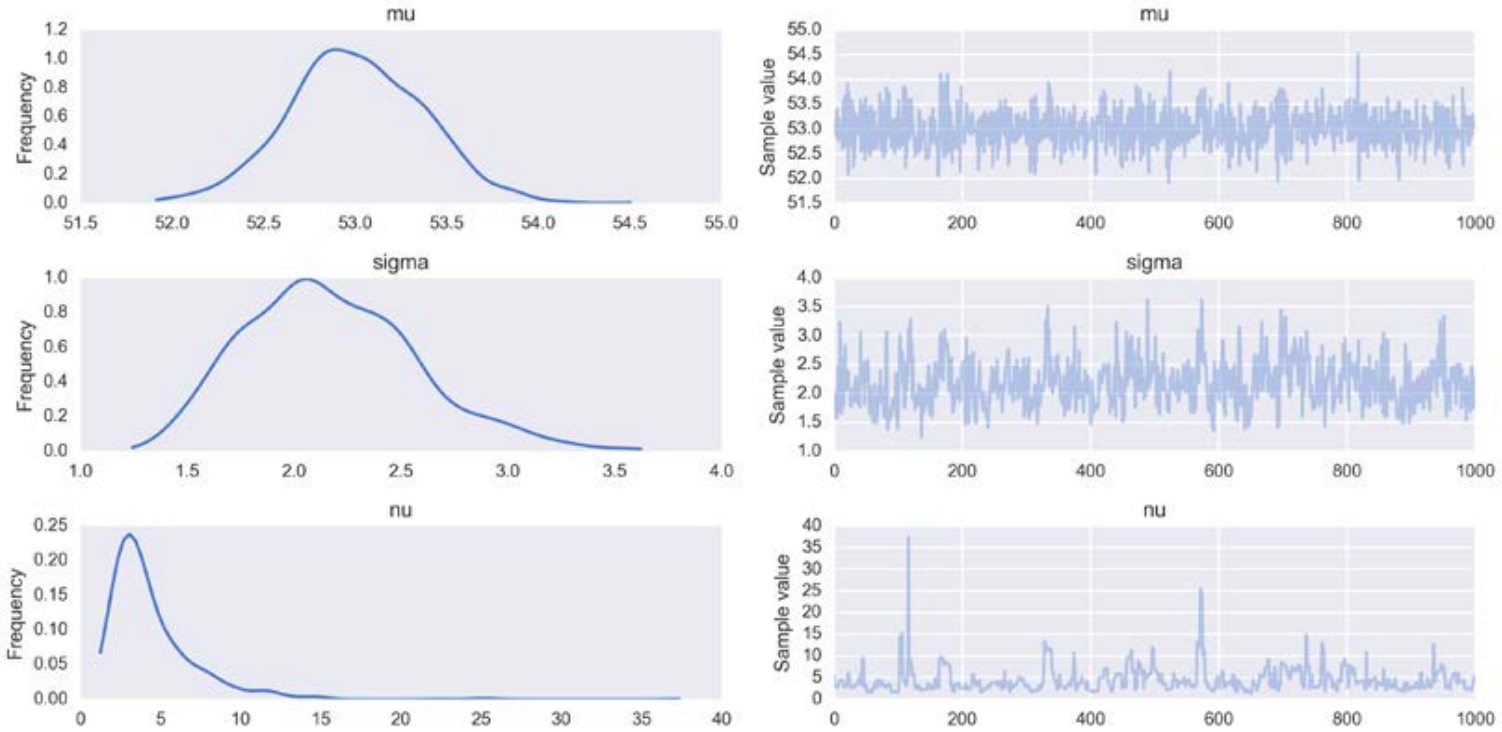
$$y \sim StudentT(\mu, \sigma, v)$$

The main difference with the previous, Gaussian, model is that now the likelihood is a Student's t-distribution and since we have a new parameter, we need one more prior for it. We are going to use an exponential distribution with mean 30. From the preceding figure, we can see that a Student's t -distribution looks pretty similar to a Gaussian (even when it is not). In fact, from the same figure, we can see most of the action happens for relatively small values of $\nu$ . Hence, the exponential prior with mean 30 is a weakly informative prior saying we more or less think $\nu$ should be around 30, but can move to smaller and larger values with ease. Graphically, we have the following:




As usual, PyMC3 allows us to modify our model adding just a few lines. The only cautionary word here is that the exponential in PyMC3 is parameterized with the inverse of the mean of the distribution:

```
with pm.Model() as model_t:
    mu = pm.Uniform('mu', 40, 75)
    sigma = pm.HalfNormal('sigma', sd=10)
    nu = pm.Exponential('nu', 1/30)
    y = pm.StudentT('y', mu=mu, sd=sigma, nu=nu, observed=data)
    trace_t = pm.sample(1100)
chain_t = trace_t[100:]
pm.trace_plot(chain_t)
```




Now print the summary and compare it with our previous results. Before you keep reading, take a moment to spot the difference between both results. Did you notice something interesting?

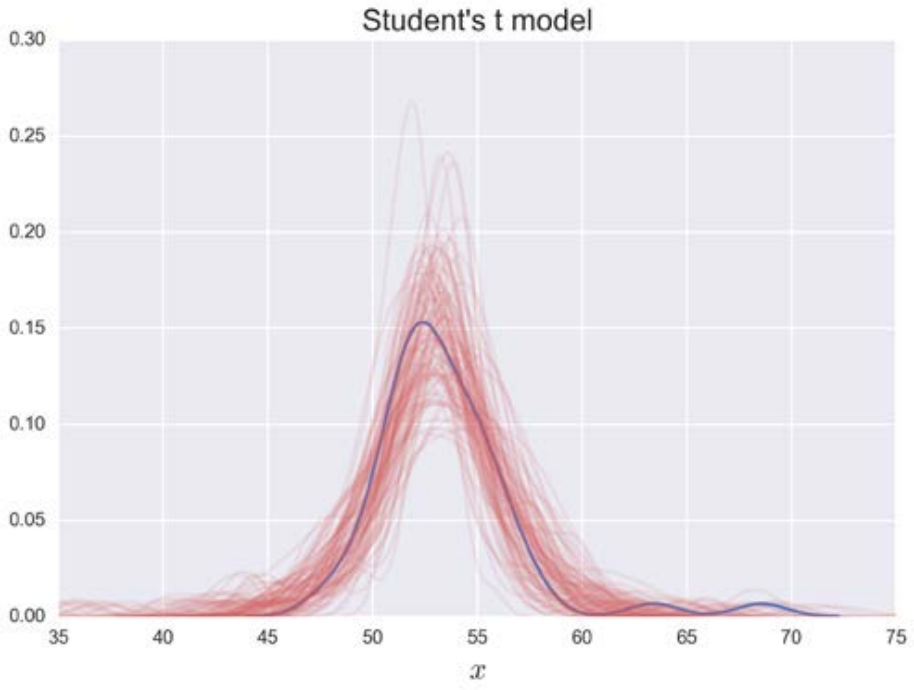
```
pm.df_summary(chain_t)
```

| | **mean** | **sd** | **mc_error** | **hpd_2.5** | **hpd_97.5** |
|---|---|---|---|---|---|
| mu | 52.99 | 0.38 | 0.01 | 52.28 | 53.81 |
| sigma | 2.15 | 0.39 | 0.02 | 1.42 | 2.97 |
| nu | 4.13 | 2.78 | 0.19 | 1.19 | 8.54 |

Well, we can see that the estimation of $\mu$ between both models is similar, with a difference of ~*0.5*. The estimation of $\sigma$ 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 the value of μ is $\nu$ ~*4*, that is, we do not have a very Gaussian-like distribution, but one with heavier tails.

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

```
y_pred = pm.sample_ppc(chain_t, 100, model_t, size=len(data))
sns.kdeplot(data, c='b')
for i in y_pred['y']:
    sns.kdeplot(i, c='r', alpha=0.1)
plt.xlim(35, 75)
plt.title("Student's t model", fontsize=16)
plt.xlabel('$x$', fontsize=16)
```




As we can now see, 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 also how the samples extend far away from the bulk of the data. This is a direct consequence of the Student's t-distribution expecting to see data points far away from the bulk of the data in both directions. The Student's t-distribution in our model allows us to have a more **robust estimation** because the outliers, instead of pulling the mean toward them and increasing the standard deviation, have the effect of decreasing $\nu$ so the mean and the scale are estimated with more weight from the bulk of the data points. 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 $\nu$ above ~2, the value of the scale tends to be pretty close (at least for all practical purposes) to the values estimated after removing the outliers. So, as a rule of thumb, for values of $\nu$ not that small, and taking into account that it is not really theoretically correct, we may consider the scale of a Student's t-distribution as a reasonable practical proxy for the standard deviation of the data after removing outliers.

# Comparing groups

One common task in statistical analysis is to compare groups; for example, we may be interested in how well a patient responds to some drug, the reduction of car accidents by the introduction of a new traffic regulation, or students' test responses 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 one hand, statistical significance is not necessarily practical significance; on the other, a really small effect can be declared significant just by collecting enough data. Also, the idea of statistical significance is connected to computing p-values. 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 for scientists who use statistics on a daily basis. Under the Bayesian framework, we do not need to compute p-values, so we are going to leave them on the side. Instead we are going to focus on estimating how different two groups are. After all, in practice, what we most often really want to know is the **effect size**, that is, a quantitative estimation of the strength of the phenomenon under study.

Sometimes when comparing groups, people talk about a control group and a treatment group (or maybe more than one control and treatment group). 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 our drug works to cure a disease compared to doing nothing (or, as is generally done, against the placebo effect). Another interesting question will be how good our drug is compared with the (already approved) most-used 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 good way to lie, using statistics. For example, imagine you work for an evil dairy-product company that wants to sell overly sugared yogurts to kids by telling their parents that your yogurt boosts the immune system. One way to back up your claim with *research* is by using milk or water as a control group, instead of another cheaper, less sugary, less marketed yogurt. It may sound silly put this way, but when someone says something is harder, better, faster, stronger, remember to ask what baseline is used for comparison.

# The tips dataset

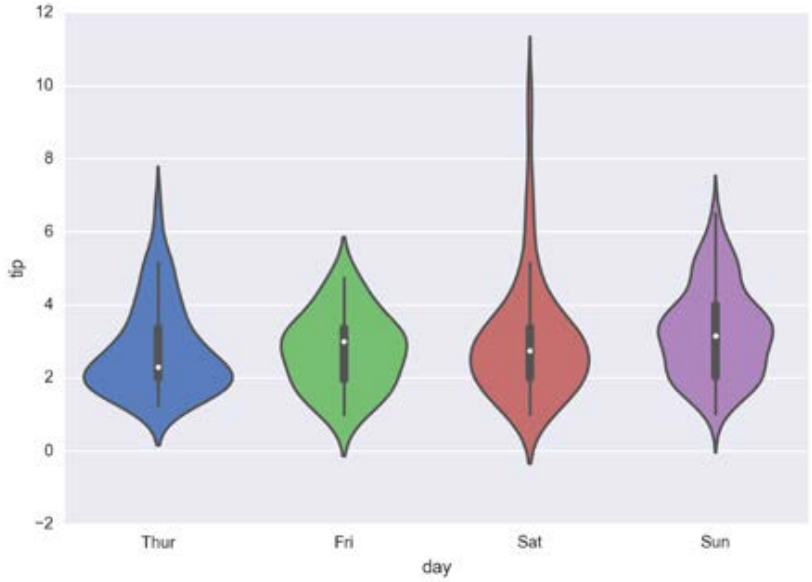
To explore the ideas in this section, we are going to use the `tips` dataset that comes with `seaborn`. We want to study the effect of the day of the week on the amount of tips at a restaurant. For this example, notice there is not really a control group and a treatment group. This is just an observational study, not an experiment like in the drug example. If we wish, we can arbitrarily establish one day, for example, Thursday, as the reference, or *control*, even when we are not controlling a thing. One of the caveats of observational studies is that it is not possible to establish a causal relationship, just correlations. In fact, the study of how to get causal relationships from data is a very active research area. We will come back to this in *Chapter 4*, *Understanding and Predicting Data with Linear Regression Models*. For now, let's start the analysis by loading the dataset as a Pandas data frame using just one line of code. If you are not familiar with Pandas, the `tail` command is just showing the last portion of the data frame (you can also try using `head`):

```
tips = sns.load_dataset('tips')
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 |

From this data frame, 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)
```

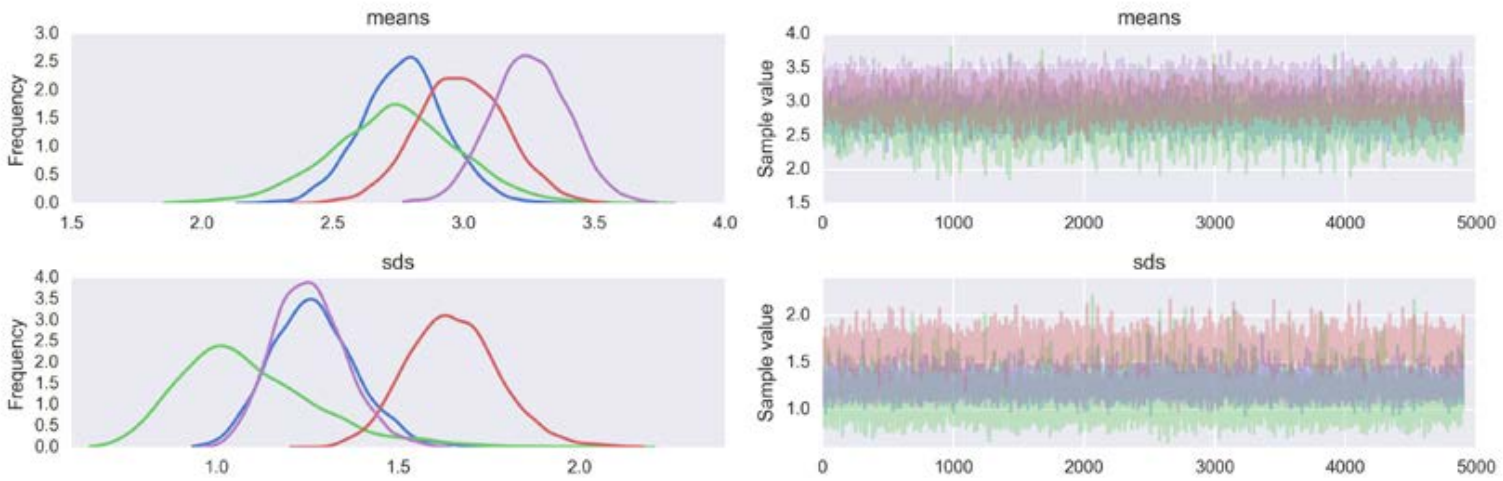


Just to simplify things, we are going to create two variables: the `y` variable with the amount of the `tips` and `idx` with a categorical dummy variable or index; that is, instead of having Thursday, Friday, Saturday, and Sunday, we are going to have the values *0*, *1*, *2*, and *3*:

```
y = tips['tip'].values
idx = pd.Categorical(tips['day']).codes
```

The model for this problem is almost the same as before; the only difference is that now, $\mu$ and $\sigma$ are going to be vectors instead of scalar random variables. In other words, every time we sample from our priors, we are going to get four values for $\mu$ and four values for $\sigma$, instead of one, as in our previous model. PyMC3 syntax is extremely helpful for this situation: instead of writing `for` loops, we can write our model in a vectorized way. The changes from the previous, normal, model are minimal. For the priors, we need to pass a shape argument and for the likelihood, we need to properly index $\mu$ and $\sigma$, and that's why we create the `idx` variable:

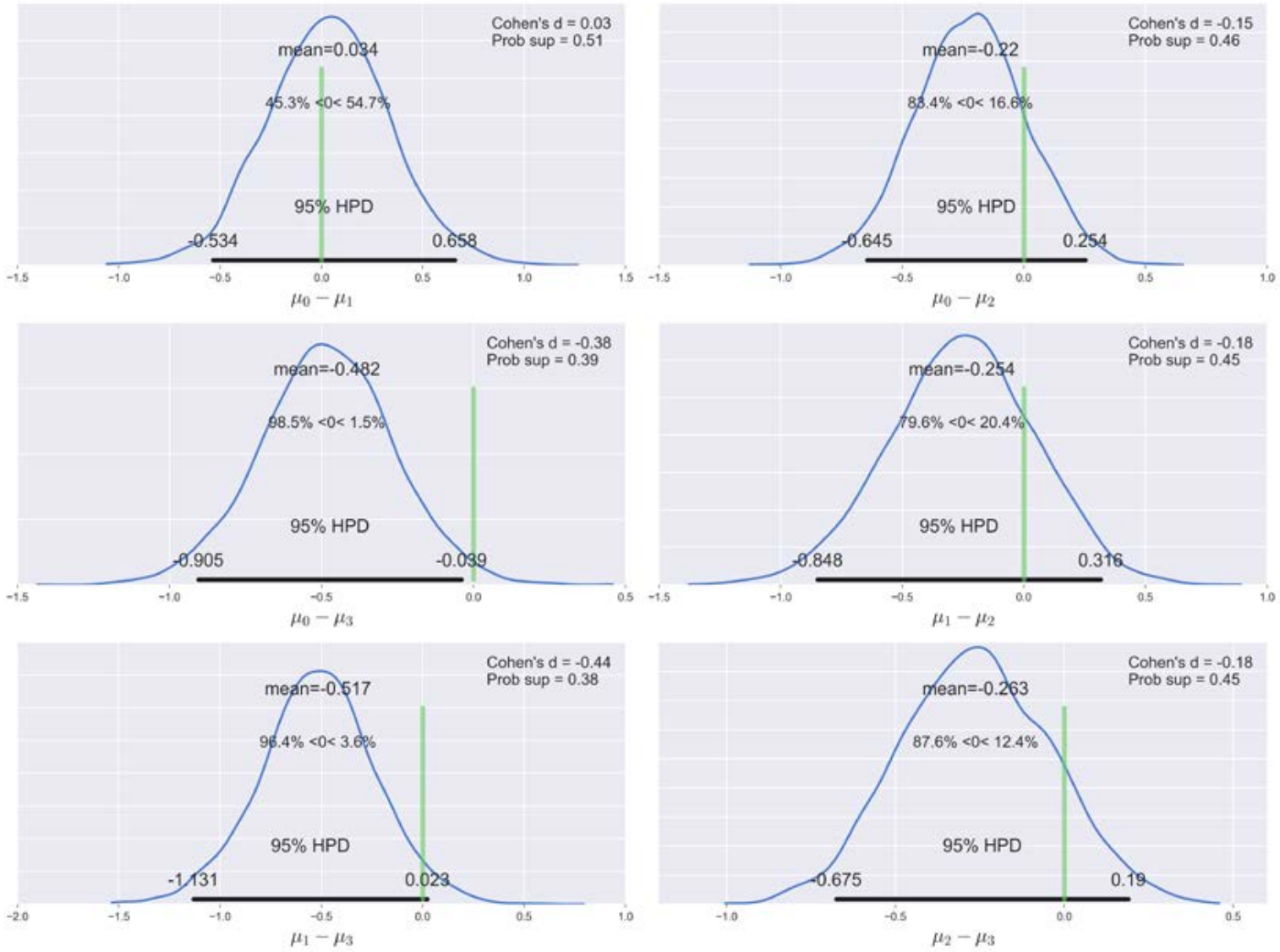
```
with pm.Model() as comparing_groups:
    means = pm.Normal('means', mu=0, sd=10, shape=len(set(x)))
    sds = pm.HalfNormal('sds', sd=10, shape=len(set(x)))

    y = pm.Normal('y', mu=means[idx], sd=sds[idx], observed=y)

    trace_cg = pm.sample(5000)
chain_cg = trace_cg[100::]
pm.traceplot(chain_cg)
```



As always, one option is to report the estimated values as summarized by the `df_summary` function; you can do that (and also the diagnostic tests!). It's important to remember that a Bayesian analysis returns a complete distribution of credible values (given the data and model). And, hence, we can operate with a posterior and ask the posterior all the questions we may think reasonable. For example, we may ask for the distribution of the difference of the means between groups, and that is what we are going to do next.

We are going to use the PyMC3 function `plot_posterior` with a reference value (`ref_val`) of *0* because we want to compare the posterior against this value (no difference). The following code is just a way to plot the difference without repeating the comparison. Instead of getting the *all-against-all* matrix, we are just plotting the upper triangular portion. The only weird part in the code and the plot is something called the **Cohen's d** and the **probability of superiority**, which we will explain in the coming paragraphs. But they are nothing more than ways to express the effect size:

```
dist = dist = stats.norm()
_, ax = plt.subplots(3, 2, figsize=(16, 12))

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 = chain_cg['means'][:,i]-chain_cg['means'][:,j]
    d_cohen = (means_diff / np.sqrt((chain_cg['sds'][:,i]**2 +
      chain_cg['sds'][:,j]**2) / 2)).mean()
    ps = dist.cdf(d_cohen/(2**0.5))

    pm.plot_posterior(means_diff, ref_val=0, ax=ax[k, l],
      color='skyblue')
    ax[k, l].plot(0, label="Cohen's d = {:.2f}\nProb sup =
      {:.2f}".format(d_cohen, ps) ,alpha=0)
    ax[k, l].set_xlabel('$\mu_{}-\mu_{}$'.format(i, j),
      fontsize=18)
    ax[k,l ].legend(loc=0, fontsize=14)
```




In the preceding example, one way to interpret the results is by comparing the reference value with the HPD interval. According to the previous figure, we have only one case when the 95% HPD excludes *0* (our reference value), the difference in tips between Thursday and Sunday. For all the other examples, we cannot rule out a difference of *0* (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 waiter and waitress the same amount of tip money? Those kinds of questions cannot be answered by statistics; they can only be informed by statistics.

There are several ways to try to measure the effect size; we are going to see two of them: the Cohen's d and the probability of superiority.

# Cohen's d

A common way to measure the effect size (at least in some fields, such as psychology) is the **Cohen's d**:

$$\frac{\mu_1 - \mu_2}{\sqrt{\frac{\sigma_1^2 + \sigma_2^2}{2}}}$$

So the effect size is the difference of the means with respect to the average standard deviation of both groups. In the preceding code, we compute the Cohen's d from the estimated means and standard deviation, so we could also report a distribution of Cohen's d and not just the mean value.

When comparing groups is important to include the variability of each group, for example using the standard deviation. A change of about *x* units from one group to another could be explained by every individual data point changing exactly *x* units, or by half of them changing *0* and the other half changing *2x* units, and of course by many other combinations. The effect size computed in terms of the Cohen's d, can be interpreted as a *Z*-score, so 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. The problem of using Cohen's d is that it does not sound like a very interpretable quantity and we need to calibrate ourselves to say if a given value is big, small, medium, and so on. Of course, this calibration can be acquired with practice, but once again, what is a big effect is context-dependent; if we perform several analyses for more or less the same type of problem, 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!). You'll find a very nice web page to explore what different values of Cohen's d look like at `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.

# Probability of superiority

This is another way to report the effect size and is defined as the probability that a data point from one group has a larger value than one taken from the other group. If we assume that the distribution for both groups is normal, we can compute the probability of superiority from the Cohen's d (as we did in the preceding code) using the following expression:

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

Here, $\Phi$ is the cumulative normal distribution and $\delta$ is the Cohen's d. We can compute the point estimation of the probability of superiority (what is usually reported) or we can compute a whole distribution of values. Also note that we can use this formula, which assumes normality, for computing the probability of superiority from the Cohen's d, or we can just compute it from the posterior (see the exercise section). This is a very nice advantage of using MCMC methods. Once we get samples from the posterior, we can compute quantities from it, such as the probability of superiority (or many others), without relying on distributional assumptions.

# Hierarchical models

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

- Estimate variables for each neighborhood/zone as separate entities
- Pool 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, we put shared priors over our priors. Instead of fixing the parameters of our priors to some constant numbers, we estimate them directly from the data by placing priors over them. These higher-level priors are often called **hyper-priors**, and their parameters **hyper-parameters**; hyper means *over* in Greek. Of course, it is 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 does not help to make better inferences. On the contrary, we end up entangled in a web of hyper-priors and hyper-parameters 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.

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 the following values 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 generate the synthetic data with the following code:

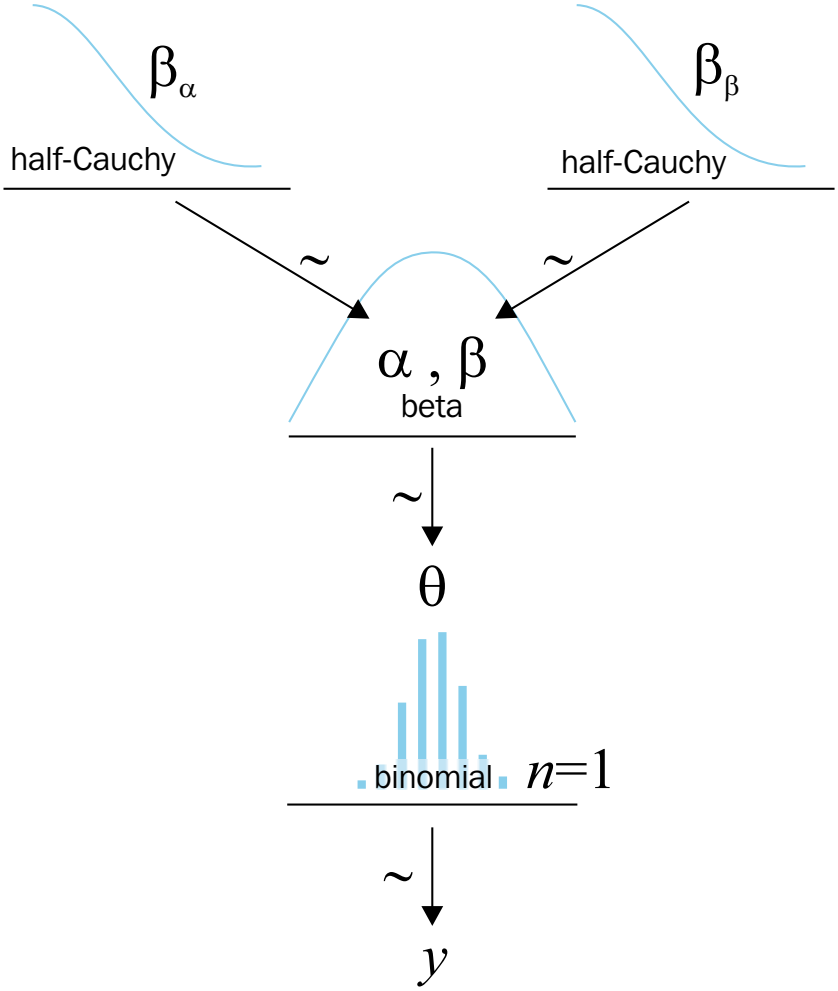
```
N_samples =  [30, 30, 30]
G_samples =  [18, 18, 18]

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 list `G_samples`, we keep a record of the number of good-quality samples per group. The rest of the code is there just to generate the list `data`, filled with *0*s and *1*s.

The model is essentially the same one we use for the coin problem, except that now we have to specify the hyper-priors that will influence the beta-prior:

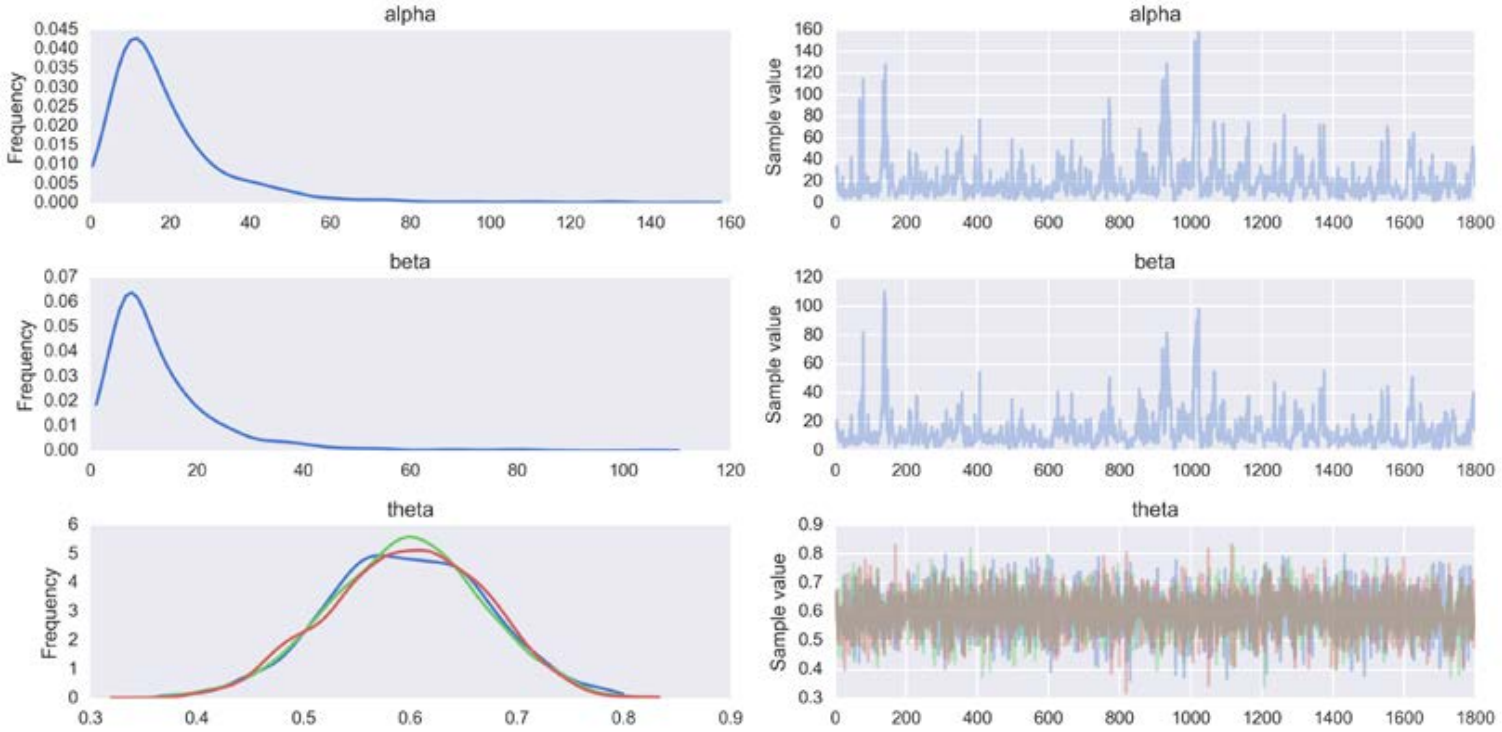
$$\alpha \sim HalfCauchy(\beta_\alpha)$$
$$\beta \sim HalfCauchy(\beta_\beta)$$
$$\theta \sim Beta(\alpha, \beta)$$
$$y \sim Bern(\theta)$$

Using Kruschke diagrams, it is evident that this new model has one additional level compared to all previous models:




```
with pm.Model() as model_h:
    alpha = pm.HalfCauchy('alpha', beta=10)
    beta = pm.HalfCauchy('beta', beta=10)

    theta = pm.Beta('theta', alpha, beta, shape=len(N_samples))

    y = pm.Bernoulli('y', p=theta[group_idx], observed=data)
```

```
    trace_j = pm.sample(2000)
chain_h = trace_h[200:]
pm.traceplot(chain_h)
```




# Shrinkage

Please now join me in a brief experiment. I will need you to print the summary and save the result for later use. Then I want you to re-run the model two more times, always keeping a record of the summary, one time setting all the elements of `G_samples` to three, and one last time 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 theta in each experiment. Based on the first two runs of the model, could you predict the outcome of the third case?

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

| G_samples | Theta (mean) |
|---|---|
| *18, 18, 18* | *0.6, 0.6, 0.6* |
| *3, 3, 3* | *0.1, 0.1, 0.1* |
| *18, 3, 3* | *0.53, 0.14, 0.14* |

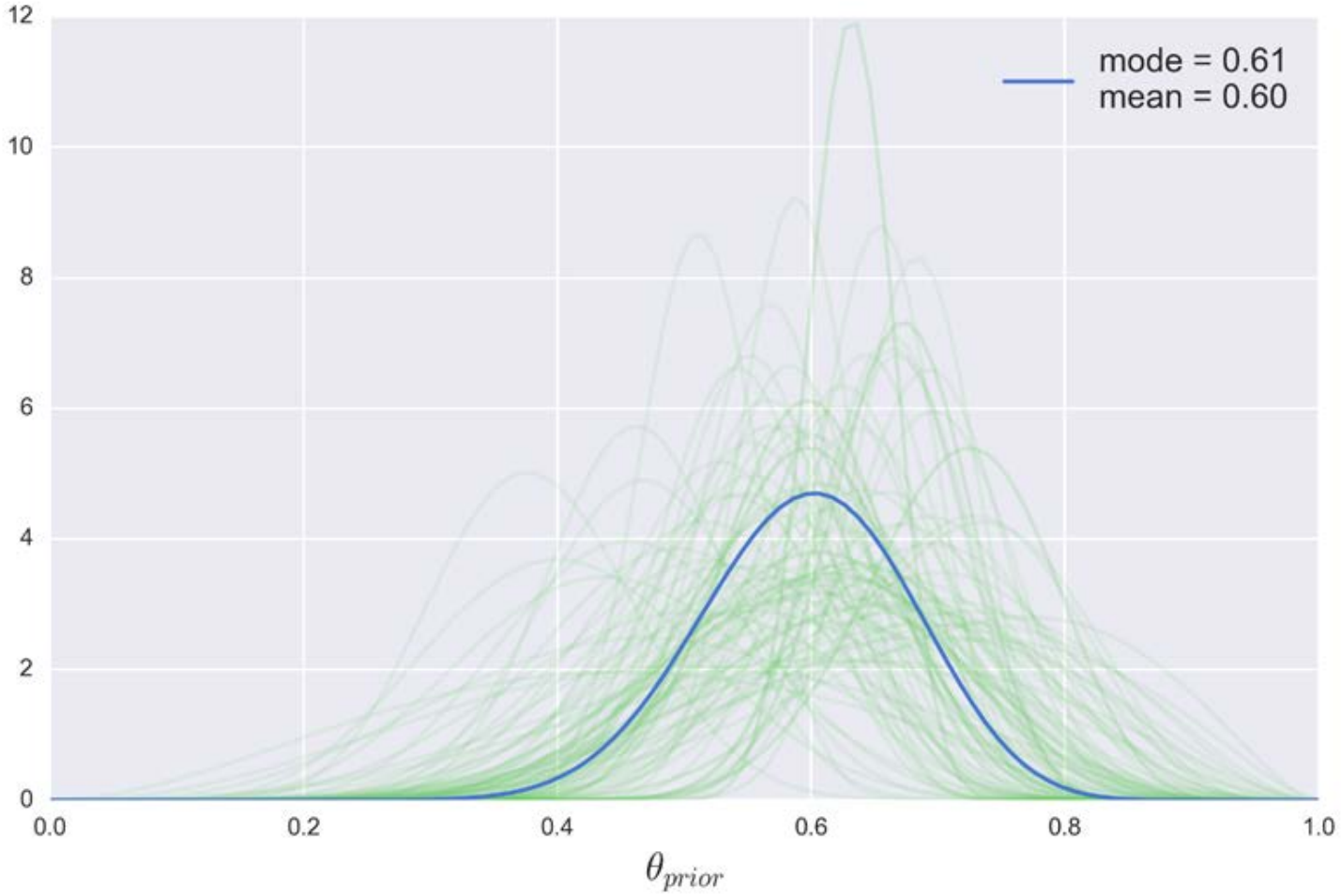
In the first row, we see that for *18* good samples out of *30* we get that the mean estimation for $\theta$ is *0.6*; remember that now, theta is a vector, since we have three groups, and thus we have a mean value for each group. Then on the second row, we have only *3* good samples out of *30* and the mean of $\theta$ is *0.1*. At the end, on the last row, we get something interesting and probably unexpected. Instead of getting a mix of the mean estimates of $\theta$ from the other rows, such as *0.6*, *0.1*, *0.1*, we get different values, namely *0.53*, *0.14*, *0.14*. What on Earth happened? Maybe a convergence problem or some error with the model specification? Nothing of that, we get our estimate shrunk toward the common mean. And this is totally OK; indeed, this is just a consequence of our model; by putting hyper-priors, we are estimating the values of our (beta) prior from the data itself and each group is informing the rest, and in turn is informed by the estimation of the others. In other words, the groups are effectively sharing information through the hyper-prior and we are observing what is known as **shrinkage**. This effect is the consequence of partially pooling the data; we are modeling the groups neither as independent from each other nor as a single big group. We have something in the middle, and one of the consequences is the shrinkage effect.

Why is this useful? Because having shrinkage contributes to more stable inference. This is in many ways similar to what we saw with the Student t -distribution and the outliers. Using 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 use "*conservative*" in a flattering way), one that is less responsive to extreme values in individual groups. To illustrate this, imagine we have a neighborhood with a different number of samples; 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 other hand, 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 mis-estimation 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 of 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 thinking that each group is a data point and we are estimating the standard deviation at the group-level. We generally do not trust an estimation with a 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(chain_h), size=100):
    pdf = stats.beta(chain_h['alpha'][i], chain_h['beta'][i]).pdf(x)
    plt.plot(x, pdf,  'g', alpha=0.05)

dist = stats.beta(chain_h['alpha'].mean(), chain_h['beta'].mean())
pdf = dist.pdf(x)
mode = x[np.argmax(pdf)]
mean = dist.moment(1)
plt.plot(x, pdf, label='mode = {:.2f}\nmean = {:.2f}'.format(mode,
  mean))

plt.legend(fontsize=14)
plt.xlabel(r'$\theta_{prior}$', fontsize=16)
```




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 learning how to use them to build better models, including a more detailed discussion of the overfitting and underfitting problem when building models, in *Chapter 6*, *Model Comparison*.

# Summary

In this chapter, we built from the last two by extending our abilities to manage models with more than one parameter. This turned out to be something very simple to do with the help of PyMC3. For example, obtaining the marginal distribution from the posterior is just a matter of properly indexing the trace. We also explored a few examples of using the posterior to derive quantities of interest from it, such as synthetic data or measures to better explain the data. We found the Gaussian model for the first, but certainly not the last, time, since it is one of the pillars of data analysis. Before we had any time to glorify the Gaussian model, we pushed it to its limits with the help of potential outliers in the data. Therefore, we learned to relax the normality assumption by using the Student's t-distribution, which led us to the concept of robust models and how we can change a model to better fit our problem. We used the Gaussian model in the context of comparing groups, a common data analysis task in many circumstances, and we discussed some measures of *effect size* to help us quantify how different groups are. We save for the last, as we usually do with a fine dessert, one of the most important concepts to learn: hierarchical models or how to structure a problem to get better inferences and shrunken estimates by partially pooling information from groups.

At the turn of this page, we will learn about linear models and how to use them to make sense of data.

# Keep reading

- *Doing Bayesian Data Analysis, Second Edition* by *John Kruschke*. Chapter 9.
- *Statistical Rethinking* by *Richard McElreath*. Chapter 12.
- *Bayesian Data Analysis, Third Edition* by *Andrew Gelman* and others. Chapter 5.
- After reading *Chapter 4*, *Understanding and Predicting Data with Linear Regression Models*, be sure to check the GLM-hierarchial and Rugby examples from PyMC3's documentation.

# Exercises

1. For the first model, 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, to model bounded data like this? Remember we said is not possible to get values below *0* or above *100*.
2. Using the data from the first example, 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 adding more outliers.
3. Modify the *tips example* to make it robust to outliers. Try with one shared $\nu$ for all groups and also with one $\nu$ per group. Run posterior predictive checks to assess these three models.
4. Compute the probability of superiority directly from the posterior (without computing Cohen's d first). You can use the function `sample_ppc()` to take a sample from each group. Is it really different from the calculation assuming normality? Can you explain the result?
5. Repeat the exercise we did on the main text with the water quality example, but this time without hierarchical structure. Just use a flat prior such as Beta (*alpha=1, beta=1*). Compare the results of both models.
6. Create a hierarchical version of the *tips example* by partially pooling across the days of week. Compare the results to those obtained without the hierarchical structure.
7. Repeat the examples in this chapter, but initialize the sampling with the `find_MAP()` function . Did you get the same inferences? How the use of `find_MAP()` changes the amounts of burn-in? Are the inferences faster, slower, or the same?
8. Run diagnostic tests for all the models and take action, such as running more steps if necessary.
9. Use your own data with at least one of the models in this chapter. Remember the three steps of model building from *Chapter 1*, *Thinking Probabilistic – A Bayesian Inference Primer*.

# 4

# Understanding and Predicting Data with Linear Regression Models

In this chapter, we are going to see one of the most widely used models in statistics and machine learning: the linear regression model. This model is very useful on its own and also can be considered as a building block of several other methods. If you 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, and this is the main topic of this chapter.

In this chapter, we will cover the following topics:

- Linear regression models
- Simple linear regression
- Robust linear regression
- Hierarchical linear regression
- Polynomial regression
- Multiple linear regression
- Interactions

# Simple linear regression

Many problems we find in science, engineering, and business are of the following form. We have a continuous variable, and by continuous we mean a variable represented using real numbers (or floats if you wish). We call this variable the **dependent**, **predicted**, or **outcome** variable. And we want to model how this dependent variable *depends* on one or more variables, which we call **independent**, **predictor**, or **input** variables. The independent variable can be continuous or it can be categorical. These type of problems can be modeled using linear regression. If we have only one independent variable we may use a simple linear regression model problem; if we have more than one independent variable then we may apply a multiple linear regression model. Some typical situations that linear regression models can be used in are as follows:

- 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** (**ML**) 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 in a state of uncertainty. ML and statistics are really intertwined subjects and the connections are made clearer if you take a probabilistic perspective, as Kevin Murphy does in his great book. 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 ML terminology we say a regression problem is an example of supervised learning. Under the ML framework, we have a regression problem when we want to learn a mapping from $\boldsymbol{x}$ to $\boldsymbol{y}$, with $\boldsymbol{y}$ being a continuous variable. We say the learning process is supervised because we know the values of $\boldsymbol{x}$-$\boldsymbol{y}$ 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 $\boldsymbol{x}$ but not $\boldsymbol{y}$.

# The core of 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 to build linear models.

Chances are high that you are already familiar with the following equation:

$$y_i = \alpha + \beta x_i$$

This equation says there is a linear relation between the variable $\boldsymbol{x}$ and the variable $\boldsymbol{y}$. The exact form is given by the parameter $\beta$ which controls the slope on the line. The slope can be interpreted as the change in the variable $\boldsymbol{y}$ per unit change in the variable $\boldsymbol{x}$. The other parameter, $\alpha$, is known as the intercept and tells us the value of $y_i$ when $x_i = 0$. Graphically, the intercept is the point where the line intercepts the $y$ axis.

There are several ways to find the parameters for a linear model; one of these methods is known as least squares fitting. Every time you fitted a line using some software you were most likely using this method under the hood. The method returns the values of $\alpha$ and $\beta$ that produce the lowest average quadratic error between the observed $\boldsymbol{y}$ and the predicted $\boldsymbol{y}$ (for each value of $\boldsymbol{x}$). Expressed in that way the problem of estimating $\alpha$ and $\beta$ is an optimization problem, that is, a problem where we try to minimize (or maximize) some function. Optimization is not the only way to find the solution to a regression model. The same problem can be stated under a probabilistic (Bayesian) framework. Thinking probabilistically gave us several advantages; we can obtain the best values of $\alpha$ and $\beta$ (the same as with optimization methods) together with the uncertainty estimation of those parameters. Optimization methods require extra work to provide this information. Additionally, we get the flexibility of Bayesian methods, meaning we will be able to adapt our models to our particular problems; for example, moving away from normality assumptions, or building hierarchical linear models as we will see next in this chapter.

Probabilistically, a linear regression model can be expressed as:

$$\boldsymbol{y} \sim N\left(\mu = \alpha + \beta \boldsymbol{x}, \sigma = \varepsilon\right)$$

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

Since we do not know the values of $\alpha$, $\beta$, or $\varepsilon$, we have to set prior distributions for them. A reasonable choice would be:

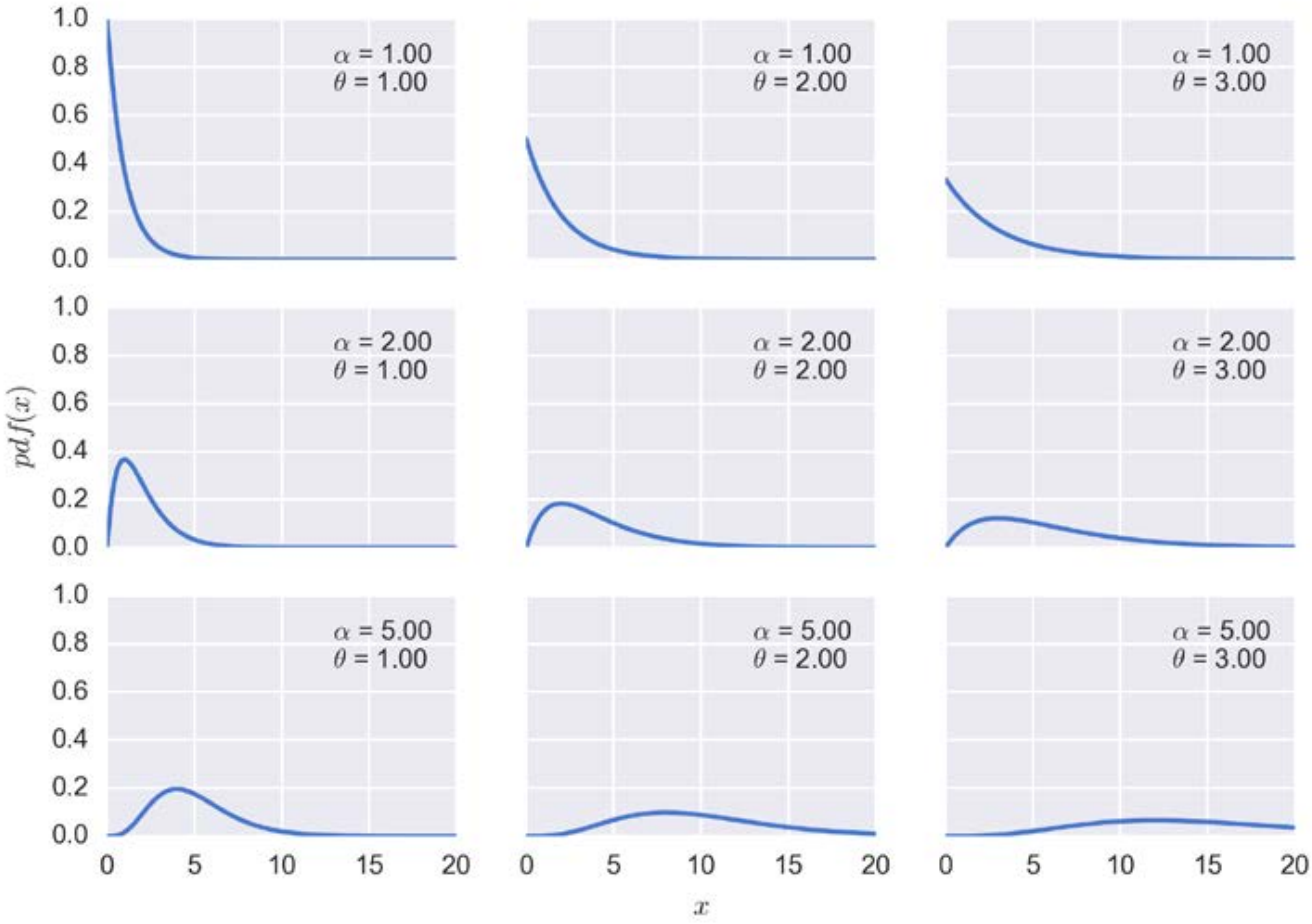
$$\alpha \sim N(\mu_\alpha, \sigma_\alpha)$$
$$\beta \sim N(\mu_\beta, \sigma_\beta)$$
$$\varepsilon \sim U(0, h_s)$$

For a prior over $\alpha$ we can use a very flat Gaussian, that is one with a high value of $\sigma_\alpha$, relative to 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. The same goes for the slope, although for many problems we can at least know the sign of it a priori. For epsilon, we can set $h_s$, 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 an almost null effect of the prior over the data. Under this Bayesian model for simple linear regression with effectively flat priors, we will get essentially the same estimates as using the least squares method. Alternatives to the uniform distribution are the half normal or the half Cauchy distributions. The half Cauchy generally works well as a good regularizing prior (see *Chapter 6, Model Comparison* for details) If we want to use really strong priors around some specific value for the standard deviation we can use the gamma distribution. The default parametrization 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 or the mean and standard deviation.

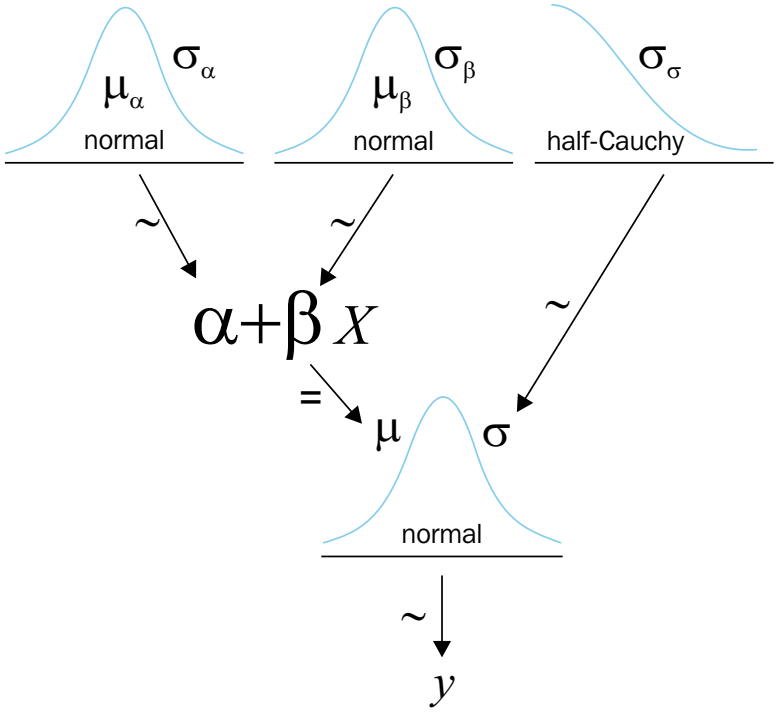
Before continuing with the linear regression let's take a look at the gamma distribution for a set of different parameters:

```
rates = [1, 2, 5]
scales = [1, 2, 3]

x = np.linspace(0, 20, 100)
f, ax = plt.subplots(len(rates), len(scales), sharex=True,
  sharey=True)
for i in range(len(ass)):
    for j in range(len(scales)):
        rate = rates[i]
        scale = scales[j]
        rv = stats.gamma(a=rate, scale=scale)
        ax[i,j].plot(x, rv.pdf(x))
```

```
        ax[i,j].plot(0, 0,
        label="$\\alpha$ = {:3.2f}\n$\\theta$ =
          {:3.2f}".format(rate, scale), alpha=0)
        ax[i,j].legend()
ax[2,1].set_xlabel('$x$')
ax[1,0].set_ylabel('$pdf(x)$')
```

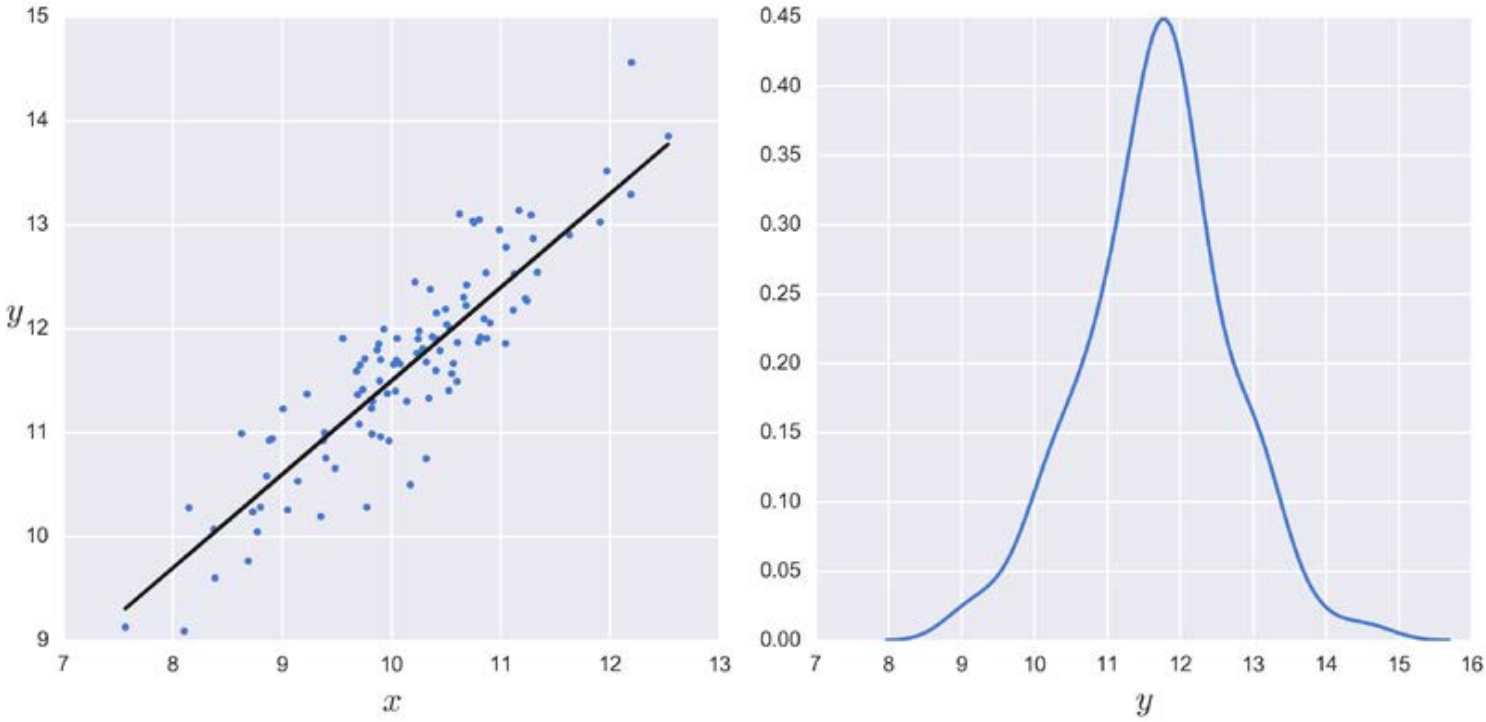


Continuing with the linear regression model, using the nice and easy-to-interpret Kruschke diagrams, we have:




Now of course we need the data to feed the model. Once again we are going to rely on a synthetic data set to build intuition on the model. We will create the datasets in such a way that we know the values of the parameters that later we are going to try to find out.

```
np.random.seed(314)
N = 100
alfa_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 = alfa_real + beta_real * x
y = y_real + eps_real

plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plt.plot(x, y, 'b.')
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$y$', fontsize=16, rotation=0)
plt.plot(x, y_real, 'k')
plt.subplot(1,2,2)
sns.kdeplot(y)
plt.xlabel('$y$', fontsize=16)
```




Now we use PyMC3 to fit the model, nothing we have not seen before. Wait, in fact there is something new! $\mu$ is expressed in the model as a deterministic variable. All the variables we have seen up to now were stochastic, that is we get a different number every time we ask for a value. Instead, a deterministic variable is fully determined by its arguments, even if the arguments are stochastic, as in the following code. If we specify a deterministic variable, PyMC3 will save the variable in the trace for us:

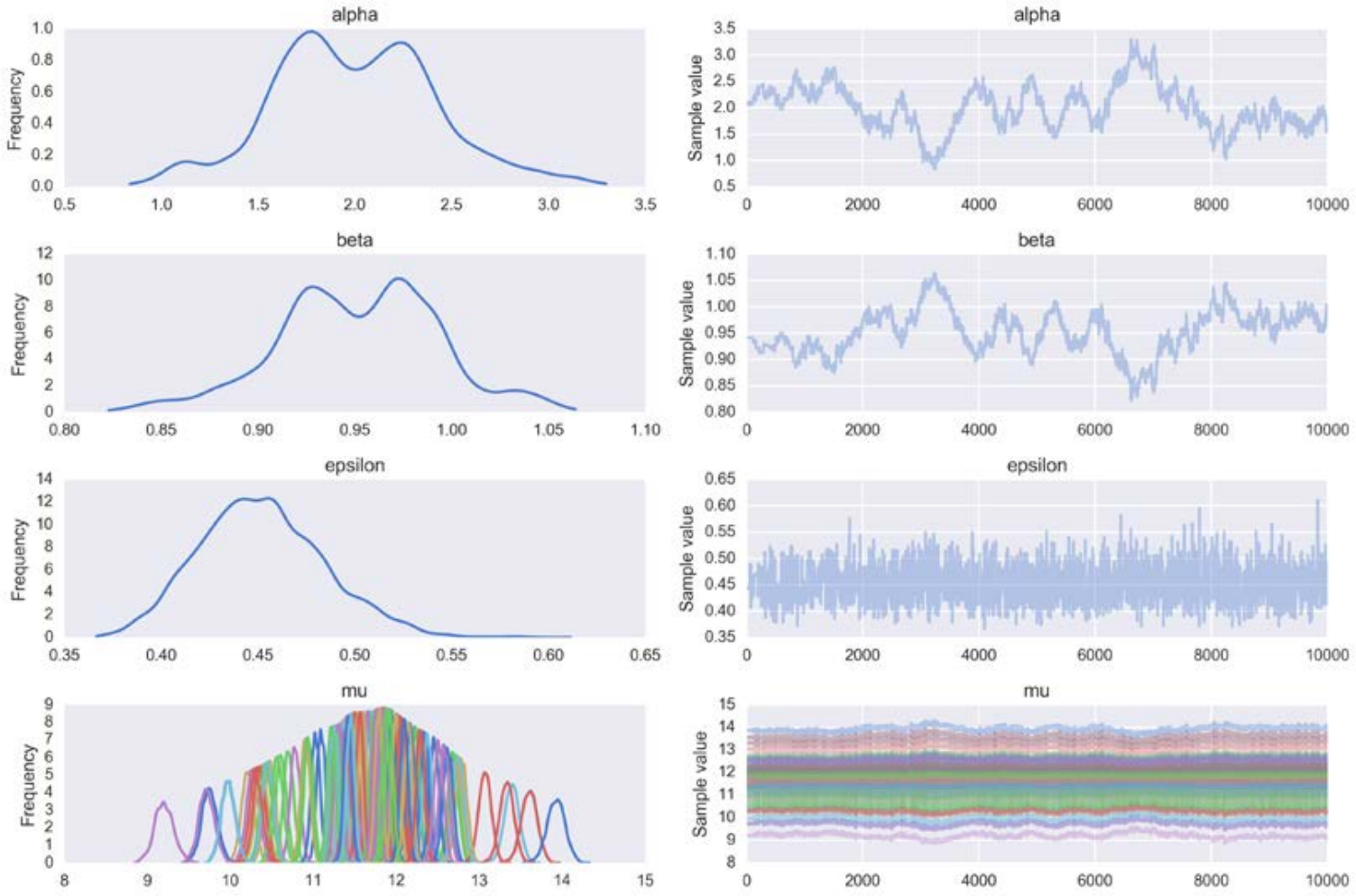
```
with pm.Model() as model:
    alpha = pm.Normal('alpha', mu=0, sd=10)
    beta = pm.Normal('beta', mu=0, sd=1)
    epsilon = pm.HalfCauchy('epsilon', 5)

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

    start = pm.find_MAP()
    step = pm.Metropolis()
    trace = pm.sample(10000, step, start)
```

Alternatively, we could have obviated specifying a deterministic variable and just written:

```
y_pred = pm.Normal('y_pred', mu= alpha + beta * x, sd=epsilon,
  observed=y)
```

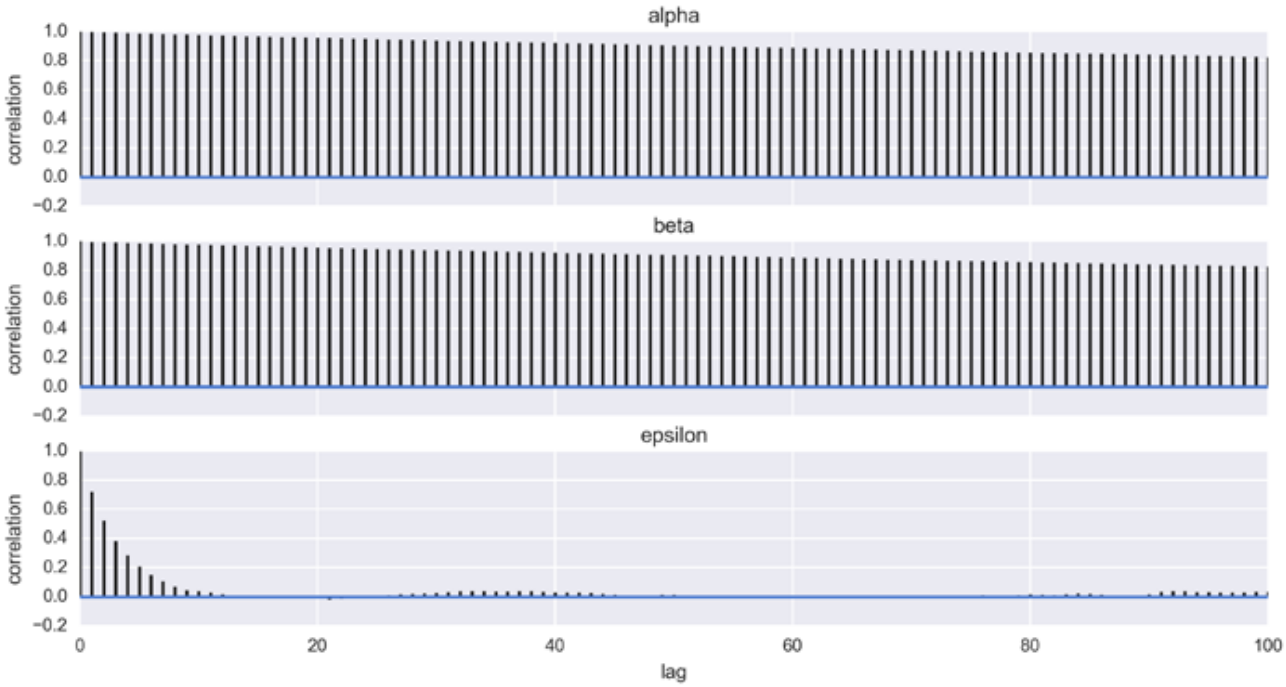
The `traceplot` does look a little bit funny. Notice how the trace for the `alpha` and `beta` parameters wanders up and down slowly; compare it with the KDE for epsilon, which is signaling good mixing. It is clear we have a problem.

```
pm.traceplot(chain);
```



The autocorrelation plot shows we have a lot of autocorrelation for $\alpha$ and $\beta$, and not for $\varepsilon$. As we will see next, this is the typical behavior for linear models. Notice that we omit plotting the deterministic variable $\mu$ by passing the `varnames` argument to the `autocorrplot` function:

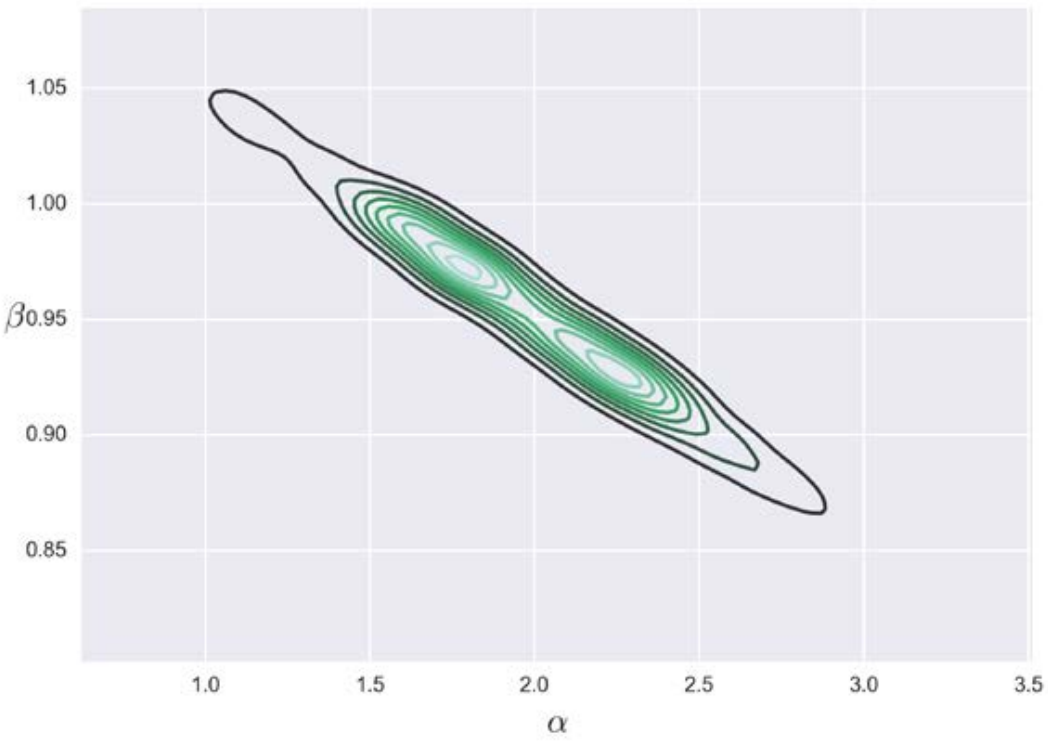
```
varnames = ['alpha', 'beta', 'epsilon']
pm.autocorrplot(trace, varnames)
```



# Linear models and high autocorrelation

So, we have a very ugly autocorrelation for $\alpha$ and $\beta$ and therefore we get very poor sampling and a low number of effective samples compared to the actual number of samples. That's a good diagnostic but what is really happening? Well, we are the victims of our own assumptions. No matter which line we fit to our data, all of them should pass for one point, 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. An increase in the slope means a decrease of the intercept and the other way around. Both parameters are going to be correlated by the definition of the model. This can be clearly seen if we plot the posterior (we are excluding $\varepsilon$ for clarity):

```
sns.kdeplot(trace['alpha'], trace['beta'])
plt.xlabel(r'$\alpha$', fontsize=16)
plt.ylabel(r'$\beta$', fontsize=16, rotation=0)
```




Hence, the shape of the posterior (excluding $\varepsilon$) is a very diagonal space. For algorithms such as Metropolis-Hastings, this can be very problematic. Because, if we propose large steps for individual parameters, most of the proposed values will fall outside of this high probability zone; if we propose small steps, they will need to be very really small to accept them. Either way we get a high autocorrelation in the sampling process and a very poor mixing. Even more, the higher the dimensionality of our data the more stressed this problem, because the total parameter space grows much faster than the plausible parameter space. For more information about this pervasive problem in data analysis check the Wikipedia entry for the curse of dimensionality.

Before continuing and for the sake of truth, let me clarify a point. 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 constrain 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 mean. Moreover, if we use strong priors we could end with lines far away from the mean of $x$ and $y$. Nevertheless, the idea that the autocorrelation is related to a line spinning around a more or less defined point remains true, and that is all we need to understand and fix the high autocorrelation problem, as we will see now using two different approaches.

# Modifying the data before running

One simple solution to our problem is to center the $x$ data. For each $x_i$ data point, we subtract the mean of the $x$ variable ($\overline{x}$):

$$x' = x - \overline{x}$$

As a result, $x'$ will be centered at *0*, and hence the pivot point when changing the slope is the intercept and the plausible parameter space is now more circular or less autocorrelated.

Centering data is not only a computational trick, it can also help in interpreting the data. 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 $x_i$. 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 hence the estimated intercept can provide us with valuable information; however, extrapolations have their caveats, 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:

$$\alpha = a' - \beta' \overline{x}$$

This correction is the result of the following algebraic reasoning:

$$\boldsymbol{x}' = \boldsymbol{x} - \overline{\boldsymbol{x}}$$
$$\boldsymbol{y} = a' + \beta' \boldsymbol{x}' + \varepsilon$$
$$\boldsymbol{y} = a' + \beta' \left( \boldsymbol{x} - \overline{\boldsymbol{x}} \right) + \varepsilon$$
$$\boldsymbol{y} = a' - \beta' \overline{\boldsymbol{x}} + \beta' \boldsymbol{x} + \varepsilon$$

Then it follows that:

$$\alpha = \alpha' + \beta' \overline{\boldsymbol{x}}$$
$$\beta = \beta'$$

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

$$\boldsymbol{x}' = \frac{\boldsymbol{x} - \overline{\boldsymbol{x}}}{\boldsymbol{x}_{sd}}$$

$$\boldsymbol{y}' = \frac{\boldsymbol{y} - \overline{\boldsymbol{y}}}{\boldsymbol{y}_{sd}}$$

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, because we have re-scaled the data! For standardized data, the intercept will always be around zero and the slope around *-1* and *1*. Standardizing the data allows us to talk in terms of Z-scores, that is, in units of standard deviation. If someone tells us that the value of a parameter is *-1.3 in Z-score units* we automatically know that the value is 1.3 standard deviations below the mean irrespective of the actual value of the mean and the actual value of the standard deviation of the data. A change in one 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 variables in the same scale can simplify the interpretation of the data.

# Changing the sampling method

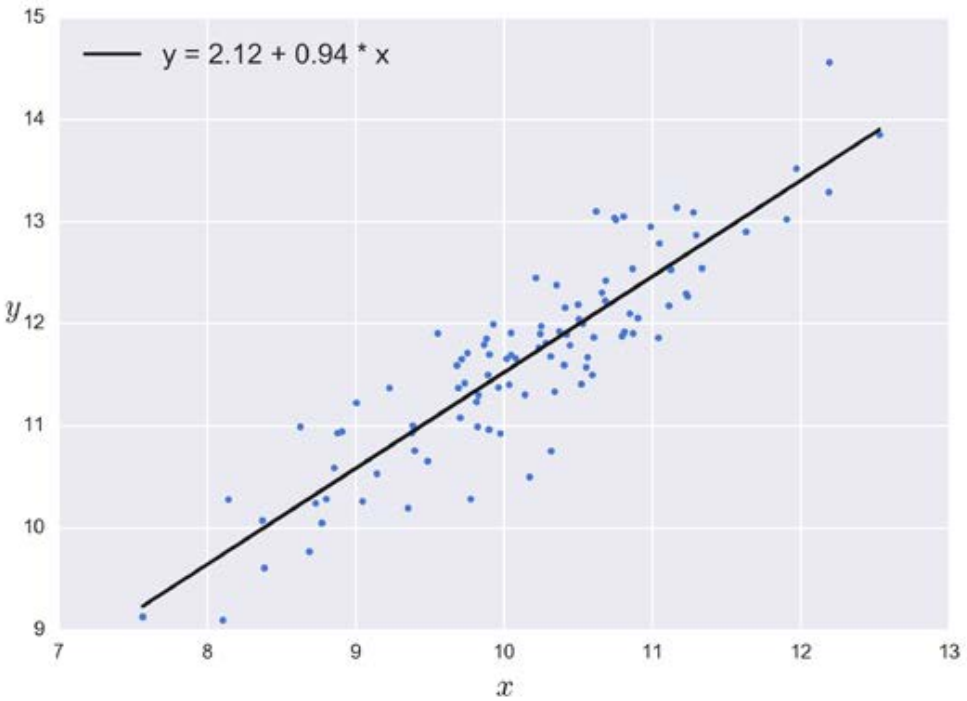
Another approach to ameliorating the autocorrelation problem is to use a different sampling method. NUTS has fewer difficulties than Metropolis in sampling such restricted diagonal spaces. The reason is that NUTS moves according to the curvature of the posterior and hence it is easier for it to move along a diagonal space. As we have already seen, NUTS can be slower than Metropolis per step, but usually needs far fewer steps to get a reasonable approximation to the posterior.

The following section corresponds to results obtained from the previous model and using the NUTS sampler instead of Metropolis.

# Interpreting and visualizing the posterior

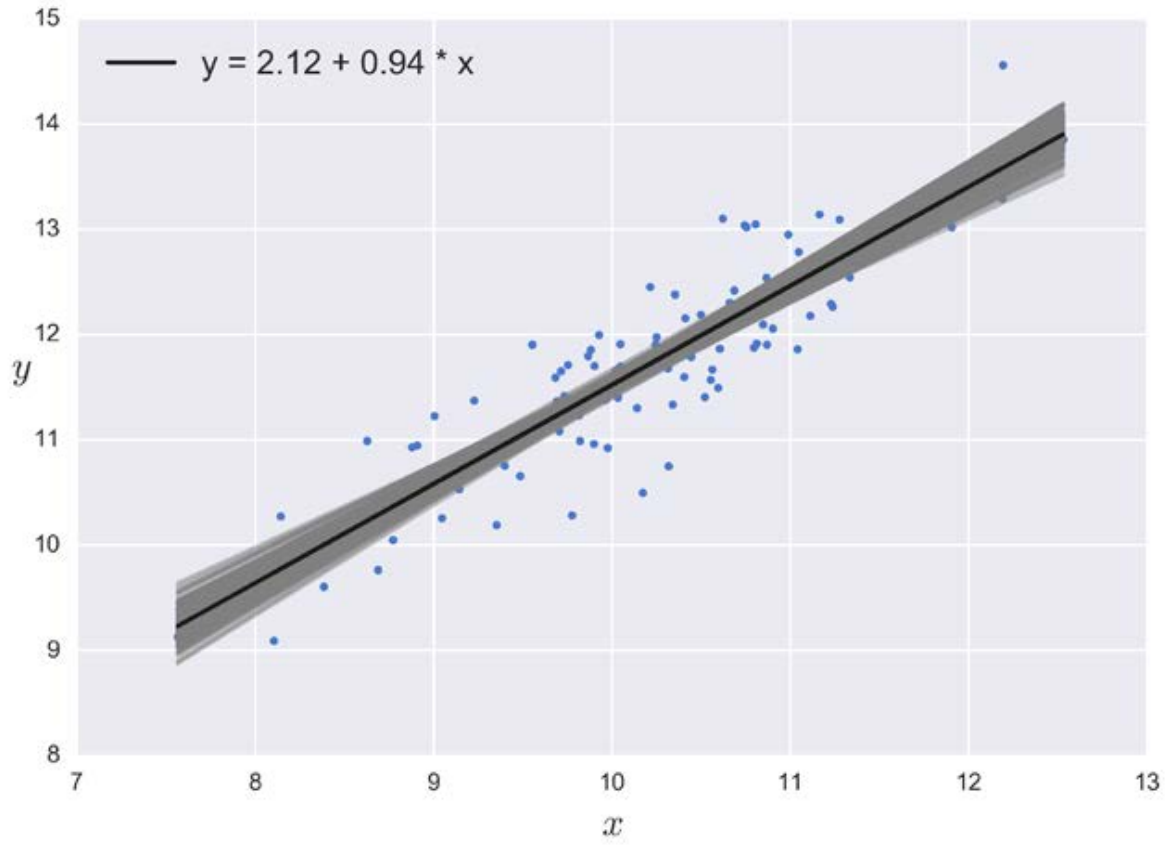
As we have already seen, we can explore the posterior using the PyMC3 functions, `traceplot` and `df_summary`, or we can use our own functions. For a linear regression it could be useful to plot the average line that fits the data together with the average mean values of $\alpha$ and $\beta$:

```
plt.plot(x, y, 'b.');
alpha_m = trace_n['alpha'].mean()
beta_m = trace_n['beta'].mean()
plt.plot(x, alpha_m + beta_m * x, c='k', label='y = {:.2f} +
  {:.2f} * x'.format(alpha_m, beta_m))
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$y$', fontsize=16, rotation=0)
plt.legend(loc=2, fontsize=14)
```



Alternatively, we can also make a plot reflecting the posterior uncertainty using semitransparent lines sampled from the posterior:

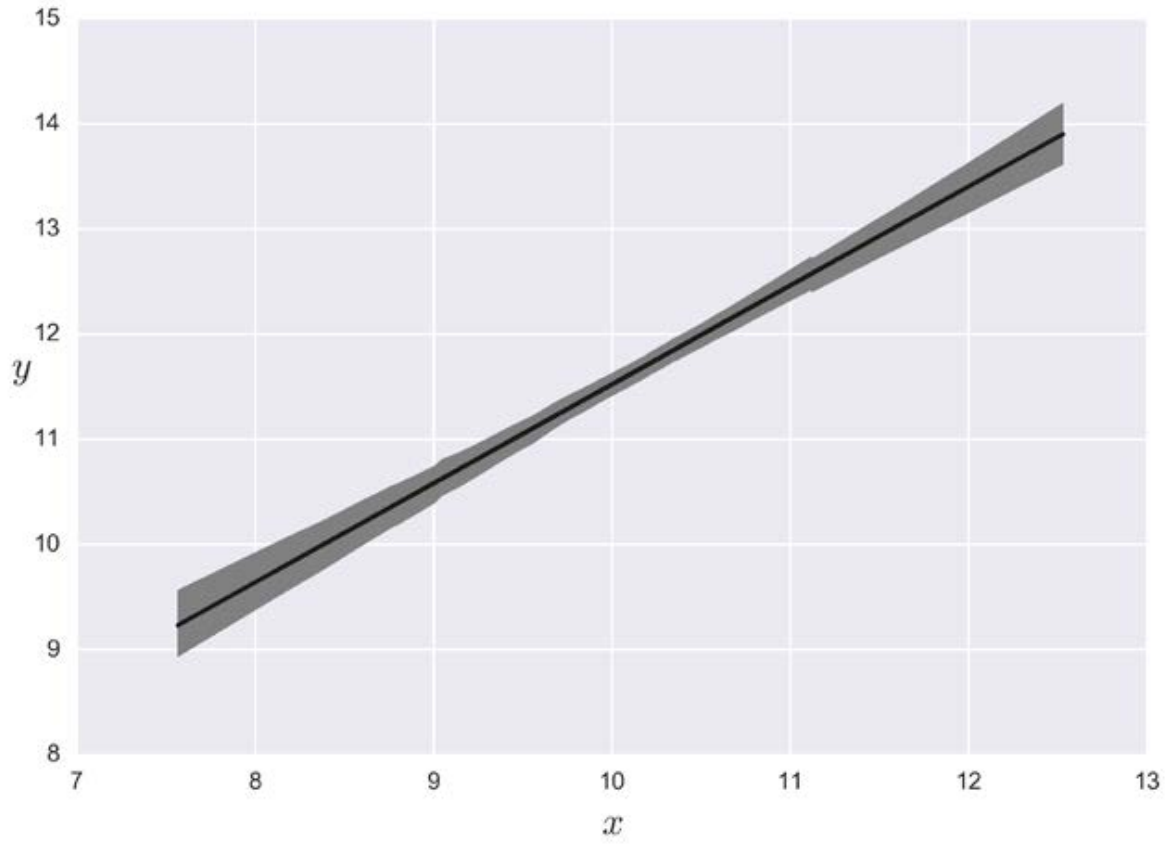
```
plt.plot(x, y, 'b.');

idx = range(0, len(trace_n['alpha']), 10)
plt.plot(x, trace_n['alpha'][idx] + trace_n['beta'][idx] *
  x[:,np.newaxis], c='gray', alpha=0.5);

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

plt.xlabel('$x$', fontsize=16)
plt.ylabel('$y$', fontsize=16, rotation=0)
plt.legend(loc=2, fontsize=14)
```




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 for 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 use instead a semitransparent band to illustrate the **Highest Posterior Density** (**HPD**) interval of $\mu$. Notice this was the main reason for defining the variable `mu` as a deterministic in the model, just to simplify the following code:

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

idx = np.argsort(x)
x_ord = x[idx]
sig = pm.hpd(trace_n['mu'], alpha=.02)[idx]
plt.fill_between(x_ord, sig[:,0], sig[:,1], color='gray')

plt.xlabel('$x$', fontsize=16)
plt.ylabel('$y$', fontsize=16, rotation=0)
```

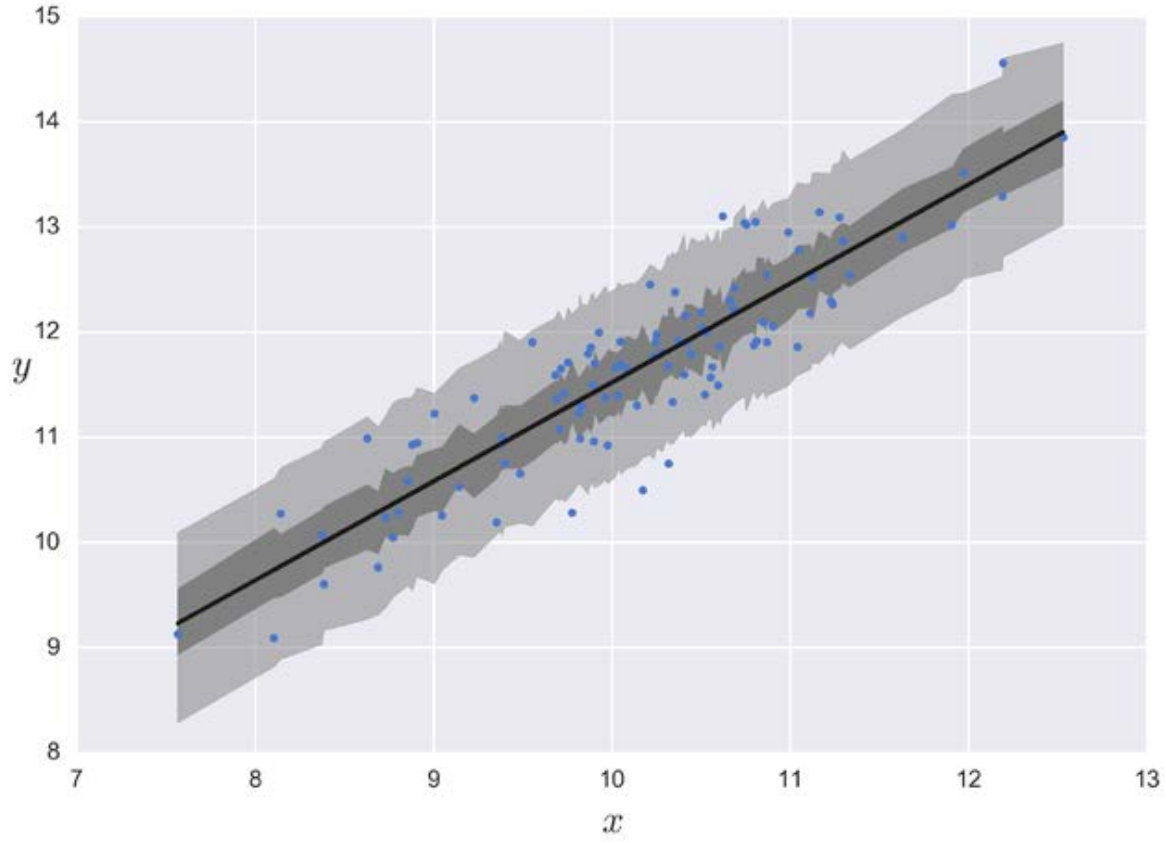



One more option is to plot the HPD (*95* and *50*, for example) of the predicted data $\hat{y}$, that is, where we expect to see the *95%* and *50%* of future data, according to our model. For the figure we are going to use a darker gray for the HPD *50* and a lighter gray for the HPD *95*. Getting the posterior predictive samples is easy in PyMC3 using the `sample_ppc` function:

```
ppc = pm.sample_ppc(chain_n, samples=1000, model=model_n)
```

And now we plot the results:

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

sig0 = pm.hpd(ppc['y_pred'], alpha=0.5)[idx]
sig1 = pm.hpd(ppc['y_pred'], alpha=0.05)[idx]
plt.fill_between(x_ord, sig0[:,0], sig0[:,1], color='gray', alpha=1)
plt.fill_between(x_ord, sig1[:,0], sig1[:,1], color='gray', alpha=0.5)

plt.xlabel('$x$', fontsize=16)
plt.ylabel('$y$', fontsize=16, rotation=0)
```




The irregular look of the boundaries of the HPD comes from the fact that, to make this plot, we are taking posterior predictive samples from the observed values of $x$ and not from a continuous interval, and also from the fact that `fill_between` is just doing a simple linear interpolation between successive points. Notice how the plot is more serrated the more data points we have. The serrated aspect can be minimized by taking more samples from `y_pred` (try it, for example, with *10,000* samples).

# Pearson correlation coefficient

Many times 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 just with a lowercase *r*. When the value of *r* is *+1* we have a perfect positive linear correlation, that is, an increase of one variable predicts an increase of the other. When we have *-1*, we have a perfect negative linear correlation and the increase of one variable predicts a decrease of the other. When *r* is *0* we have no linear correlation. The Pearson correlation coefficient says nothing about non-linear correlations. It is easy to confuse *r* with the slope of a regression. Check the following very nice image from Wikipedia showing that both quantities are not necessarily the same thing: `https://en.wikipedia.org/wiki/Correlation_and_dependence#/media/File:Correlation_examples2.svg`.

Part of the confusion may be explained by the following relationship:

$$r = \beta \frac{\sigma(x)}{\sigma(y)}$$

That is, the slope and the Pearson correlation coefficient have the same value only when the standard deviation of $x$ and $y$ are equal. Notice that it is true, for example, when we standardize the data. Just to clarify:

- The Pearson correlation coefficient is a measure of the degree of correlation between two variables and is always restricted to the interval *[-1, 1]*. The scale of the data is irrelevant.
- The slope 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 it is just the square of the Pearson coefficient, that is, $r^2$ (or $R^2$ and pronounced just as *r squared*). The determination coefficient can be interpreted as the proportion of the variance in the dependent variable that is predictable from the independent variable.

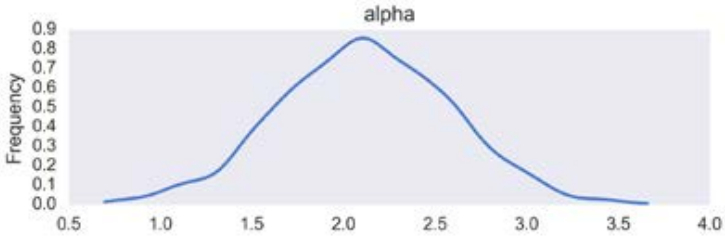
Now we are going to see how to compute *r* and $r^2$ with PyMC3 by extending the simple linear regression model. And we are going to do it in two different ways:

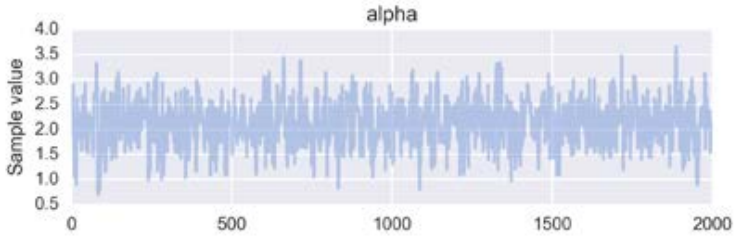
- One way is to use the equation we just saw, relating the slope and the Pearson correlation coefficient. See the deterministic variable `rb`.

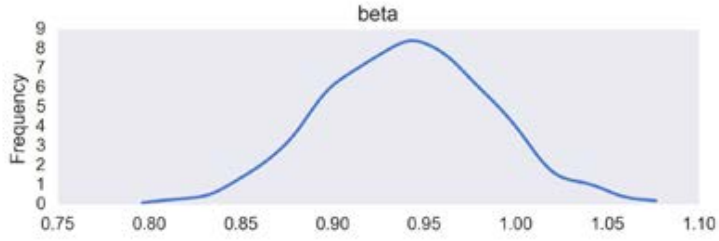
- The other way is related to the least squares method and we are going to skip the details of its derivation. See the deterministic variable `rss`. If we check the code, we will see the variable `ss_reg`. This is a measure of the dispersion between the fitted line and the mean of the data, and is proportional to the variance in the model. Notice that the formula resembles the once for the variance; the difference is that we are not dividing by the number of data points. The variable `ss_tot` is proportional to the variance of the predicted variable.

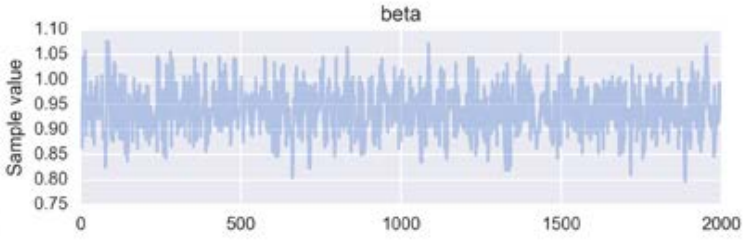
Then, the full mode is:

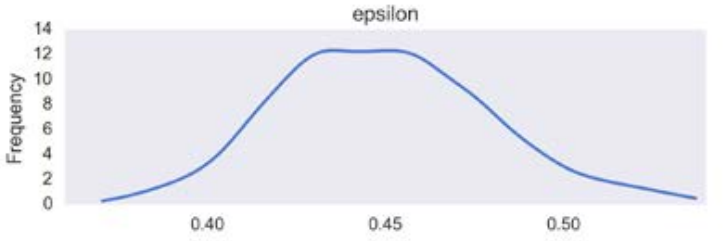
```
with pm.Model() as model_n:
    alpha = pm.Normal('alpha', mu=0, sd=10)
    beta = pm.Normal('beta', mu=0, sd=1)
    epsilon = pm.HalfCauchy('epsilon', 5)

    mu = alpha + beta * x
    y_pred = pm.Normal('y_pred', mu=mu, sd=epsilon, observed=y)

    rb = pm.Deterministic('rb', (beta * x.std() / y.std()) ** 2)

    y_mean = y.mean()
    ss_reg = pm.math.sum((mu - y_mean) ** 2)
    ss_tot = pm.math.sum((y - y_mean) ** 2)
    rss = pm.Deterministic('rss', ss_reg/ss_tot)

    start = pm.find_MAP()
    step = pm.NUTS()
    trace_n = pm.sample(2000, step=step, start=start)
pm.traceplot(chain_n)
```

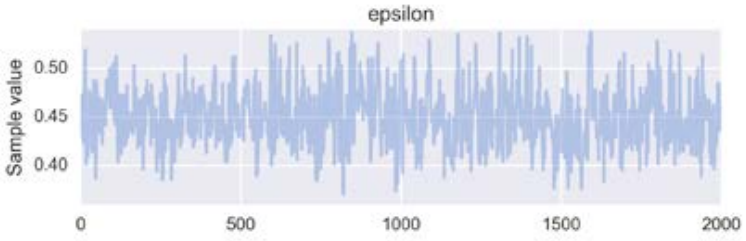


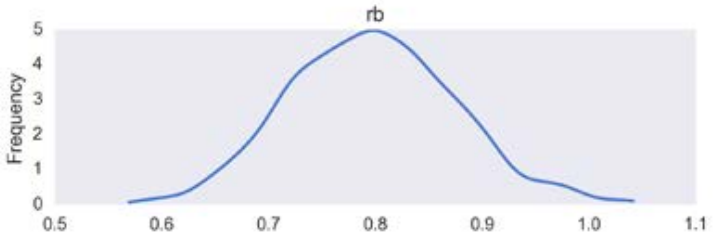



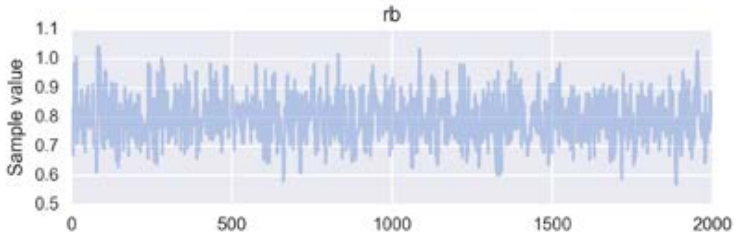



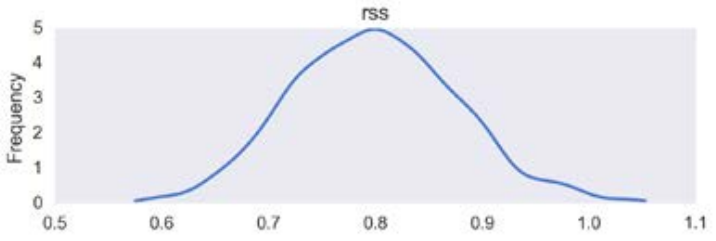



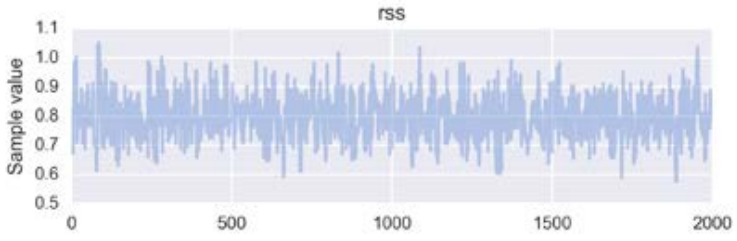




```
pm.df_summary(cadena_n, varnames)
```

| | mean | sd | mc_error | hpd_2.5 | hpd_97.5 |
|---|---|---|---|---|---|
| alpha | 2.11 | 0.49 | 1.87e-02 | 1.21 | 3.13 |
| beta | 0.94 | 0.05 | 1.82e-03 | 0.84 | 1.03 |
| epsilon | 0.45 | 0.03 | 1.30e-03 | 0.39 | 0.52 |
| rb | 0.80 | 0.08 | 3.09e-03 | 0.64 | 0.96 |
| rss | 0.80 | 0.08 | 3.22e-03 | 0.64 | 0.95 |

# 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 *2* means (or a vector with two elements), one for each marginal Gaussian. We also need *2* standard deviations, right? Well not exactly; we need a *2 x 2* **covariance matrix**, which looks like this:

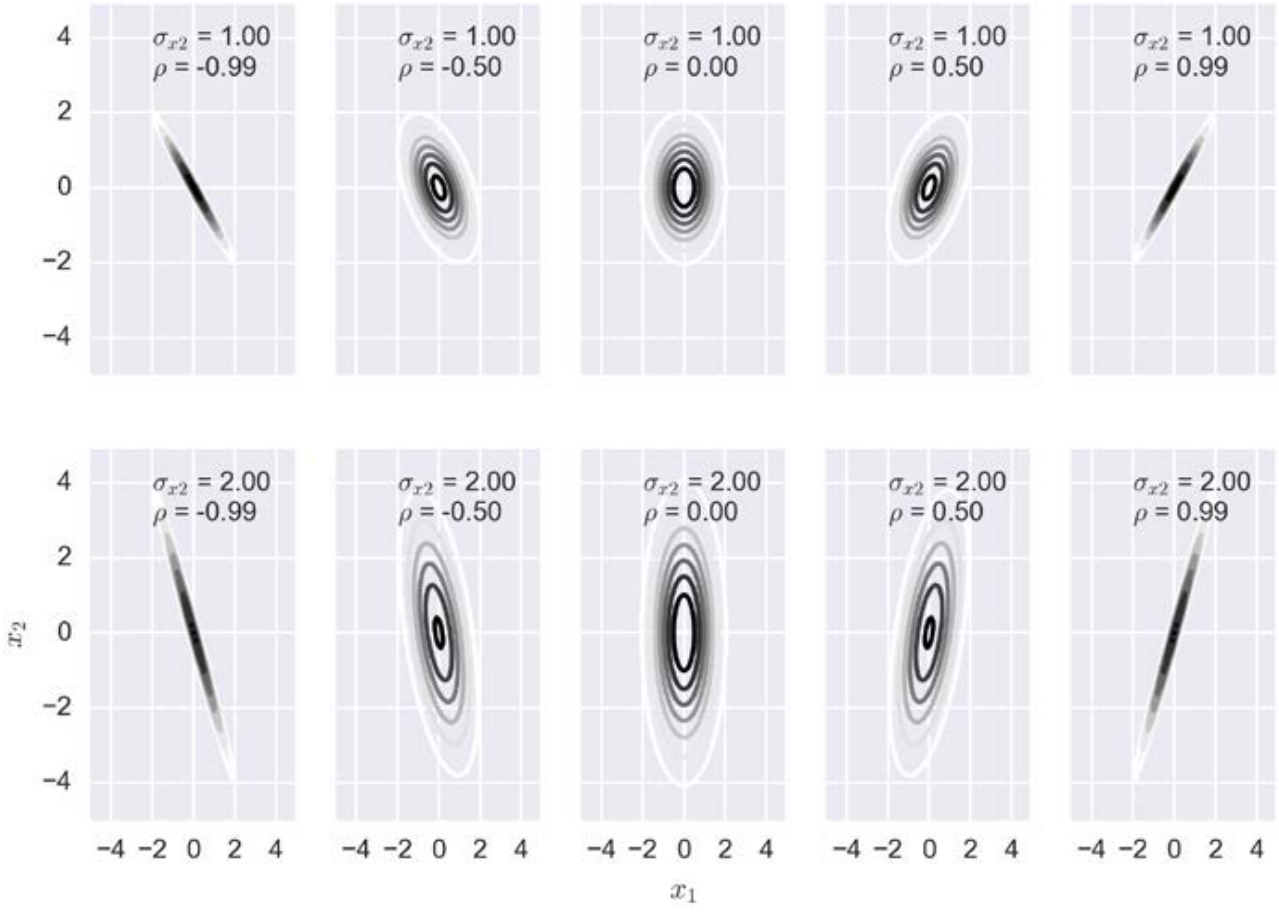
$$\sum = \begin{bmatrix} \sigma_{x1}^2 & \rho\sigma_{x2}\sigma_{x1} \\ \rho\sigma_{x2}\sigma_{x1} & \sigma_{x2}^2 \end{bmatrix}$$

Where $\sum$ is the Greek capital sigma letter; it is a 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 $\sigma_{x1}$ and $\sigma_{x2}$. The rest of the elements in the matrix are the covariances (the variance between variables) expressed in terms of the individual standard deviations and $\rho$, the Pearson correlation coefficient between variables. Notice we have a single $\rho$ because we have only two dimensions/variables. For three variables we would have 3 Pearson coefficients. For 4 variables we will get 6. To compute these numbers you can use the binomial coefficient, remember the definition of the binomial distribution from *Chapter 1*, *Thinking Probabilistically - A Bayesian Inference Primer*.

The following code generates contour plots for bivariate Gaussian distributions with both means fixed at *0*, *0*, one standard deviation fixed at *1*, the other taking the values *1* or *2*, and different values for the Pearson correlation coefficient.

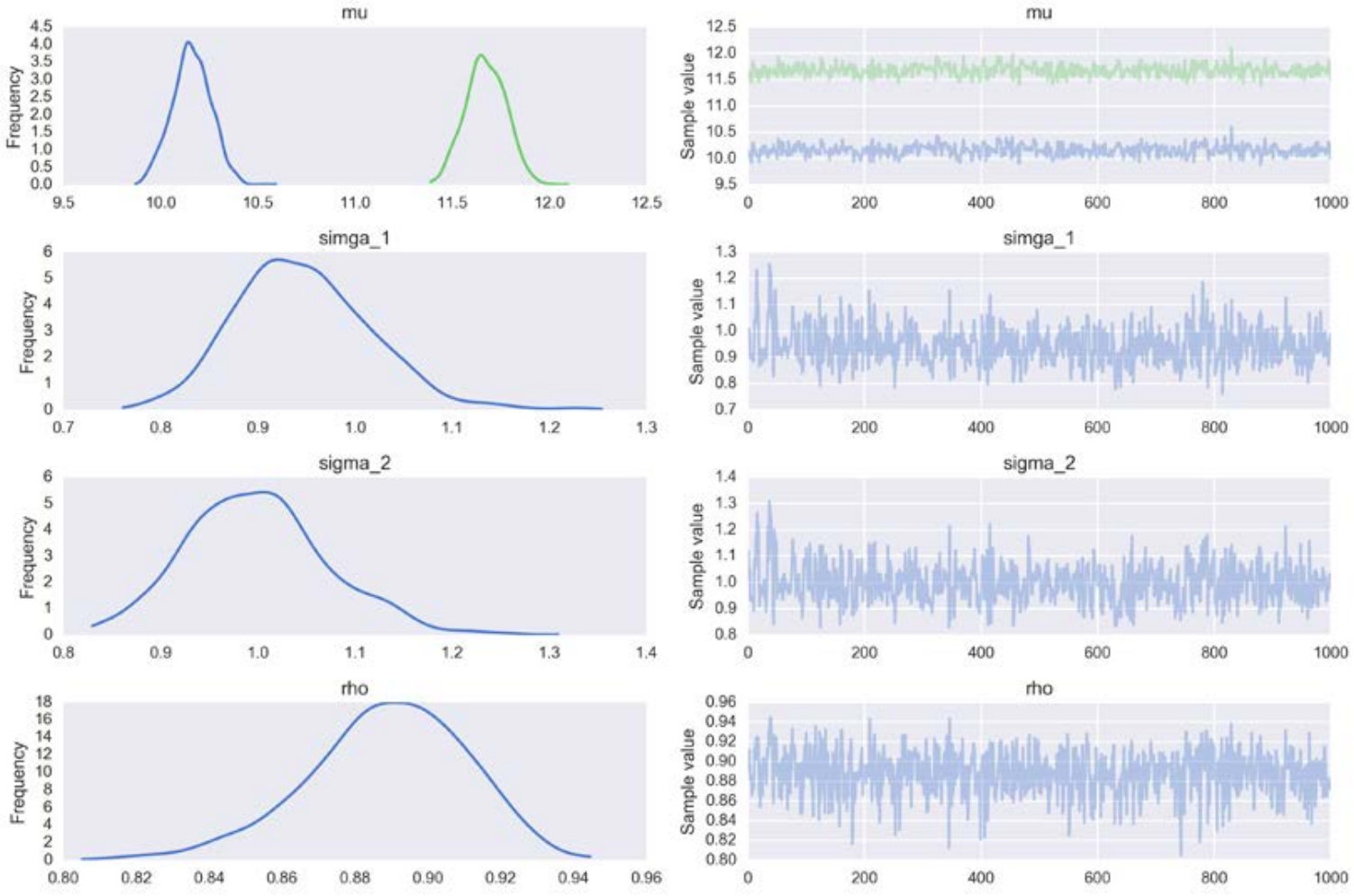
```
sigma_x1 = 1
sigmas_x2 = [1, 2]
rhos = [-0.99, -0.5, 0, 0.5, 0.99]

x, y = np.mgrid[-5:5:.1, -5:5:.1]
pos = np.empty(x.shape + (2,))
pos[:, :, 0] = x; pos[:, :, 1] = y

f, ax = plt.subplots(len(sigmas_x2), len(rhos), sharex=True,
  sharey=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(x, y, rv.pdf(pos))
        ax[i,j].plot(0, 0,
        label="$\\sigma_{{x2}}$ = {:3.2f}\n$\\rho$ =
         {:3.2f}".format(sigma_x2, rho), alpha=0)
        ax[i,j].legend()
ax[1,2].set_xlabel('$x_1$')
ax[1,0].set_ylabel('$x_2$')
```



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 as the generalization of the chi squared distribution. A second option is to use the LKJ prior; 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 $\sigma_{x1}$, $\sigma_{x2}$ and $\rho$ and then use those values to manually build the covariance matrix.

```
with pm.Model() as pearson_model:

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

    sigma_1 = pm.HalfNormal('simga_1', 10)
    sigma_2 = pm.HalfNormal('sigma_2', 10)
    rho = pm.Uniform('rho', -1, 1)

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

    y_pred = pm.MvNormal('y_pred', mu=mu, cov=cov, observed=data)

    start = pm.find_MAP()
    step = pm.NUTS(scaling=start)
    trace_p = pm.sample(1000, step=step, start=start)
```




Notice that we are getting $\rho$ the Pearson coefficient, an in the previous example we got the squared of the Pearson coefficient. Taking this into account you will see that the values obtained in this and the previous example closely match.

# Robust linear regression

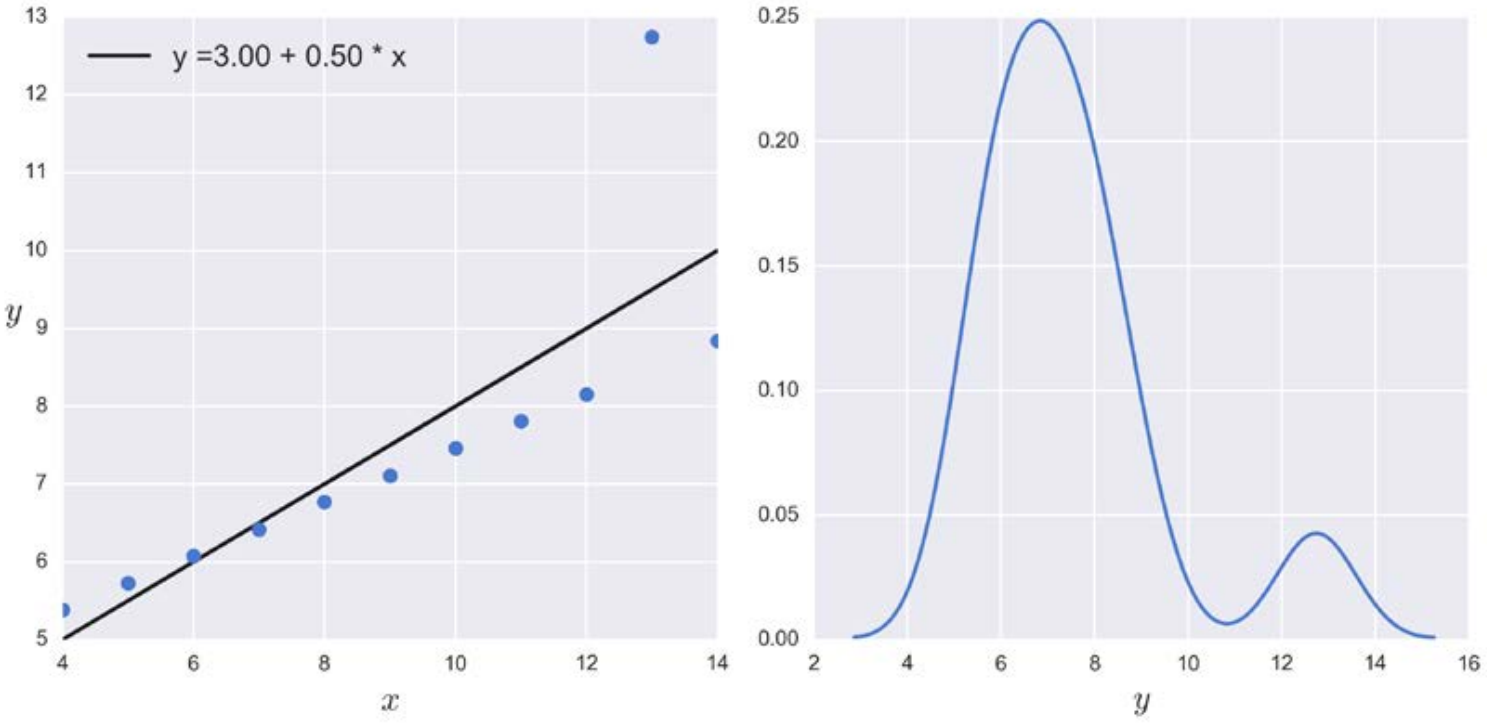
Assuming our data follows a Gaussian distribution is perfectly reasonable in many situations. By assuming Gaussianity, we are not necessarily saying that our data is really Gaussian; instead we are saying that it is a reasonable approximation for our current problem. As we saw in the previous chapter, 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. We can upload it from `seaborn`:

```
ans = sns.load_dataset('anscombe')
x_3 = ans[ans.dataset == 'III']['x'].values
y_3 = ans[ans.dataset == 'III']['y'].values
```

And now let's check what this tiny dataset looks like:

```
plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
beta_c, alpha_c = stats.linregress(x_3, y_3)[:2]
plt.plot(x_3, (alpha_c + beta_c* x_3), 'k', label='y ={:.2f} + {:.2f}
* x'.format(alpha_c, beta_c))
plt.plot(x_3, y_3, 'bo')
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$y$', rotation=0, fontsize=16)
plt.legend(loc=0, fontsize=14)
plt.subplot(1,2,2)
sns.kdeplot(y_3);
plt.xlabel('$y$', fontsize=16)
```



Now we are going to rewrite the model using a Student's t-distribution instead of a Gaussian. This change also introduces the need to specify the value of $\nu$, the normality parameter. If you do not remember the role of this parameter check the previous chapter before continuing.

Also in the following model we are using a shifted exponential, to avoid values of nu 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 non-to-moderate outliers but for data with extreme outliers (or data with few bulk points), as in the Anscombe's third data set, it is better to avoid such low values. Take this, as well as other prior recommendations, with a pinch of salt. These recommendations are generally observations based on some data sets (and problems) I (or others) have worked on and may not apply to your problems and data sets. Other common priors for $\nu$ are *gamma(2, 0.1)* or *gamma(mu=20, sd=15)*.

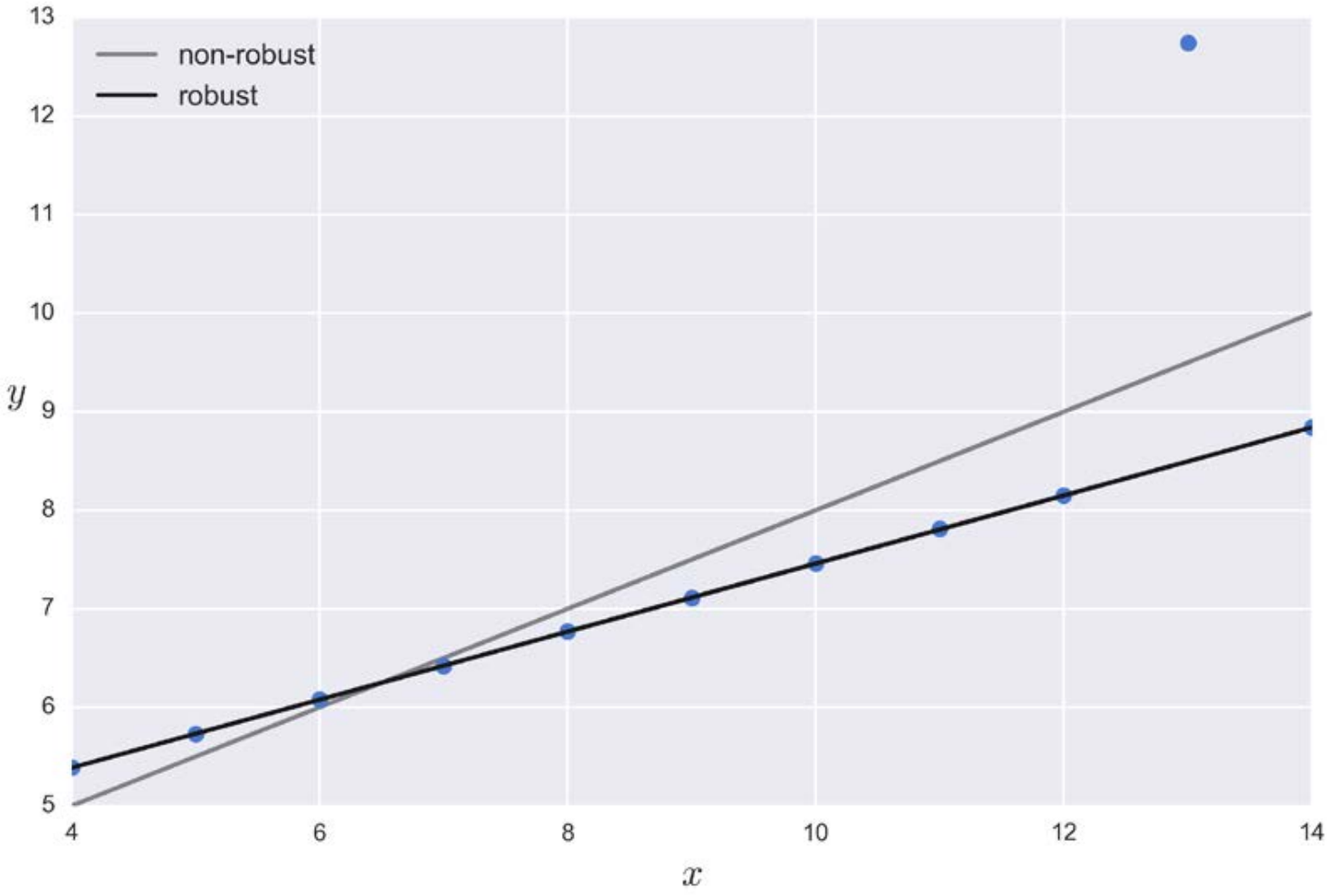
As a side note, by adding an underscore _, to the end of a PyMC3 variable, as in `nu_`, the variable is hidden from the user.

```
with pm.Model() as model_t:
    alpha = pm.Normal('alpha', mu=0, sd=100)
    beta = pm.Normal('beta', mu=0, sd=1)
    epsilon = pm.HalfCauchy('epsilon', 5)
    nu = pm.Deterministic('nu', pm.Exponential('nu_', 1/29) + 1)

    y_pred = pm.StudentT('y_pred', mu=alpha + beta * x_3,
      sd=epsilon, nu=nu, observed=y_3)

    start = pm.find_MAP()
    step = pm.NUTS(scaling=start)
    trace_t = pm.sample(2000, step=step, start=start)
```

I am going to skip some code and plots here (just to save space). I am going to omit plots such as the `traceplot` and the autocorrelation plot, but you should not. Instead I am going to plot just the mean fitted line. I am using also the SciPy `linregress` (frequentist) method to plot the non-robust line. You may try plotting the line obtained using the Bayesian method from the previous example.

```
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, 'bo')
alpha_m = trace_t['alpha'].mean()
beta_m = trace_t['beta'].mean()
```

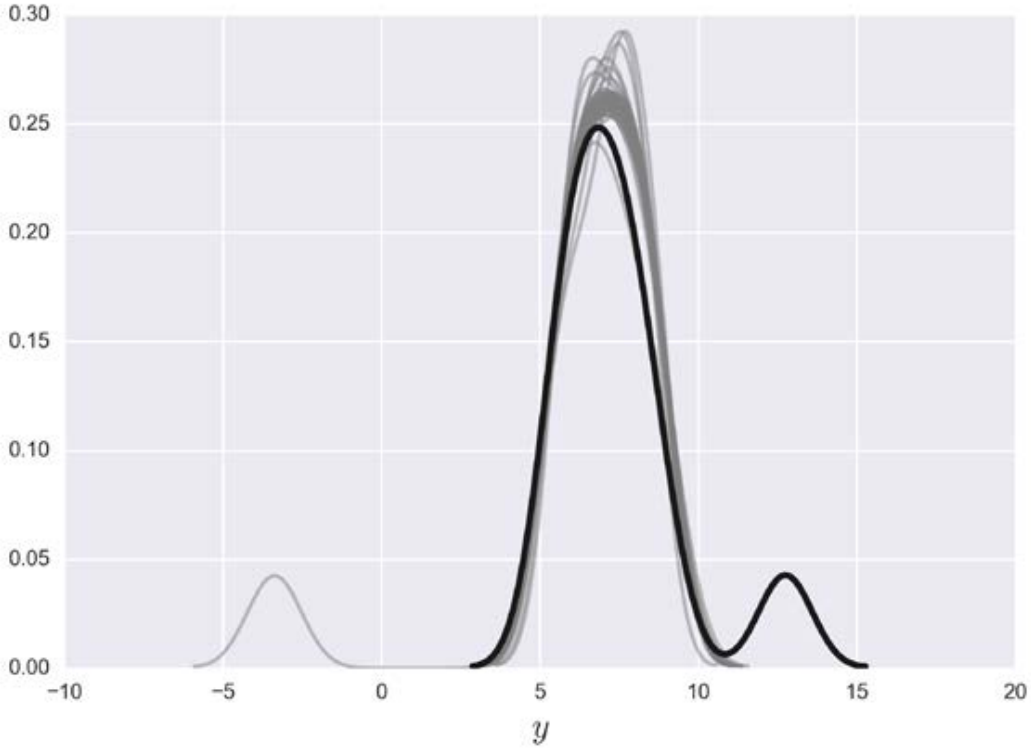
```
plt.plot(x_3, alpha_m + beta_m * x_3, c='k', label='robust')

plt.xlabel('$x$', fontsize=16)
plt.ylabel('$y$', rotation=0, fontsize=16)
plt.legend(loc=2, fontsize=12)
```




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_ppc(chain_t, samples=200, model=model_t, random_
seed=2)
for y_tilde in ppc['y_pred']:
    sns.kdeplot(y_tilde, alpha=0.5, c='g')
sns.kdeplot(y_3, linewidth=3)
```

You should see something like this:




For the bulk of the data, we get a very good match. Also notice that our model predicts values away from the bulk 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 negative values. In such a case, we should probably go back and change the model and restrict the possible values of $y$ to positive values.

# Hierarchical linear regression

In the previous chapter, we learned the rudiments of hierarchical models. We can apply these concepts to linear regression and model several groups at the same time including estimations at the group level and estimations above the group level. As we saw, this is done by including hyperpriors.

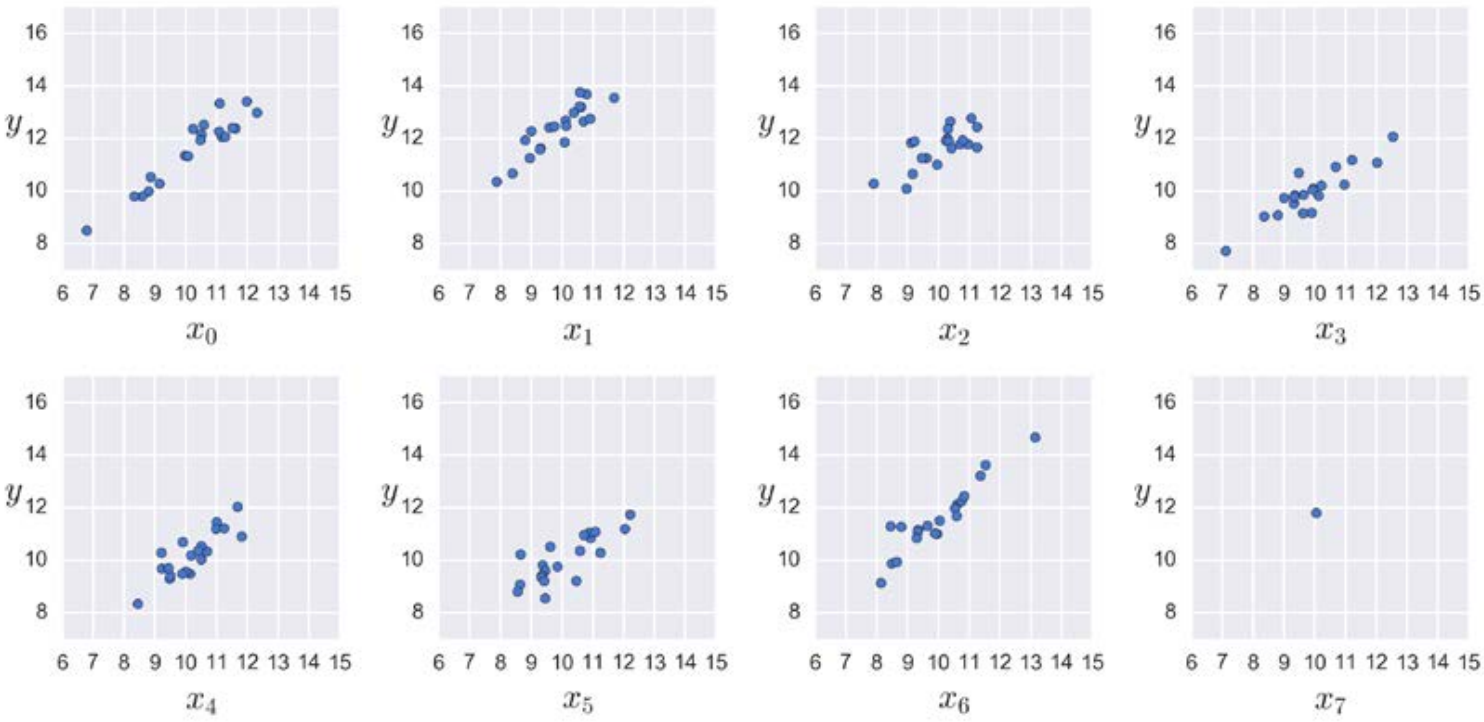
We are going to create eight related data groups, including one with just one data point:

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

alpha_real = np.random.normal(2.5, 0.5, size=M)
```

```
beta_real = np.random.beta(60, 10, 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
```

Our data looks like this:

```
j, k = 0, N
for i in range(M):
    plt.subplot(2,4,i+1)
    plt.scatter(x_m[j:k], y_m[j:k])
    plt.xlim(6, 15)
    plt.ylim(7, 17)
    j += N
    k += N
plt.tight_layout()
```




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

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

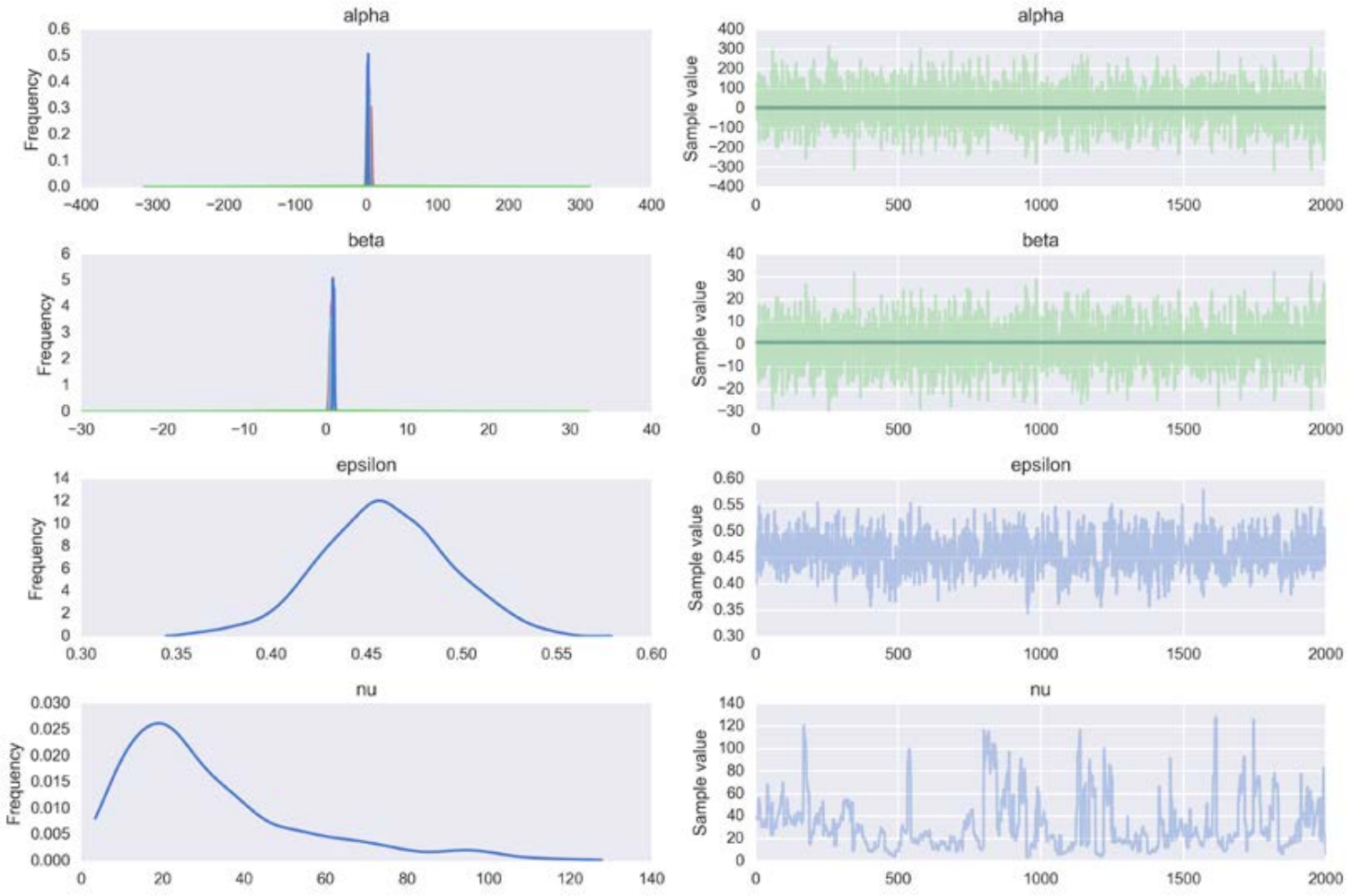
First we are going to fit a non-hierarchical model, just as we have already seen. The only difference is that now we are going to include code to re-scale alpha to the original scale.

```
with pm.Model() as unpooled_model:
    alpha_tmp = pm.Normal('alpha_tmp', mu=0, sd=10, shape=M)
```

```
    beta = pm.Normal('beta', mu=0, sd=10, shape=M)
   epsilon = pm.HalfCauchy('epsilon', 5)

    nu = pm.Exponential('nu', 1/30)

    y_pred = pm.StudentT('y_pred', mu=alpha_tmp[idx] + beta[idx]
* x_centered, sd=epsilon, nu=nu, observed=y_m)

    alpha = pm.Deterministic('alpha', alpha_tmp - beta *
      x_m.mean())

    start = pm.find_MAP()
    step = pm.NUTS(scaling=start)
    trace_up = pm.sample(2000, step=step, start=start)
```

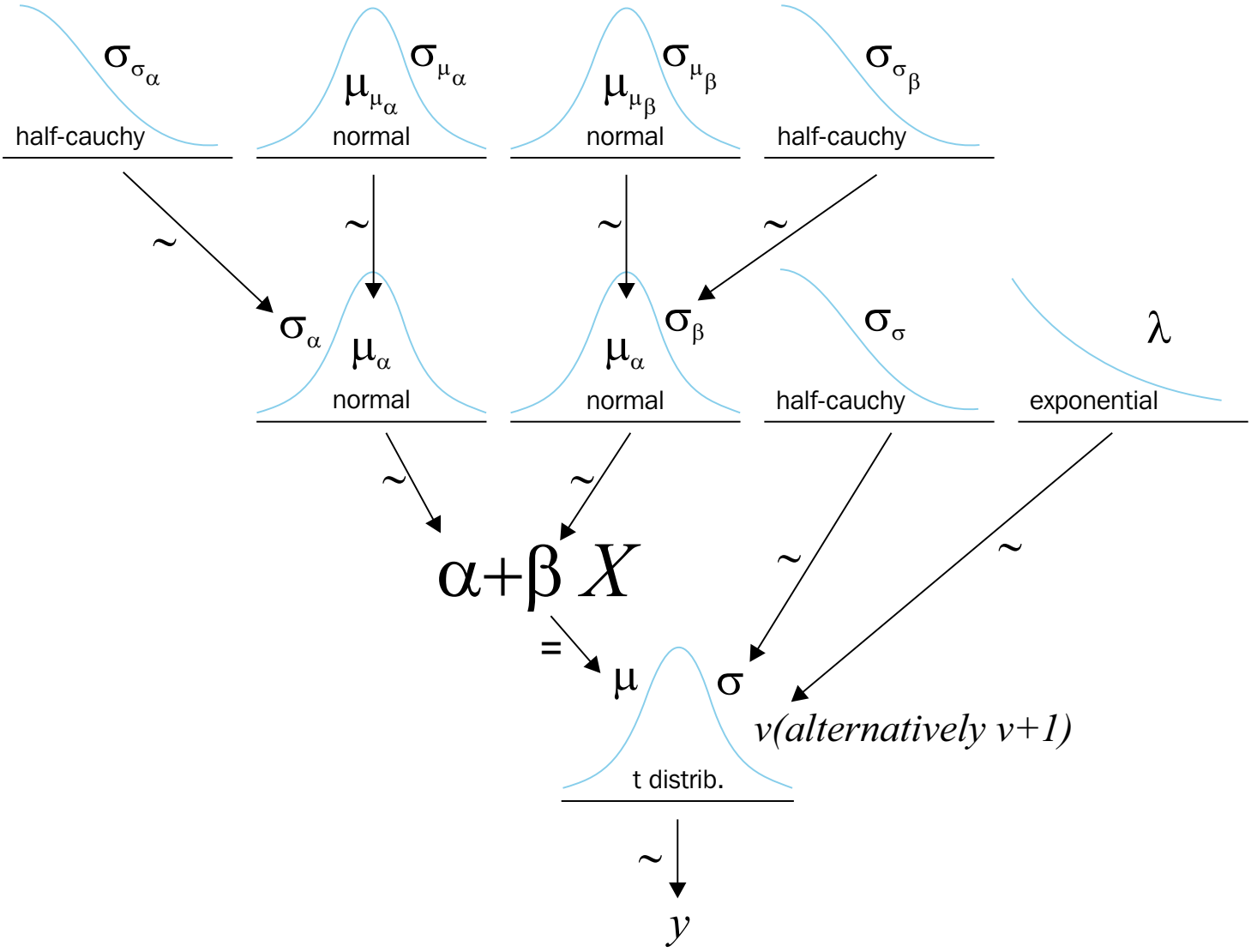
Now let's check the results. As we can see, everything looks fine except for one alpha and one beta parameter. We can see from the trace that for one case the trace is not converging and is just wandering freely:

```
varnames=['alpha', 'beta', 'epsilon', 'nu']
pm.traceplot(trace_up, varnames);
```



You may have already guessed what is going on. Of course it does not make sense to try to fit a unique line through only one point. We need at least two points, otherwise the parameters `alpha` and `beta` are unbounded. That is totally true unless we provide some more information; we can do this by using priors. Putting a strong prior for alpha 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. 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), just to make the message clear.

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:



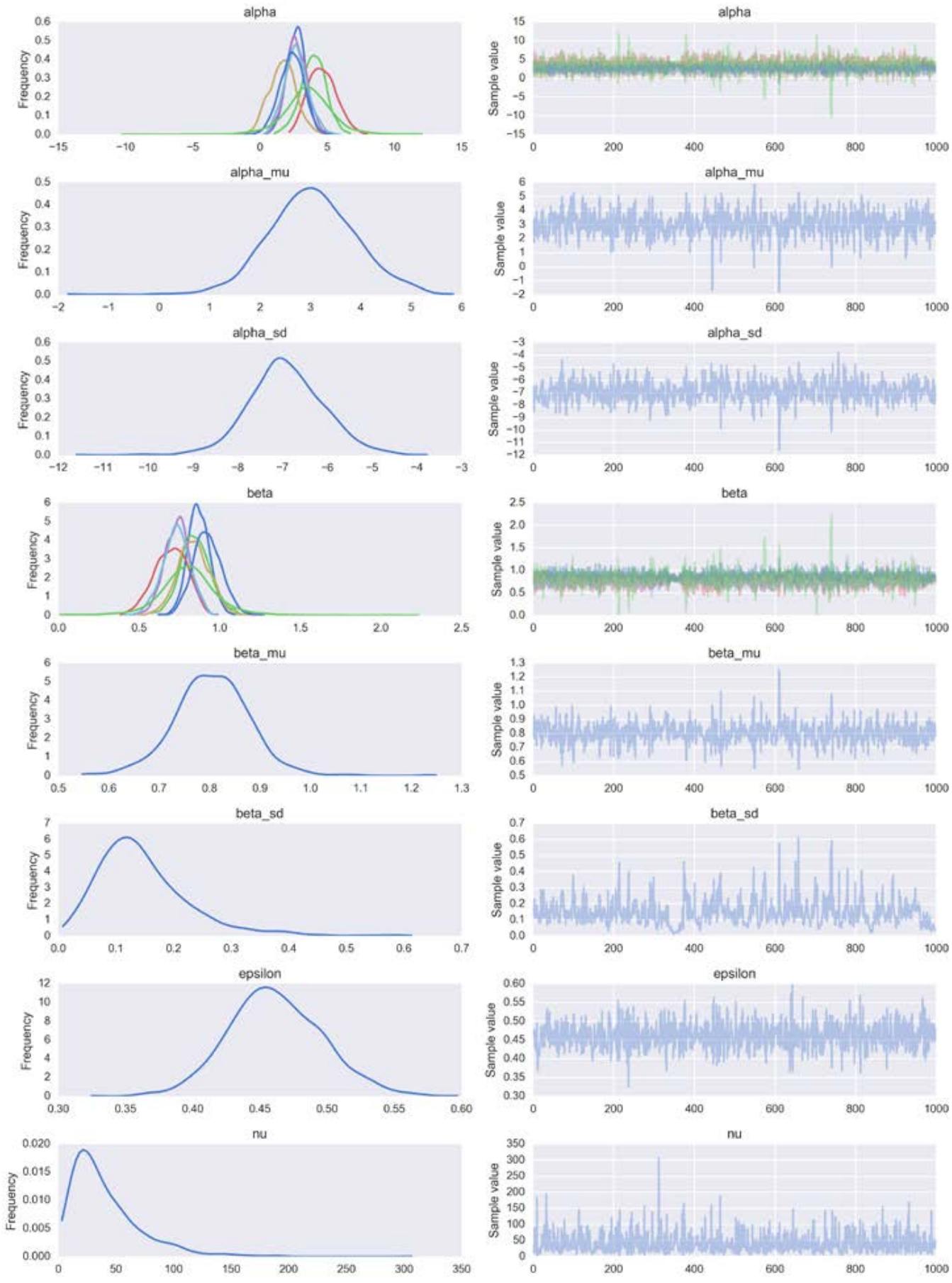
In the following PyMC3 model the main differences with the previous model are:

- We add the hyperpriors.
- We also add a few lines to transform the parameters back to the original uncentered scale. Remember this is not mandatory; we can keep the parameters in the transformed scale but we should be careful when interpreting the results.
- We use ADVI, instead of `find_MAP()`, to initialize and provide a covariance scaling matrix for NUTS. If you do not remember about ADVI go back and check *Chapter 2, Programming Probabilistically – A PyMC3 Primer* for a short theoretical description. Also check the exercise section of this chapter for a more practical discussion.

```
with pm.Model() as hierarchical_model:
    alpha_tmp_mu = pm.Normal('alpha_tmp_mu', mu=0, sd=10)
    alpha_tmp_sd = pm.HalfNormal('alpha_tmp_sd', 10)
    beta_mu = pm.Normal('beta_mu', mu=0, sd=10)
    beta_sd = pm.HalfNormal('beta_sd', sd=10)



    alpha_tmp = pm.Normal('alpha_tmp', mu=alpha_tmp_mu, sd=alpha_
tmp_sd, shape=M)
    beta = pm.Normal('beta', mu=beta_mu, sd=beta_sd, shape=M)
    epsilon = pm.HalfCauchy('epsilon', 5)
    nu = pm.Exponential('nu', 1/30)

    y_pred = pm.StudentT('y_pred', mu=alpha_tmp[idx] + beta[idx] *
x_centered, sd=epsilon, nu=nu, observed=y_m)

    alpha = pm.Deterministic('alpha', alpha_tmp - beta * x_m.
mean())
    alpha_mu = pm.Deterministic('alpha_mu', alpha_tmp_mu - beta_mu
* x_m.mean())
    alpha_sd = pm.Deterministic('alpha_sd', alpha_tmp_sd - beta_mu
* x_m.mean())

    mu, sds, elbo = pm.variational.advi(n=100000, verbose=False)
    cov_scal = np.power(hierarchical_model.dict_to_array(sds), 2)
    step = pm.NUTS(scaling=cov_scal, is_cov=True)
    trace_hm = pm.sample(1000, step=step, start=mu)
```

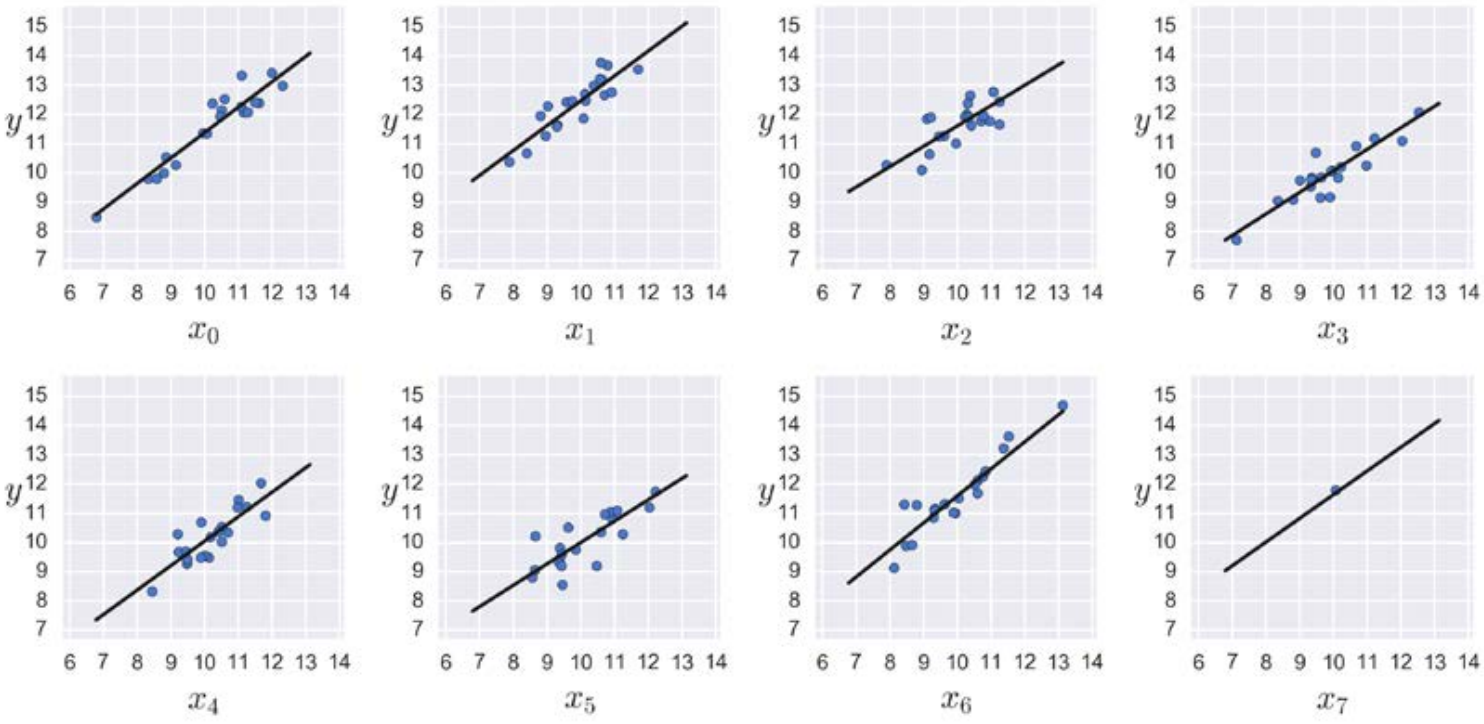
And now the `traceplot` and summary dataframe, even for the little lonely point:

```
varnames=['alpha', 'alpha_mu', 'alpha_sd', 'beta',
  'beta_mu', 'beta_sd', 'epsilon', 'nu']
pm.traceplot(trace_hm, varnames)
```



Let's plot the fitted lines, including the one for the lonely point! Of course that line is mostly informed by the data points in the rest of the groups.

```
j, k = 0, N
x_range = np.linspace(x_m.min(), x_m.max(), 10)
for i in range(M):
    plt.subplot(2,4,i+1)
    plt.scatter(x_m[j:k], y_m[j:k])
    alfa_m = cadena_a['alpha'][:,i].mean()
    beta_m = cadena_a['beta'][:,i].mean()
    plt.plot(x_range, alfa_m + beta_m*x_range, c='k', label='y
      = {:.2f} + {:.2f} * x'.format(alfa_m, beta_m))
    plt.xlim(x_m.min()-1, x_m.max()+1)
    plt.ylim(y_m.min()-1, y_m.max()+1)
    j += N
    k += N
plt.tight_layout()
```



# Correlation, causation, and the messiness of life

Suppose we want to predict how much are we are going to pay for gas to heat our home during winter and suppose we know the amount of sun radiation in the zone we live in. 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 preventing us from inverting the question and asking about the sun radiation, given the bill. If we establish a linear relationship (or any other relation for that matters), 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 us develop this idea a little bit more. It is possible 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 is not true that we can control the radiation the sun emits 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 that shows 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 global warming is highly correlated to the increasing levels of atmospheric $CO_2$. From this observation alone, we cannot conclude if higher temperatures are causing an increase in the levels of $CO_2$ or if the higher levels of the gas are increasing the temperature. Further more, it could happen that there is a third variable that we are not taking into account and which is producing both higher temperatures and higher levels of $CO_2$. However, and pay attentions to this, we can do an experiment where we build a set of glass tanks filled with different quantities of $CO_2$. We can have one with regular air (that contains *~0.04%* of $CO_2$) and the others with air plus increasing amounts of $CO_2$. We then let these tanks receive sunlight for, let's say, three hours. If we do this we will verify that tanks with higher levels of $CO_2$ have higher final temperatures. Hence. we will conclude that indeed $CO_2$ is a green-house effect gas. Using the same experiment, we can also measure the concentration of $CO_2$ at the end of the experiment to check that temperature does not cause the $CO_2$ level to increase, at least not from air. In fact, higher temperature can contribute to higher levels of $CO_2$ because oceans are a reservoir of $CO_2$ and $CO_2$ is less soluble in water when temperatures increase. Long story short, summer is coming and we are not doing enough to solve this self-inflected problem.

Another important aspect we can discuss with this example 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. Let's see, 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 wind pressure. 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 greed), and also how tightly the State regulates the company. We are trying to model all this mess with just a line and two variables!!! So, taking into account the context of our problems is always a necessity and can lead us to better interpretation, less risk of making nonsensical statements, and better predictions; taking into account the context may even give us clues on how to improve our model.

# Polynomial regression

I hope you are excited about the skills you have learned 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:

$$\mu = \beta_0 \boldsymbol{x}^0 + \beta_1 \boldsymbol{x}^1 + \beta_2 \boldsymbol{x}^2 + \beta_3 \boldsymbol{x}^3 \ldots + \beta_n \boldsymbol{x}^n$$

If we pay attention, we can see the simple linear model hidden in this polynomial. To uncover it, all we need to do is to make all the $\beta_n$ coefficients labeled higher than 1 exactly zero. And then we will get:

$$\mu = \beta_0 \boldsymbol{x}^0 + \beta_1 \boldsymbol{x}^1$$

Polynomial regression is still linear regression, the linearity in the model is related to how the parameters enter in to the model, not the variables.

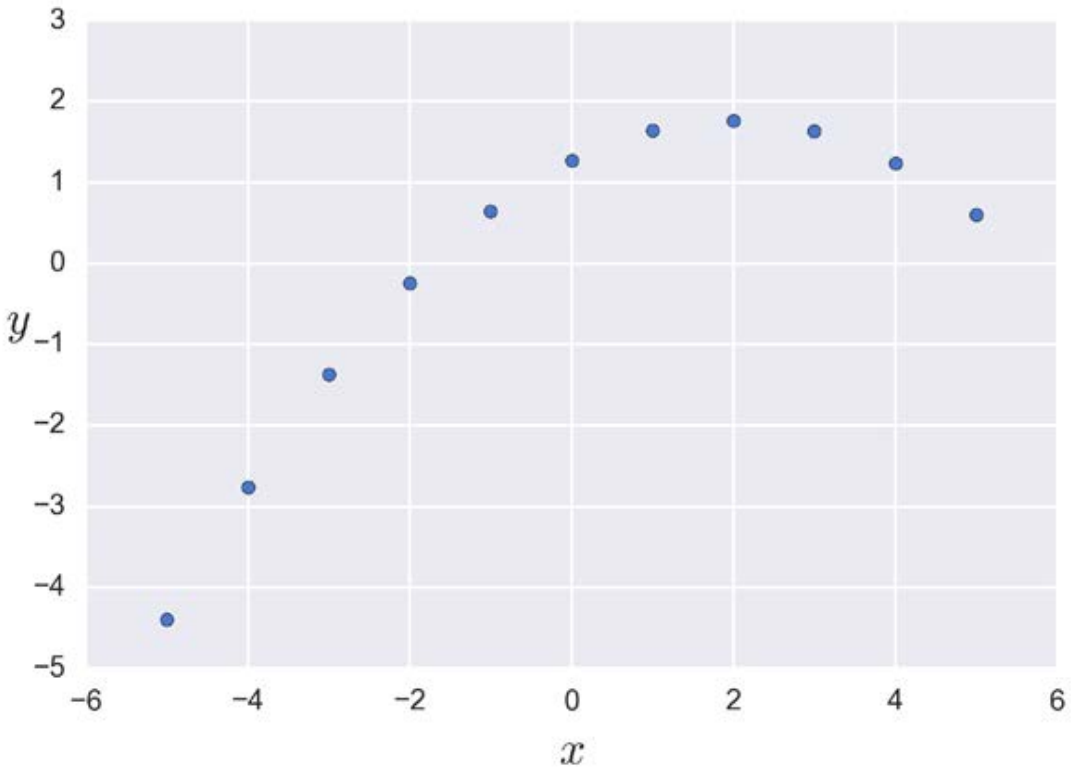
Let's try building a polynomial regression starting from the simpler polynomial model (after a constant and line), a parabola.

$$\mu = \beta_0 \boldsymbol{x}^0 + \beta_1 \boldsymbol{x}^1 + \beta_2 \boldsymbol{x}^2$$

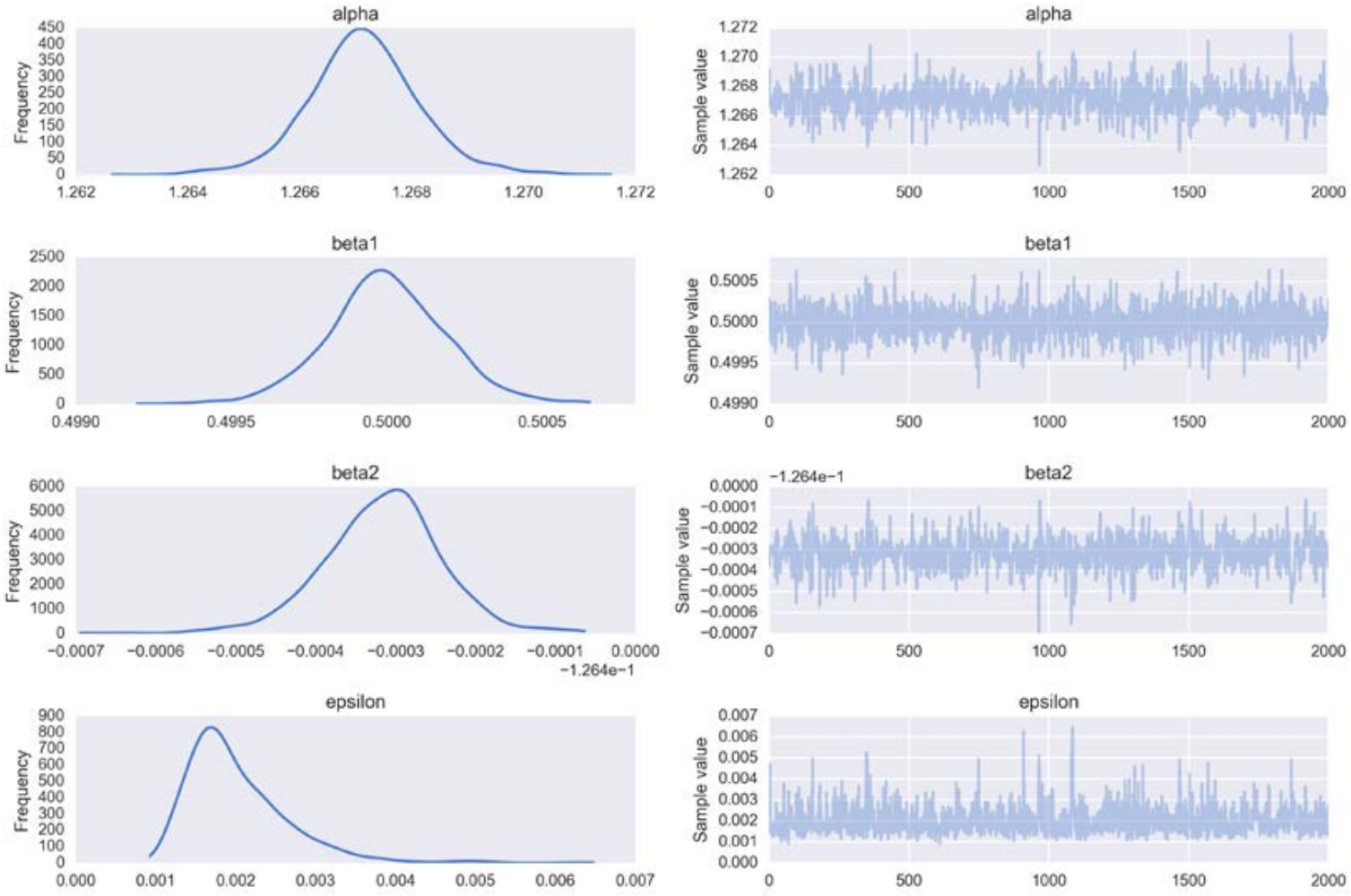
The third term controls the curvature of the relationship.

As a dataset, we are going to use the second group of the `anscombe` quartet. We are going to upload this dataset from `seaborn` and we are going to plot it.

```
ans = sns.load_dataset('anscombe')
x_2 = ans[ans.dataset == 'II']['x'].values
y_2 = ans[ans.dataset == 'II']['y'].values
x_2 = x_2 - x_2.mean()
y_2 = y_2 - y_2.mean()
plt.scatter(x_2, y_2)
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$y$', fontsize=16, rotation=0)
```

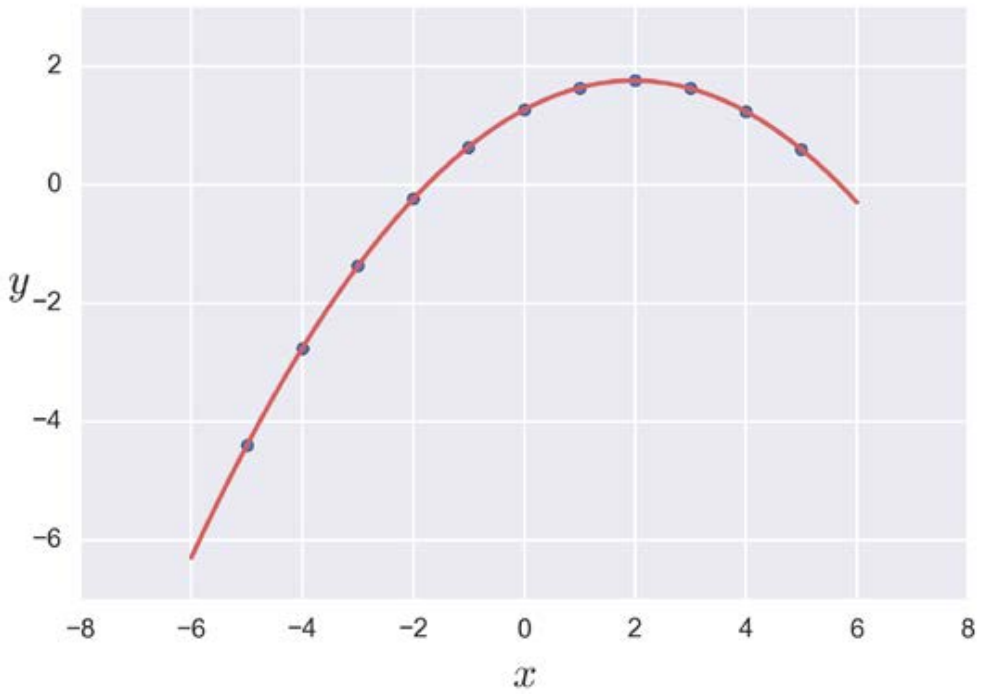



```
with pm.Model() as model_poly:
    alpha = pm.Normal('alpha', mu=0, sd=10)
    beta1 = pm.Normal('beta1', mu=0, sd=1)
    beta2 = pm.Normal('beta2', mu=0, sd=1)
    epsilon = pm.Uniform('epsilon', lower=0, upper=10)

    mu = alpha + beta1 * x_2 + beta2 * x_2**2

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

    start = pm.find_MAP()
    step = pm.NUTS(scaling=start)
    trace_poly = pm.sample(3000, step=step, start=start)
pm.traceplot(trace_poly)
```




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

```
x_p = np.linspace(-6, 6)
y_p = trace_poly['alpha'].mean() + trace_poly['beta1'].mean()
  * x_p + trace_poly['beta2'].mean() * x_p**2
plt.scatter(x_2, y_2)
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$y$', fontsize=16, rotation=0)
plt.plot(x_p, y_p, c='r')
```




# 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 $\boldsymbol{y}$ changes per unit change of $\boldsymbol{x}$, we cannot just check the value of $\beta_1$, because $\beta_2$, and higher coefficients if present, had an effect on such a quantity. So the $\beta$ coefficients are no longer slopes, they are something else. In the previous example, $\beta_1$ is positive and hence the curve begins with a positive slope, but $\beta_2$ is negative and hence after a while the line begins to curve downwards. 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 $\boldsymbol{x}$. When $x_i < \sim 11$, the dominating contribution comes from $\beta_1$, and when $x_i > \sim 11$ $\beta_2$ dominates.

The problem of interpreting the parameters is not just a mathematical problem, because this can be solved by careful inspection and understanding of the model. The problem is that in many cases the parameters are not translated to a meaningful quantity in our domain knowledge. We cannot relate them with the metabolic rate of a cell or the energy emitted by a star or the number of bedrooms in a house. They are just parameters without a clear physical meaning. Such a model could be useful for predicting but is not very useful for understanding the underlying process that generates the data. Also in practice, polynomials of order higher than two or three are not generally very useful models and other models are preferred, such as the ones we will discuss in *Chapter 8*, *Gaussian Processes*.

# Polynomial regression – the ultimate model?

As we saw, we can think of the line as a sub-model of the parabola when $\beta_2$ is equal to zero, and a line is also a sub-model of a cubic model when $\beta_2$ and $\beta_3$ are equal to zero. And of course, the parabola is a sub-model of the cubic one when $\beta_3$… OK, I will stop it here, but I think you already notice the pattern. This suggests an algorithm for using the linear regression model to fit an arbitrary complex model. We build a polynomial of infinite order and somehow we make most of the coefficients zero until we get a perfect fit of our data. To test this idea, we can start with a simple example. Let's use the quadratic model we just built to fit the third set of the `Anscombe` dataset. I will wait here while you do that. I am still waiting, do not 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 the previous simple experiment validates the idea of building an infinite order polynomial to 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! Why is this a problem? Well that's the subject of *Chapter 6*, *Model Comparison*, but spoiler alert! A model that fits your current data perfectly will in general be a poor description of unobserved data. The reason 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 phenomenon in statistics and machine learning. Polynomial regression makes for a convenient straw man when it comes to overfitting because it is easy to see the problem and generate intuition. Nevertheless, with more complex models it is often easy to overfit, even without realizing. Part of the job, when analyzing data, is to be sure that the model is not overfitting. We will discuss this topic in detail in *Chapter 6*, *Model Comparison*.

Besides the overfitting problem, we generally prefer models we can understand. The parameters of a line are generally easier to interpret in a physically meaningful way than the parameters of a cubic model, even if the cubic model explains the data slightly better.

# Multiple linear regression

In all previous examples we have been working with one dependent variable and one independent variable, but in many cases we will find that we have many independent variables we want to include in our model. Some examples could be:

- Perceived quality of wine (dependent) and acidity, density, alcohol level, residual sugar, and sulphate content (independent variables)
- Student average grades (dependent) and family income, distance home-school, and mother education (independent variables)

In such a cases, we will have the mean of the dependent variable modeled as:

$$\mu = \alpha + \beta_1 \boldsymbol{x}_1 + \beta_2 \boldsymbol{x}_2 + \beta_3 \boldsymbol{x}_3 \ldots + \beta_m \boldsymbol{x}_m$$

Notice that this is not exactly the same as the polynomial regression we saw before. Now we have different variables instead of successive powers of the same variable.

Using linear algebra notation we can write a shorter version:

$$\mu = \alpha + \boldsymbol{\beta X}$$

Where $\beta$ is a vector of coefficients of length $m$, that is, the number of dependent variables. The variable $\boldsymbol{X}$ 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. But basically what you need to know is we are just using a shorter and more convenient way to write our model:

$$\boldsymbol{\beta X} = \sum \beta_i \boldsymbol{x}_i = \beta_1 \boldsymbol{x}_1 + \cdots + \beta_m \boldsymbol{x}_m$$

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, an hyperplane of dimension $m$. Thus, the model for the multiple linear regression is essentially the same as we use for simple linear regression, the only difference being that now $\boldsymbol{\beta}$ is a vector and $\boldsymbol{X}$ is a matrix.

Let us define our data:

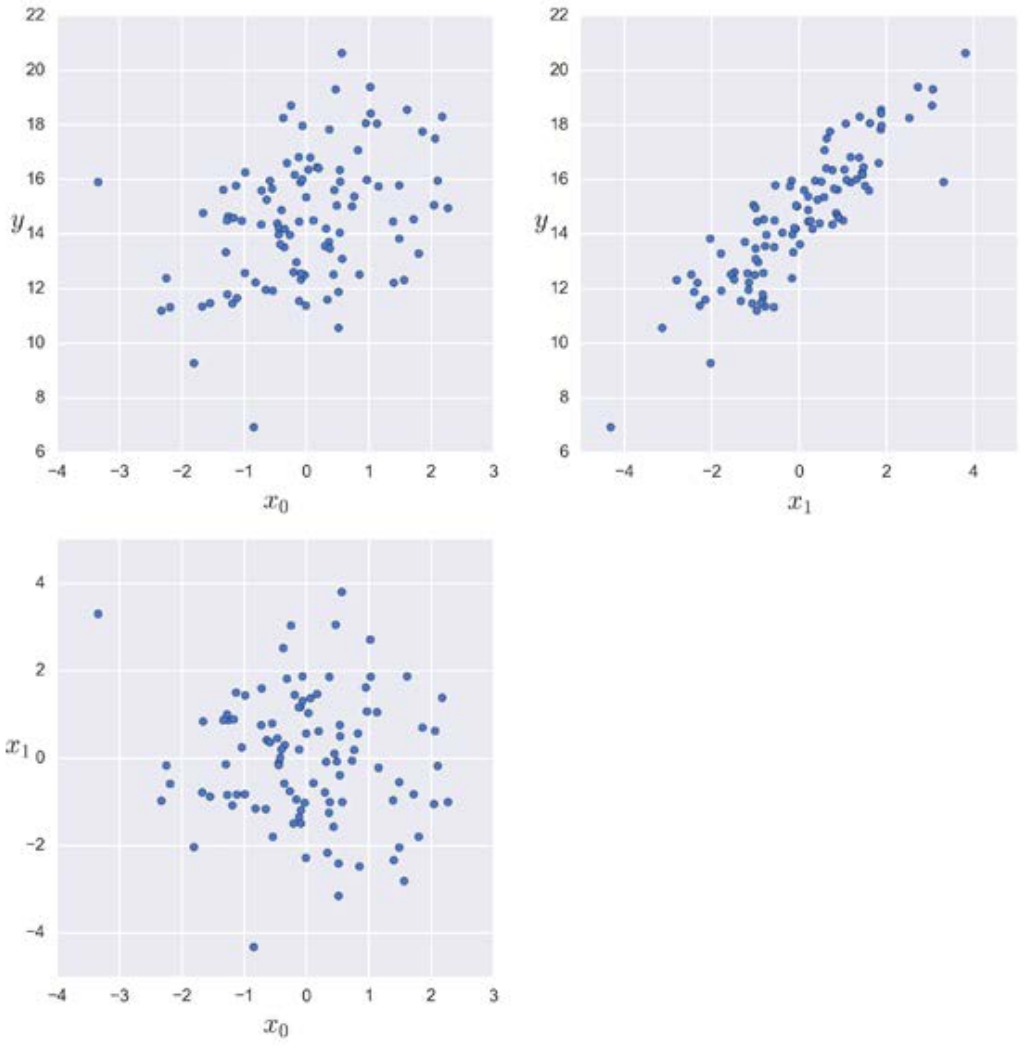
```
np.random.seed(314)
N = 100
alpha_real = 2.5
beta_real = [0.9, 1.5]
eps_real = np.random.normal(0, 0.5, size=N)

X = np.array([np.random.normal(i, j, N) for i,j in zip([10, 2],
  [1, 1.5])])
X_mean = X.mean(axis=1, keepdims=True)
X_centered = X - X_mean
y = alpha_real + np.dot(beta_real, X) + 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. 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):
        plt.subplot(2, 2, idx+1)
        plt.scatter(x_i, y)
        plt.xlabel('$x_{}$'.format(idx), fontsize=16)
        plt.ylabel('$y$', rotation=0, fontsize=16)

    plt.subplot(2, 2, idx+2)
    plt.scatter(x[0], x[1])
    plt.xlabel('$x_{}$'.format(idx-1), fontsize=16)
    plt.ylabel('$x_{}$'.format(idx), rotation=0, fontsize=16)
```

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

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

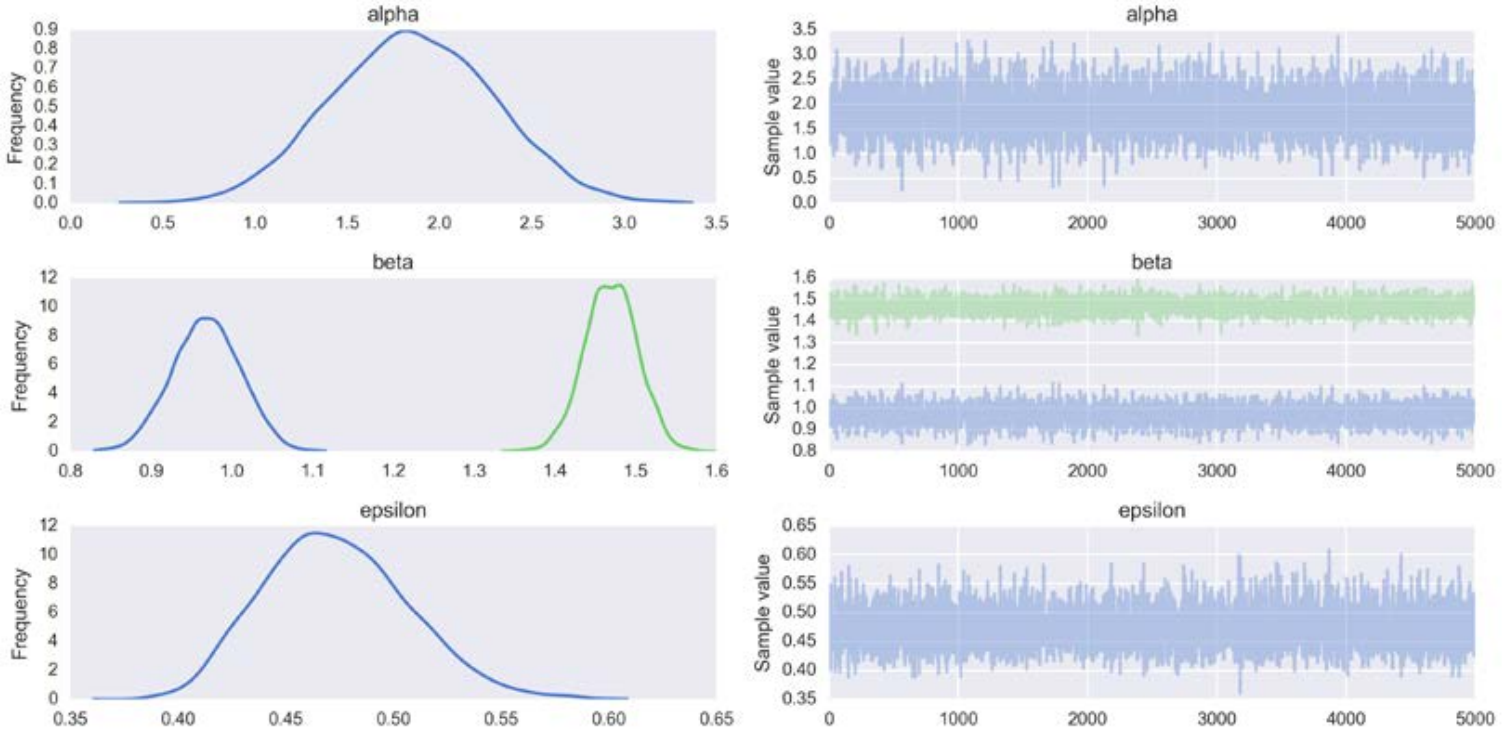



Now let's use PyMC3 to define a model suitable for multiple linear regression. As expected, the code looks pretty similar to the one used for simple linear regression. The main differences are:

- The variable `beta` is Gaussian with shape two, one slope per each independent variable.
- We define the variable `mu` using the function `pm.math.dot()`, that is, the dot product or matrix product we mention early from linear algebra.

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 also 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 `beta` is a Theano tensor and not a NumPy array.

```
with pm.Model() as model_mlr:
    alpha_tmp = pm.Normal('alpha_tmp', mu=0, sd=10)
    beta = pm.Normal('beta', mu=0, sd=1, shape=2)
    epsilon = pm.HalfCauchy('epsilon', 5)

    mu = alpha_tmp + pm.math.dot(beta, X_centered)

    alpha = pm.Deterministic('alpha', alpha_tmp -
      pm.math.dot(beta, X_mean))

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

    start = pm.find_MAP()
    step = pm.NUTS(scaling=start)
    trace_mlr = pm.sample(5000, step=step, start=start)
varnames = ['alpha', 'beta', 'epsilon']
pm.traceplot(trace_mlr, varnames)
```




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

```
pm.df_summary(trace_mlr, varnames)
```

| | mean | sd | mc_error | hpd_2.5 | hpd_97.5 |
|---|---|---|---|---|---|
| alpha_0 | 2.07 | 0.50 | 5.96e-03 | 1.10 | 3.09 |
| beta_0 | 0.95 | 0.05 | 6.04e-04 | 0.85 | 1.05 |
| beta_1 | 1.48 | 0.03 | 4.30e-04 | 1.43 | 1.54 |
| epsilon | 0.47 | 0.03 | 4.63e-04 | 0.41 | 0.54 |

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 a multiple linear regression, each parameter only makes sense in the context of the other parameters.

# Confounding variables and redundant variables

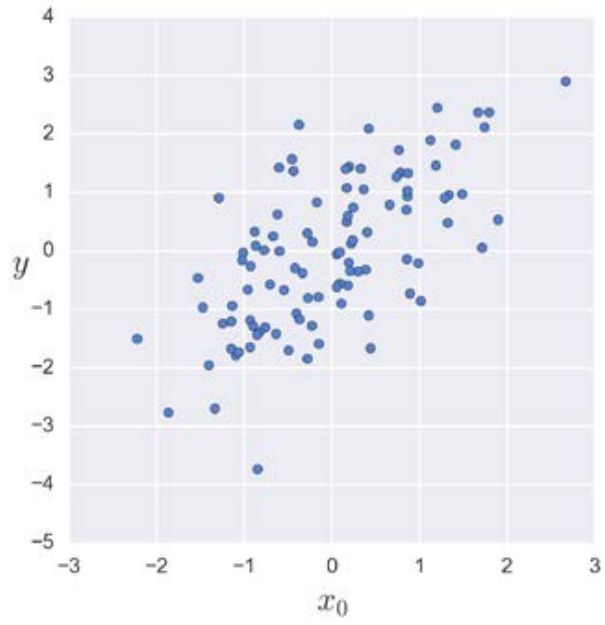
Imagine the following situation. We have a variable $z$ correlated with the predictor variable $x$ and, at the same time, with the predicted variable $y$. Suppose that the variable $z$ is the one responsible for causing $x$ and $y$. For example, *z* could be the industrial revolution (a really complex variable!), $x$ the number of pirates, and $y$ the concentration of $CO_2$; this example should be 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 and we will be missing the mechanism. Remember, we already discussed that correlation does not imply causation. One reason this relationship 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. The problem is that 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 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 things and could also be problematic when we predict something (and we do not care about understanding the underlying mechanism). Understanding the mechanism helps us translate what we have learned to new situations; blind predictions do not always transfer well. 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.

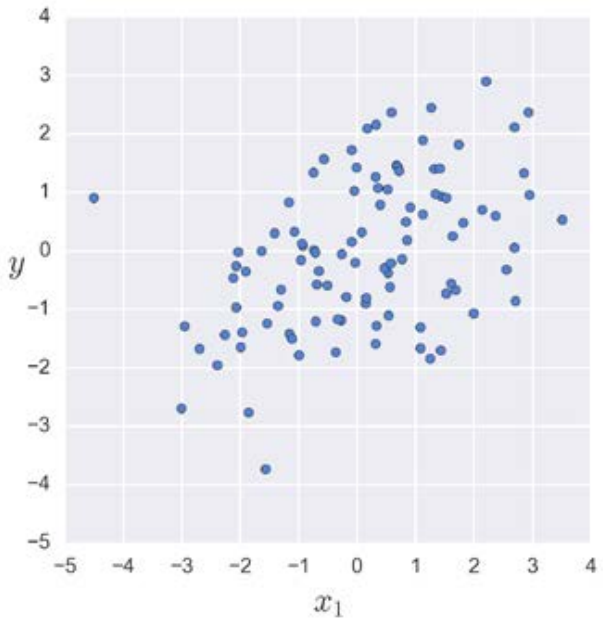
We are going to use synthetic data to explore the confounding variable issue. In the following code we are simulating a confounding variable as `x_1`. Notice how this variable has influences on `x_2` and `y`.

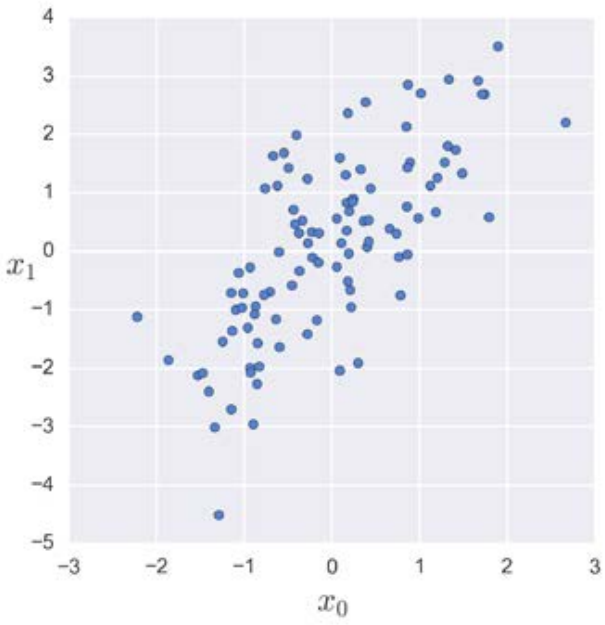
```
N = 100
x_1 = np.random.normal(size=N)
x_2 = x_1 + np.random.normal(size=N, scale=1)
y = x_1 + np.random.normal(size=N)
X = np.vstack((x_1, x_2))
```

Notice that by virtue of the way we create these variables, they are already centered, as we can easily check with the convenient function we wrote early. Therefore, we do not need to center the data to speed-up the inference process. In fact for this example the data is even standardized.

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

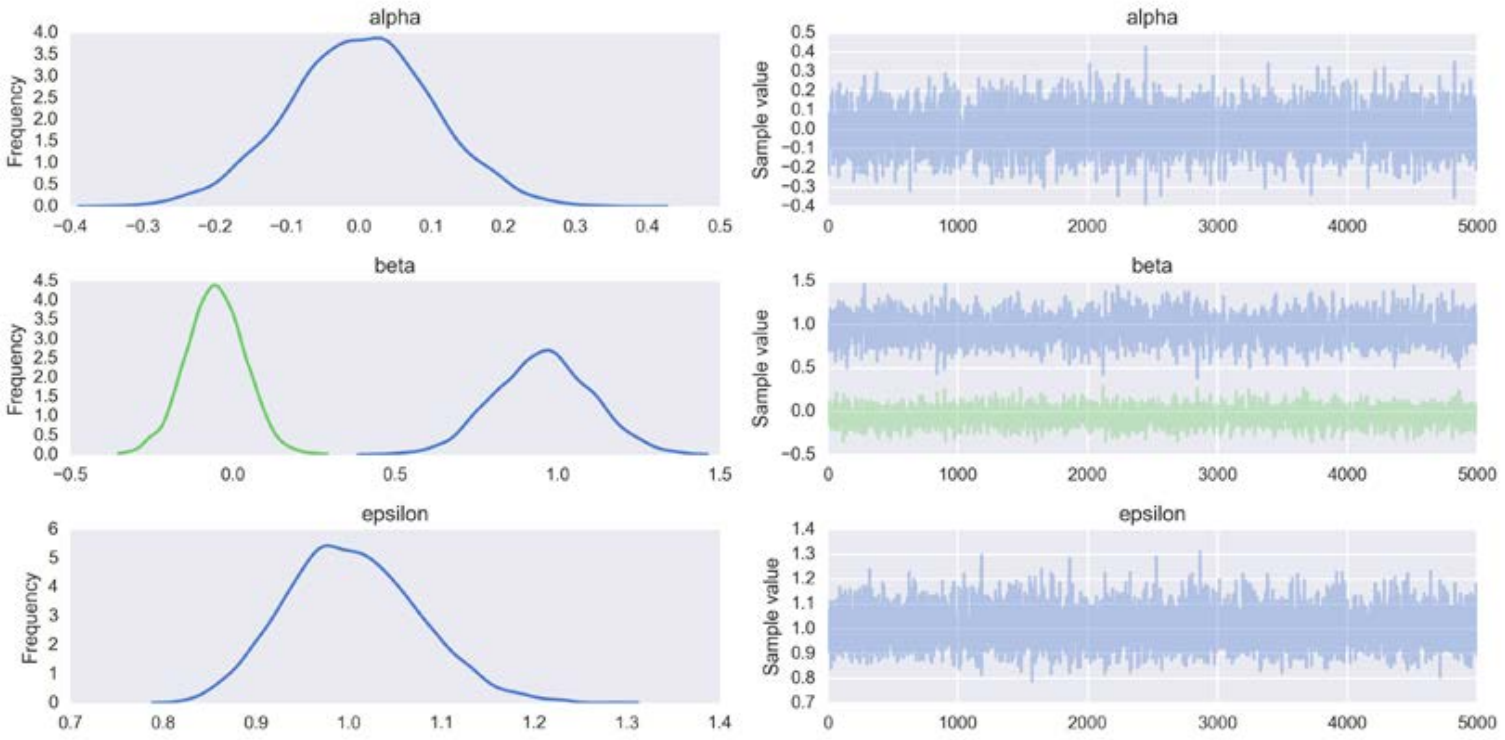


We will now use PyMC3 to create the model and sample from it. At this point, this model should look pretty familiar to you.

```
with pm.Model() as model_red:
    alpha = pm.Normal('alpha', mu=0, sd=10)
    beta = pm.Normal('beta', mu=0, sd=10, shape=2)
    epsilon = pm.HalfCauchy('epsilon', 5)

    mu = alpha + pm.math.dot(beta, X)

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

    start = pm.find_MAP()
    step = pm.NUTS(scaling=start)
    trace_red = pm.sample(5000, step=step, start=start)
```




Now let's print the summary for the results as a nice pandas dataframe. We will focus on the mean value of the beta parameters.

```
pm.df_summary(trace_red)
```

| | mean | sd | mc_error | hpd_2.5 | hpd_97.5 |
|---|---|---|---|---|---|
| alpha | 0.01 | 0.10 | 1.59e-03 | -0.20 | 0.19 |
| beta_0 | 0.96 | 0.16 | 3.13e-03 | 0.67 | 1.29 |
| beta_1 | -0.05 | 0.10 | 2.06e-03 | -0.24 | 0.14 |
| epsilon | 1.00e+00 | 0.07 | 1.15e-03 | 0.87 | 1.15 |

As we can see, `beta_1` is close to 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 real important variable is `x_1`. Anyway, the most interesting part of this example is yet to come.

Now I need you to do a couple of tests while I rest a little bit. I need you to rerun the model twice, once including just `x_1` (and not `x_2`) and then including `x_2` (and not `x_1`). If you check the accompanying code you will see I have included a couple of commented lines of code to make the thing easier for you. The question is, how different are the means for the beta coefficients for the three cases?

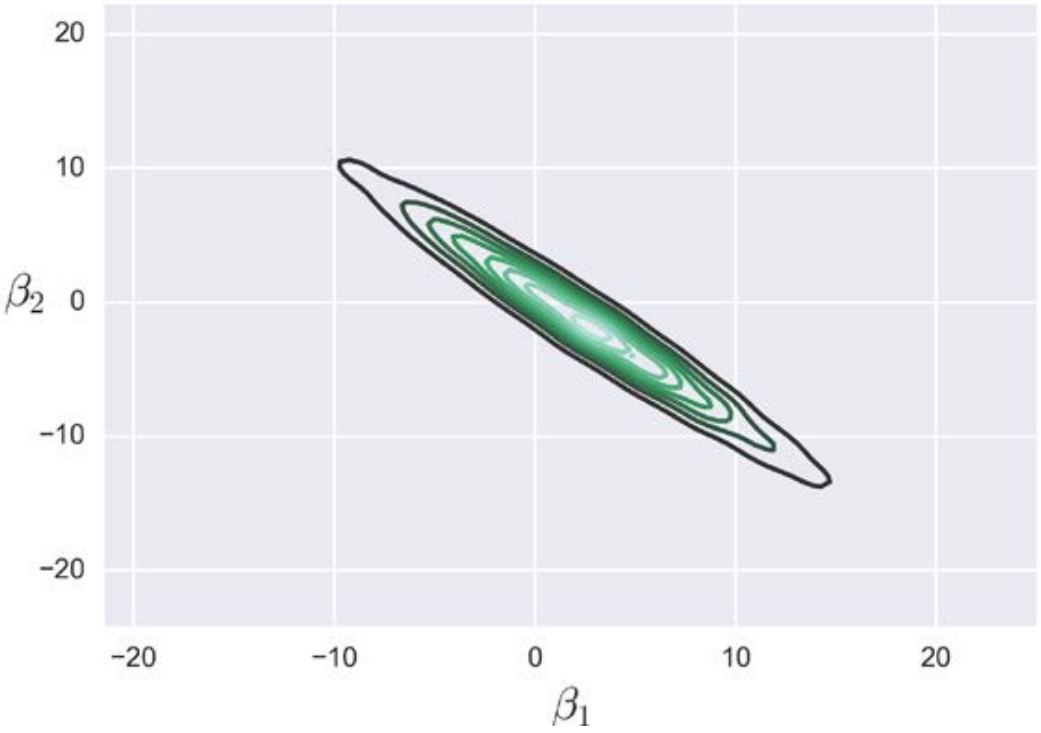
If you did the experiment, you may have noticed that, besides the exact values you get for the beta coefficients, $\beta_2$ is lower for the multiple linear regression than for the simple linear regression. In other words, the ability of `x_2` to explain `y` is lower (maybe even null) when `x_1` is included in the model.

# Multicollinearity or when the correlation is too high

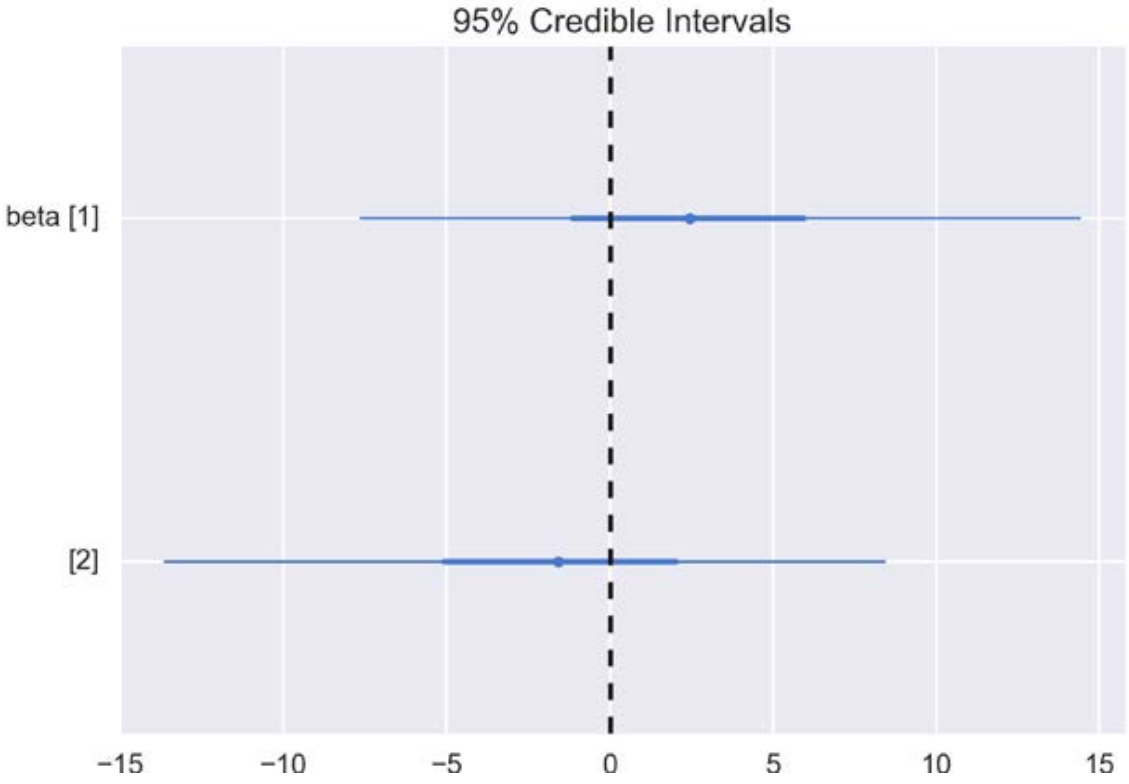
In the previous example, we saw how a multiple linear regression model reacts to redundant variables and we saw the importance of considering possible confounding variables. Now we will take the previous example to an extreme and see what happens when two variables are highly correlated. To study this problem and its consequences for our 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 decreasing the scale of the random noise we add to `x_1` to obtain `x_2`, that is:

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

This change in the data-generating code is practically equivalent to sum 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; you should see that now the scatter plot for `x_1` and `x_2` is virtually a straight line with a slope around *1*. Run the model and check the results. In the code that comes with the book, you will find a few lines to plot a 2D KDE plot for the beta coefficients. You should see something similar to the following figure:




The HPD for beta coefficients is really wide; in fact they are more or less as wide as the priors allow.



As we can see, the posterior for `beta` is a really narrow diagonal. When one beta 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 $\mu$ is:

$$\mu = \alpha + \beta_1 \boldsymbol{x}_1 + \beta_1 \boldsymbol{x}_2$$

If we assume $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ are not just practically equivalent, but identical, we can re-write the model as:

$$\mu = \alpha + \left(\beta_1 + \beta_2\right) \boldsymbol{x}$$

It turns out that it is the sum of $\beta_1$ and $\beta_2$, and not their separated values, that effects $\mu$ and then the model is indeterminate (or equivalently the data is unable to affects it). In our example, there are two reason why beta is not free to move between $[-\infty, \infty]$. First, both variables are almost the same, but they are not exactly equal; second, and most important, we have a prior restricting the plausible values that the beta coefficients 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 a wide distribution for beta; it is what it is. 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. Once again the model is capturing the data very well. Third, this may be not a very good model to understand our problem. It may be smarter just to remove one of the variables from the model. We will end up having a model that predicts 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 strongly should two or more variables be correlated to become a problem? Well *0.9845*. No, just kidding. There is no such magic number. It is 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; 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, diagnostic tools, such as checking the autocorrelation and 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.

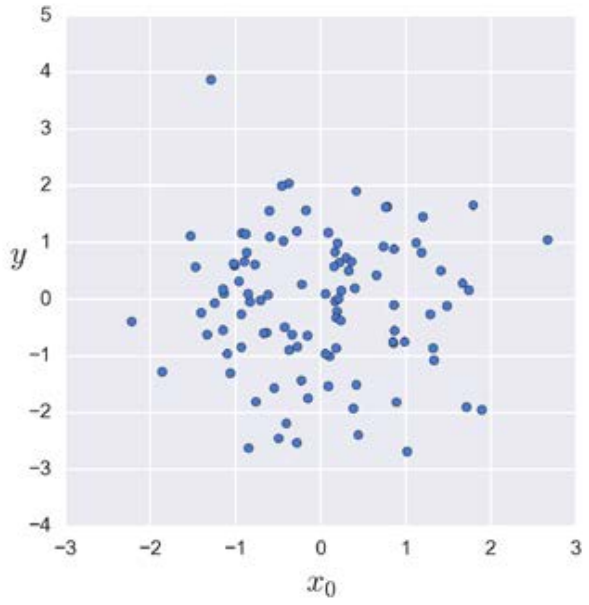
What should we do if we find highly correlated variables?

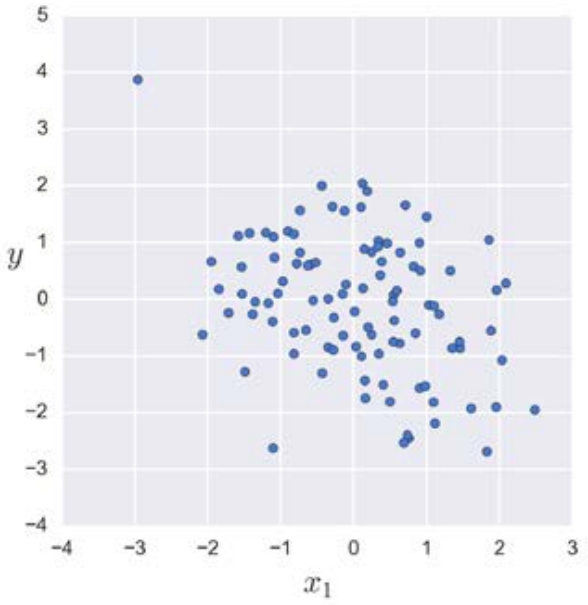
- If the correlation is really high we can eliminate one of the variables from the analysis; given that both variables have similar information, which one we eliminate is often irrelevant. We can eliminate variables based on pure convenience, such as removing the least known variable in our discipline 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 a principal component analysis. 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 in order 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**.

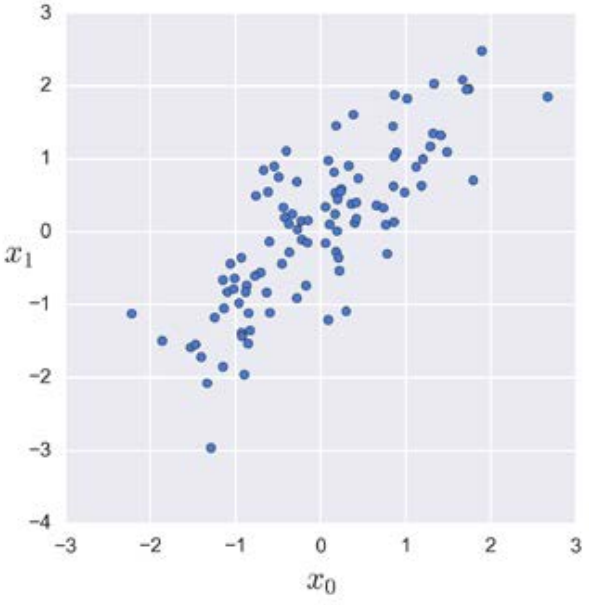
# Masking effect variables

Another similar situation to the one we saw previously occurs when one of the predicted variables is positively correlated while the other is negatively correlated with the predicted variable. Let us create toy data for such a case:

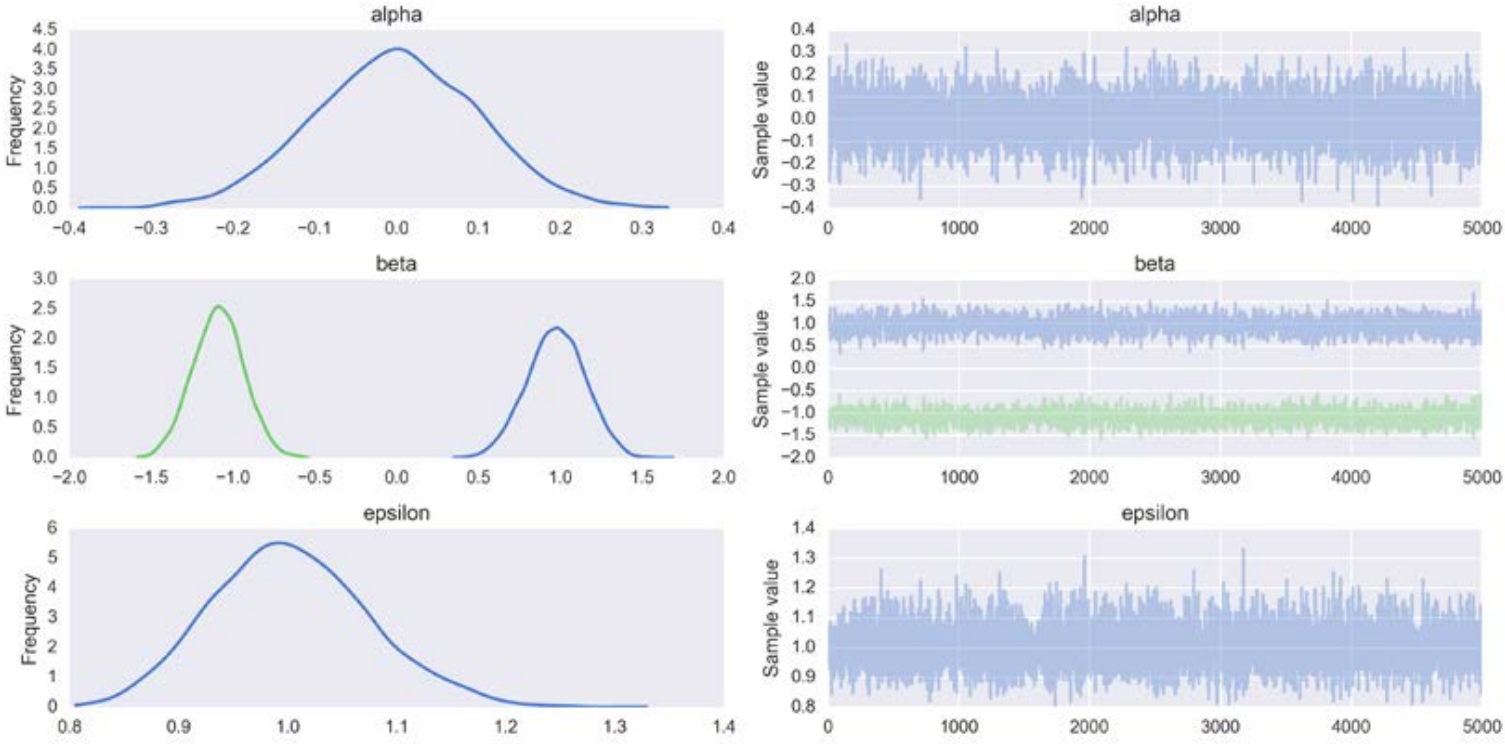
```
N = 100
r = 0.8
x_0 = np.random.normal(size=N)
x_1 = np.random.normal(loc=x_0 * r, scale=(1 - r ** 2) ** 0.5)
y = np.random.normal(loc=x_0 - x_1)
X = np.vstack((x_0, x_1))
scatter_plot(X, y)
```



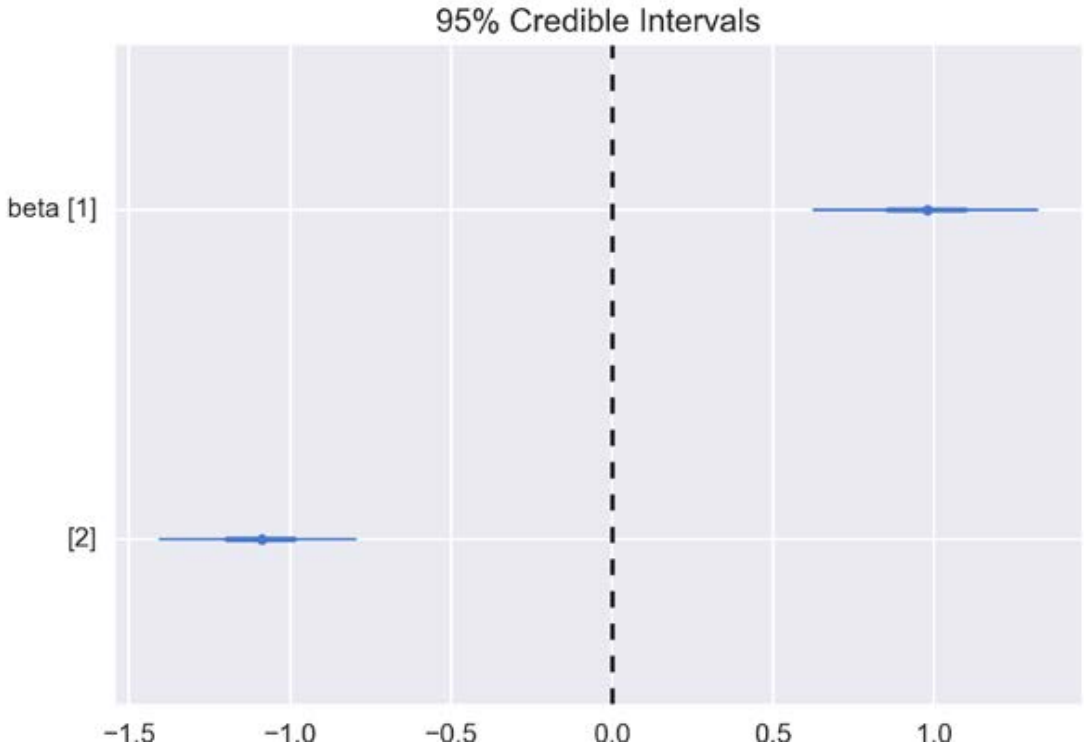



```
with pm.Model() as model_ma:
    alpha = pm.Normal('alpha', mu=0, sd=10)
    beta = pm.Normal('beta', mu=0, sd=10, shape=2)
    epsilon = pm.HalfCauchy('epsilon', 5)

    mu = alpha + pm.math.dot(beta, X)

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

    start = pm.find_MAP()
    step = pm.NUTS(scaling=start)
    trace_ma = pm.sample(5000, step=step, start=start)
pm.traceplot(trace_ma)
```



```
pm.forestplot(trace_ma, varnames=['beta'])
```




According to the posterior the values of beta are close to **1** and **-1**. That is, `x_1` is positively correlated with `y` and `x_2` is negatively correlated. Now we are going to repeat the analysis, but this time (probably you already guess) we are going to do so for each separated variable.

For each separated variable, we saw that $\beta$ is close to zero. In isolation, each `x` variable is not good for predicting `y`. Instead, when we combine them they can be used to predict `y`! When `x_1` increases `x_2` also increases and when `x_2` increases `y` decreases. Thus, if we just look at `x_1` omitting `x_2` we will declare that `y` barely increases when `x_1` increases and that `y` barely decreases when `x_2` increases. Each dependent variable has an opposite effect on the dependent variable and the dependent variables are correlated, thus omitting one of them from the analysis will result in an underestimation of the real effects.

# Adding interactions

So far in the definition of the multiple regression model it is declared (implicitly) that a change in $x_1$ 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 $x_2$ affect $y$ modulated by changes in $x_1$. 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 low doses of *B*) while the effect of *A* is negative (even lethal) for increasing doses of *B*.

In all the examples we have seen so far, the dependent 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 and one option is to multiply variables, for example:

$$\mu = \alpha + \beta_1 \boldsymbol{x}_1 + \beta_2 \boldsymbol{x}_2 + \beta_3 \boldsymbol{x}_1 \boldsymbol{x}_2$$

Notice that the $\beta_3$ coefficient is multiplying a third variable that is the result of multiplying $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. This non-additive term is an example of what is know in statistics as **interactions**, because they model the *interactions* between variables (drugs in our example). There are different functional forms for modeling interactions; multiplication is a pretty common one.

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.

# The GLM module

Linear models are widely used in statistics and machine learning. For that reason, PyMC3 includes a module named `glm`, which stand for generalized linear model, the name will become clear in the next chapter. The `glm` module simplifies writing linear models. For example, a simple linear regression will be:

```
with Model() as model:
    glm.glm('y ~ x', data)
    trace = sample(2000)
```

The second line of the preceding code takes care of adding default flat priors for the intercept and for the slope and a Gaussian likelihood. These are OK if you just want to run a default linear regression. Note that the MAP of this model will be essentially equivalent to the one obtained using the (frequentist) ordinary least square method. If you need to, you can also use the `glm` module and change priors and likelihoods. If you are not familiar with R's syntax, `'y ~ x'` specifies that we have an output variable $\boldsymbol{y}$ that we want to estimate as a linear function of $\boldsymbol{x}$. The `glm` module also includes a function to make posterior predictive plots.

# Summary

We learned that linear regression is one of the most widely used models in statistics and machine learning and it is also the building block of several more complex methods. This is a widely used model and different people tend to give different names to the same concept or object. Thus, we first introduced some commonly used vocabulary in statistics and machine learning. We studied the core of the linear model, an expression to connect an input variable to an output variable. In this chapter, we performed that connection using Gaussian and Student's t-distributions and in future chapters we will extend this model to other distributions. We dealt with computational problems and how to fix them by centering and/or standardizing the data and we had the opportunity to clearly see the advantages of using NUTS over Metropolis sampler. We adapted the hierarchical model introduced in the past chapter to simple linear regression. We also explored polynomial regression to fit curved lines and we discussed some of the problems with these models; we anticipate the main topic of *Chapter 6*, *Model Comparison*. We also discussed how to perform linear regression with more than one input variable and took some time to discuss precautions that we should take when interpreting linear models. In the next chapter, we will see how to extend the linear regression model to classify data.

# Keep reading

- *Statistical Rethinking* by *Richard McElreath*. Chapter 4 and 5
- *Doing Bayesian Data Analysis, Second Edition* by *John Kruschke*. Chapter 17 and 18
- *An Introduction to Statistical Learning*. *Gareth James* and others (second edition). Chapter 4
- *Bayesian Data Analysis, Third Edition* by *Andrew Gelman* and others. Chapter 14-17
- *Machine Learning: A probabilistic Probabilistic Perspective* by *Kevin P. Murphy*. Chapter 7
- *Data Analysis Using Regression and Multilevel/Hierarchical Models* by *Andrew Gelman* and *Jeniffer Hill*

# Exercises

1. Choose a dataset that you find interesting and use it with the simple linear regression model. Re-run the plots and also compute the Pearson correlation coefficient with the different methods. If you do not have one, try searching online, for example at `http://data.worldbank.org/` or `http://www.stat.ufl.edu/~winner/datasets.html`.
2. Read and run the following example from PyMC3's documentation `https://pymc-devs.github.io/pymc3/notebooks/LKJ.html`.
3. For the unpooled model change the value of the `sd` of the beta prior; try values of *1* and *100*. Explore how the estimated slopes change for each group. Which group is the more affected by this change?
4. See in the accompanying code the `model_t2` (and the data associated with it). Experiment with priors for `nu`. Like the non-shifted exponential and gamma priors (they are commented in the code). Plot the prior distributions to be sure do you understand them; an easy way to do this is to just comment the likelihood in the model and check the `traceplot`.
5. Read and run the following example `http://pymc-devs.github.io/pymc3/notebooks/NUTS_scaling_using_ADVI.html`.
6. Reduce the number of iterations of ADVI (currently at *100000*), for example to *10000*. What is the effect on the number of iterations per second of NUTS? Check also the `traceplot` to see the effect on the sampled values and plot the values of elbo. Replace ADVI in other models where we previously used `find_MAP()`. Did you always observe a benefit?
7. Run the `model_mlr` example, but without centering the data. Compare the uncertainty in the alpha parameter for one case and the other. Can you explain these results? Tip: remember the definition of the alpha parameter (the intercept).
8. 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`
9. Remember to actually run the exercises proposed for the multiple linear regression models.

# 5

# Classifying Outcomes with Logistic Regression

In the last chapter, we learned the core of the linear regression model; in such a model we assume the predicted variable is quantitative (or metric). In this chapter, we will learn how to deal with qualitative (or categorical) variables, such as colors, gender, biological species, political party/affiliation, just to name a few examples. Notice that some variables can be codified as quantitative or as qualitative; for example, we can talk about the categorical variables red and green if we are talking about color names or the quantitative 650 nm and 510 nm if we are talking about wavelengths. One of the problems when dealing with categorical variables is assigning a class to a given observation; this problem is known as classification and is a supervised problem since we have a sample of already classified instances and the task is basically about predicting the correct class for new instances and/or learning about the parameters of the model that describe the mapping between classes and features.

In the present chapter, we will explore:

- Logistic regression and inverse link functions
- Simple logistic regression
- Multiple logistic regression
- The softmax function and the multinomial logistic regression

# Logistic regression

My mother prepares a delicious dish called **sopa seca**, which is basically a spaghetti-based recipe and which translates literally from Spanish as dry soup. While it may sound like a misnomer or even an oxymoron, the name of the dish makes more sense when we learn how it is cooked. Something similar happens with the logistic regression, a model that despite its name is used to solve classification problems rather than regression ones. The logistic regression model is an extension of the linear regression models we saw in the previous chapter, and thus its name. To understand how we can use a regression model to classify, let us began by rewriting the core of the linear model but this time including a small twist as follows:

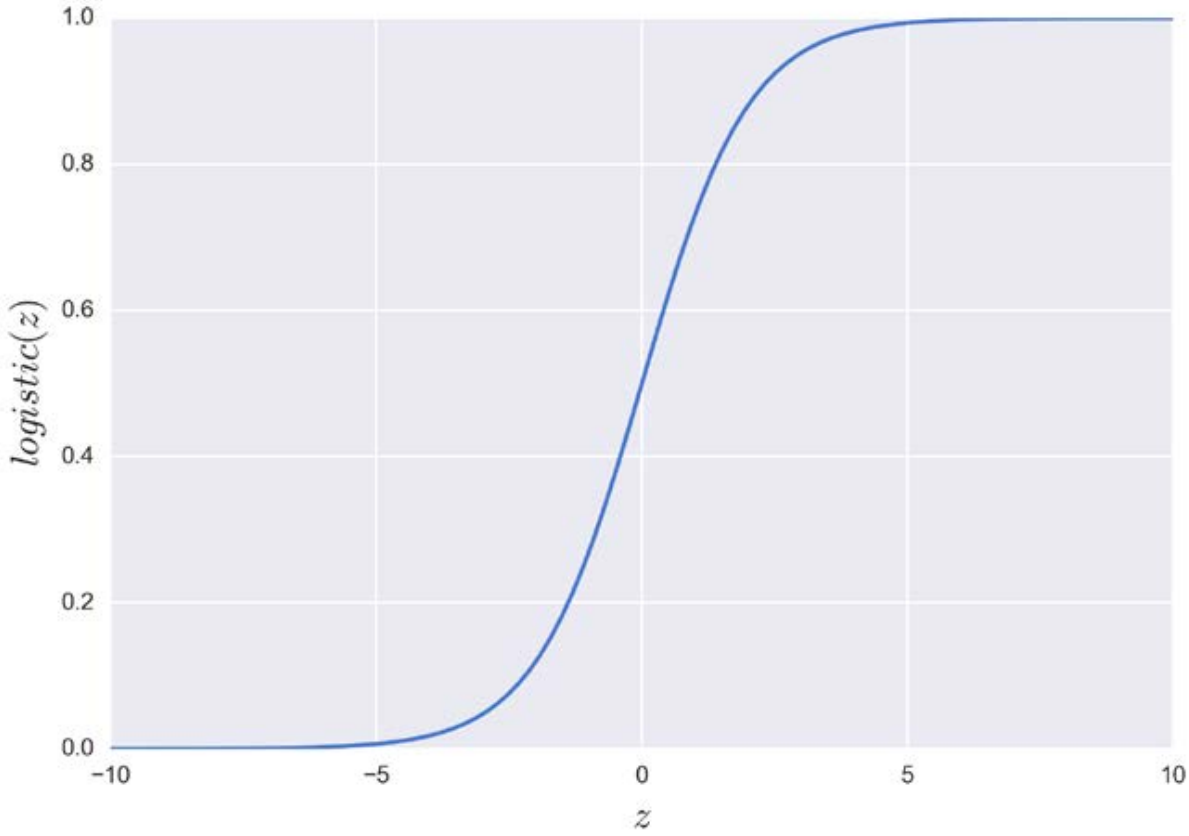
$$\mu = f\left(\alpha + \beta \boldsymbol{X}\right)$$

Where, *f* is some function known to us as the inverse link function. Why do we call *f* the inverse link function instead of just the link function? The reason is that traditionally people thought about these kinds of functions as functions linking the output variable to the linear model, but as you will see, when building Bayesian models it may be easier to think the link goes the other way around, from the linear model to the output variable. Thus, to avoid confusion we are going to talk about the inverse link function. All the linear models from the previous chapter also included an inverse link function, but we omitted writing it since it was the identity function. That is, a function that returns the same value used as its argument. 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. In principle, many functions can work as an inverse link function, but given the name of the chapter, we are going to focus on the logistic function, which we write as:

$$logistic\left(z\right) = \frac{1}{1 + exp\left(-z\right)}$$

The key property of the logistic function, from the classification perspective, is that irrespective of the values of its argument, *z*, the logistic function always returns a value between *0* and *1*. So this function *compresses* the whole real line into the interval *[0, 1]*. This 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(-10, 10, 100)
logistic = 1 / (1 + np.exp(-z))
plt.plot(z, logistic)
plt.xlabel('$z$', fontsize=18)
plt.ylabel('$logistic(z)$', fontsize=18)
```



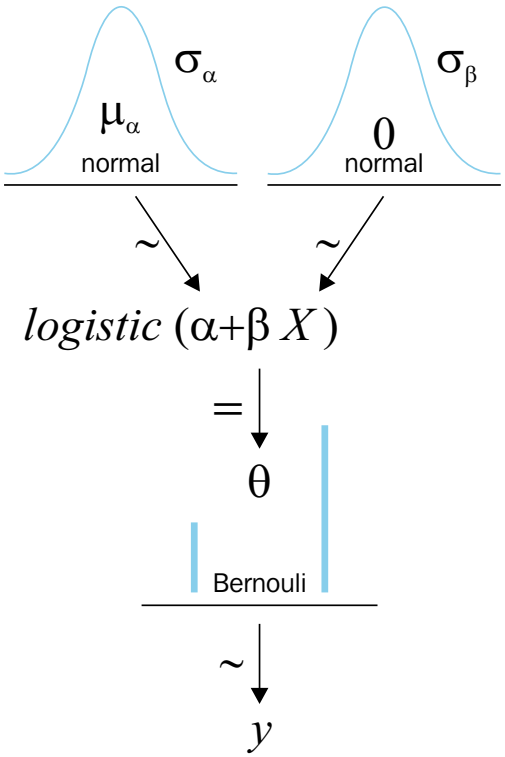

# The logistic model

Now that we know how the logistic function looks, we will continue by learning how it can help us to classify outcomes. Let us begin with a simple case when we have only two classes or instances, for example, ham-spam, safe-unsafe, cloudy-sunny, healthy-ill, and so on. First we codify these classes, saying that the predicted variable $y$ can only take two values *0* or *1*, that is, $y \in \{0,1\}$. Stated in this way, the problem begins to sound similar to the coin-flipping problem from the first two chapters, and we may remember that we used the Bernoulli distribution as the likelihood. The difference is that now $\theta$ is not going to be generated from a beta distribution; instead $\theta$ is going to be defined by a linear model. A linear model could potentially return any value from the real line, but the Bernoulli distribution is expecting values limited to the interval *[0, 1]*. So we use the inverse link function to put the values returned by the linear model in a range suitable to the Bernoulli distribution, effectively transforming a linear regression model into a classification model:

$$\theta = \log istic\left(\alpha + \beta \boldsymbol{X}\right)$$

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

The following Kruschke diagram shows the logistic regression model including, as it should be, the priors. Notice that the main difference with the simple linear regression is the use of a Bernoulli distribution instead of a Gaussian distribution (or Student's t-distribution) and the use of the logistic function that allows us to generate a $\theta$ parameter in the range *[0, 1]*, suitable for feeding the Bernoulli distribution:




# The iris dataset

We are going to apply the logistic regression to the iris dataset. So before working on the model, we are going to explore the data. The iris dataset is a classic dataset containing information about the flowers of three species from the genus iris: these are setosa, virginica, and versicolor. These are going to be our dependent variables, the classes we want to predict. We have 50 individuals of each species and for each individual the dataset contains four variables (or features, as it is more common to say in a machine learning setting). These four variables are going to be our independent variables and they are the petal length, the petal width, the sepal length, and the sepal width. Sepals are modified leaves, whose function is generally related to protecting the flowers in bud. The iris dataset is distributed with `seaborn` and we can put it into a Pandas dataframe by doing the following:

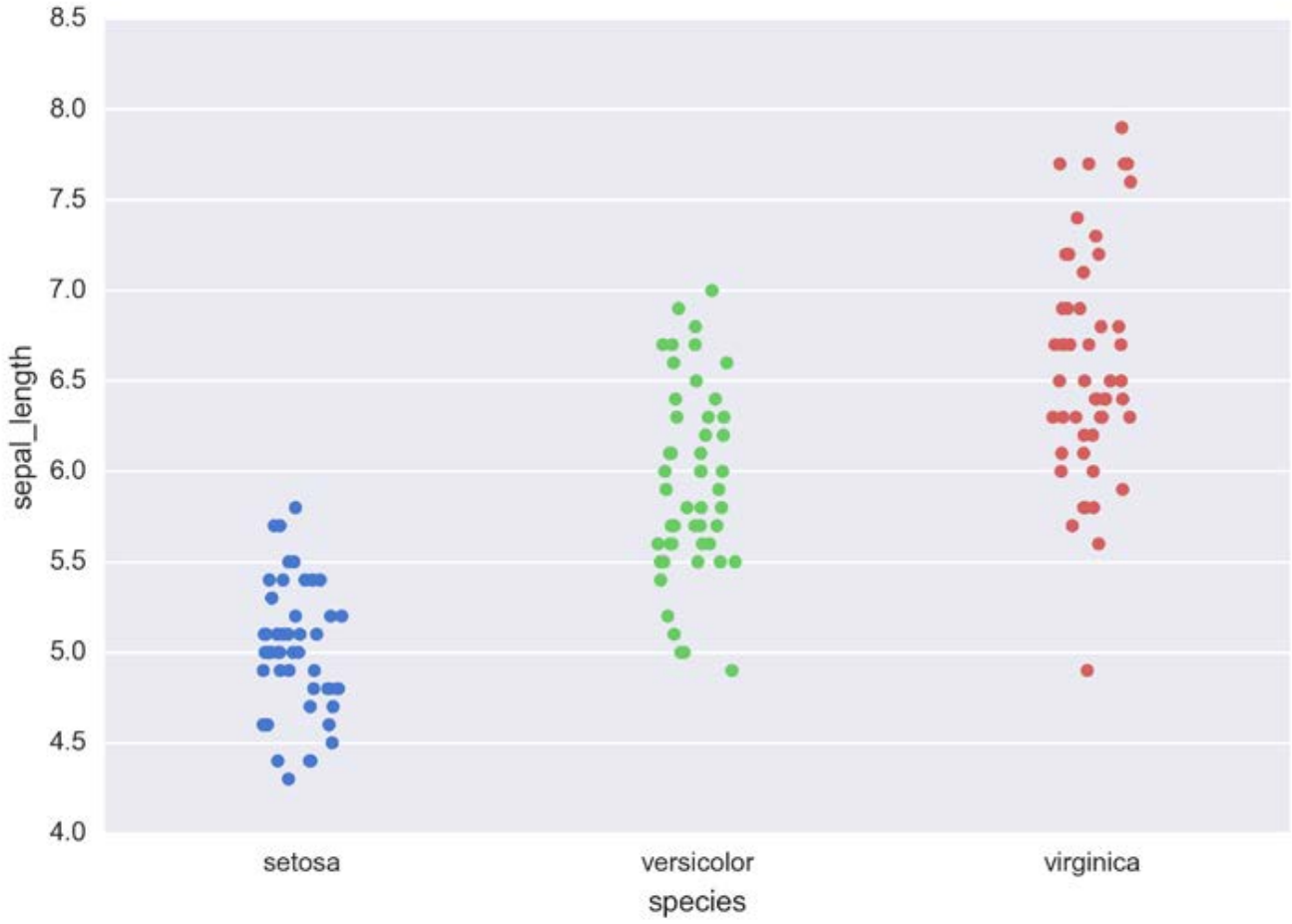
```
iris = sns.load_dataset("iris")
iris.head()
```

| | 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 |

| | sepal_length | sepal_width | petal_length | petal_width | species |
|---|---|---|---|---|---|
| 2 | 4.7 | 3.2 | 1.3 | 0.2 | setosa |
| 3 | 4.6 | 3.1 | 1.5 | 0.2 | setosa |

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

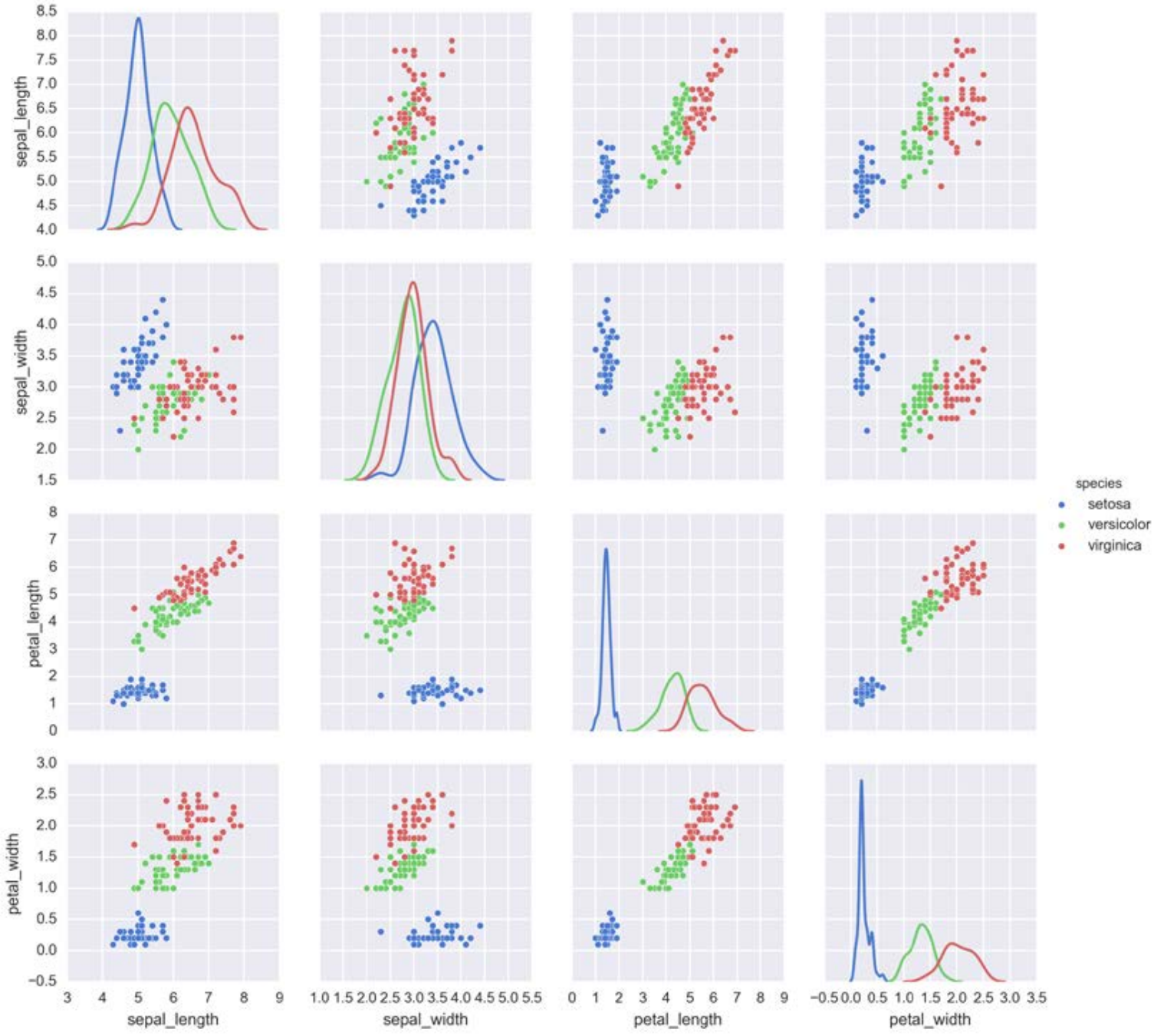
```
sns.stripplot(x="species", y="sepal_length", data=iris,
  jitter=True)
```




Notice in the `stripplot` figure that 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, and is just a trick we add, using 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 classes `setosa`, `versicolor`, or `virginica`. 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 `pairplot`. We have a scatter plot arranged in a *4x4* grid, since we have four features in the `iris` dataset. The grid is symmetrical, with the upper and lower triangles showing the same information. Since the diagonal scatter plot should correspond to the variable against itself, we have replaced those scatter plots with a `kde` plot for each feature. Inside each subplot, we have the three `species` (or classes) represented with a different color, the same used in the previous plot:

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




Before continuing, take some time to study the previous plots and to try to get familiar with the dataset and how the variables and classes are related.

# The logistic model applied to the iris dataset

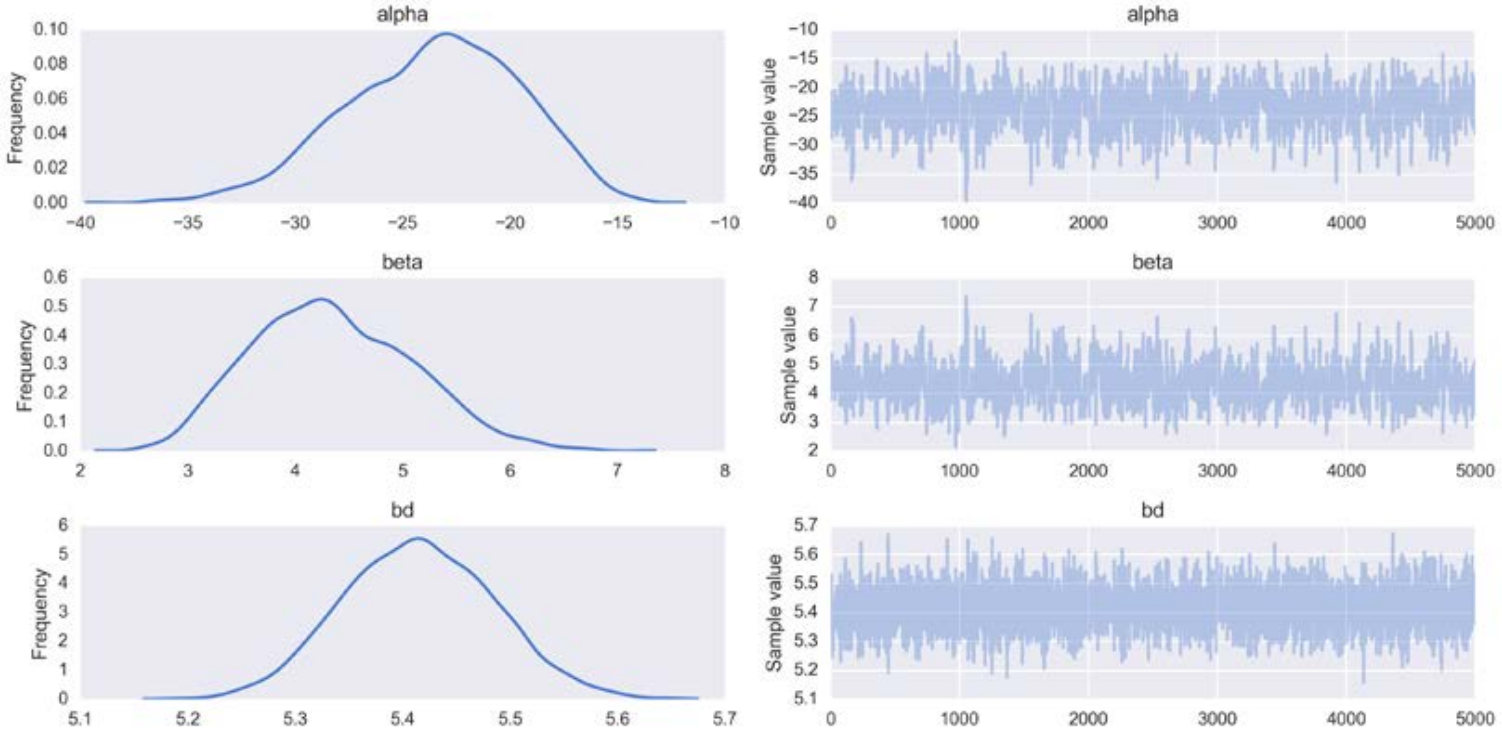
We are going to begin with the simplest possible classification problem: two classes, `setosa` and `versicolor`, and just one independent variable or feature, the sepal length. As is usually done, we are going to encode the categorical variables `setosa` and `versicolor` (with the numbers *0* and *1*).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

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

Now that we have the data in the proper format, we can finally build the model with PyMC3. Notice how the first part of the following model resembles a linear regression model. Also pay attention to the two deterministic variables; `theta` and `bd`. `theta` is the result of applying the logistic function to the variable `mu` and `bd` is the boundary decision, the value used to separate between classes; we will discuss it later in detail. Another point worth mentioning is that instead of explicitly writing the logistic function as follows, we could have called the Theano function `sigmoid`. This function is aliased in PyMC3 as `pm.math.sigmoid`.

As with other linear models, centering and/or standardizing the data can help with the sampling. Nevertheless, for this example we are going to proceed without further modification of the data:

```
with pm.Model() as model_0:
    alpha = pm.Normal('alpha', mu=0, sd=10)
    beta = pm.Normal('beta', mu=0, sd=10)

    mu = alpha + pm.dot(x_0, beta)
    theta = pm.Deterministic('theta', 1 / (1 + pm.math.exp(-mu)))
    bd = pm.Deterministic('bd', -alpha/beta)

    yl = pm.Bernoulli('yl', theta, observed=y_0)
    start = pm.find_MAP()
    step = pm.NUTS()
    trace_0 = pm.sample(5000, step, start)
chain_0 = trace_0[1000:]
varnames = ['alpha', 'beta', 'bd']
pm.traceplot(chain_0, varnames)
```



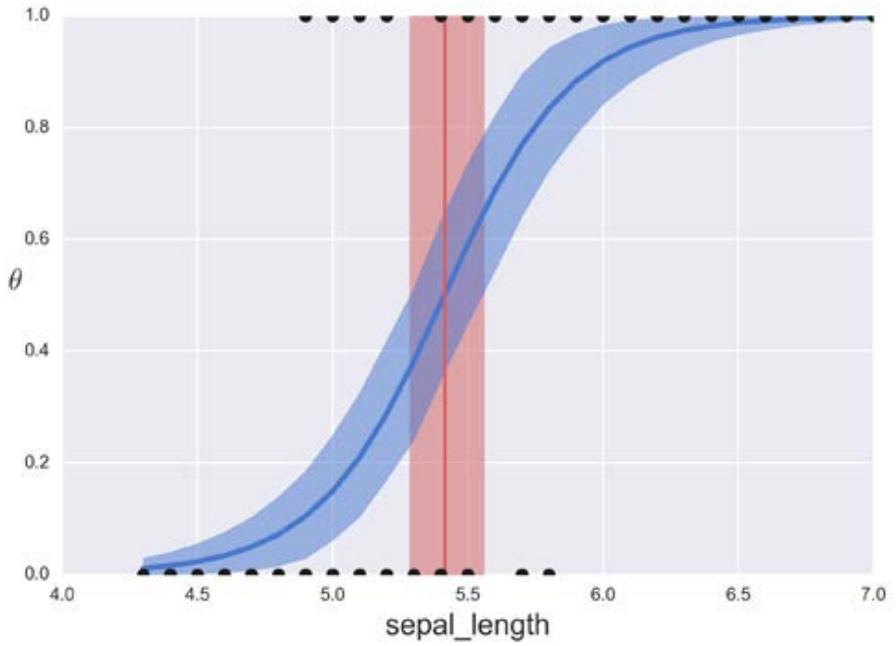

As usual, we also print the summary of the posterior. Later we will compare the value we get for the boundary decision with a value computed using another method.

```
pm.df_summary(trace_0, varnames)
```

| | mean | sd | mc_error | hpd_2.5 | hpd_97.5 |
|---|---|---|---|---|---|
| alpha | -23.49 | 4.07 | 1.77e-01 | -31.38 | -15.72 |
| beta | 4.34 | 0.75 | 3.28e-02 | 2.88 | 5.75 |
| bd | 5.42 | 0.07 | 1.09e-03 | 5.27 | 5.55 |

Now we are going to plot the data together with the fitted sigmoid (S-shaped) curve:

```
theta = trace_0['theta'].mean(axis=0)
idx = np.argsort(x_0)
plt.plot(x_0[idx], theta[idx], color='b', lw=3);
plt.axvline(trace_0['bd'].mean(), ymax=1, color=''r'')
bd_hpd = pm.hpd(trace_0['bd'])
plt.fill_betweenx([0, 1], bd_hpd[0], bd_hpd[1], color='r',
  alpha=0.5)

plt.plot(x_0, y_0, 'o', color='k')
theta_hpd = pm.hpd(trace_0['theta'])[idx]
plt.fill_between(x_0[idx], theta_hpd[:,0], theta_hpd[:,1],
  color='b', alpha=0.5)

plt.xlabel(x_n, fontsize=16)
plt.ylabel(r'$\theta$', rotation=0, fontsize=16)
```




The preceding figure shows the sepal length versus the flower species (*setosa = 0*, *versicolor =1*). A blue S-shaped line is the mean value of theta. This line can be interpreted as the probability of a flower being versicolor given that we know the value of the sepal length or, in general, $p(y = 1 \mid x)$. The semi-transparent blue band is the *95%* HPD interval. Notice that in this sense, logistic regression is actually a regression since we are regressing the probability that a data point belongs to *class 1*, given a feature (or a linear combination of features, as we will see soon). Nevertheless we should keep present that we are observing only a dichotomous variable and inferring the continuous probability. Then we introduce a rule to turn that continuous probability into a *class 0-1* response. The boundary decision is represented as a red line in the plot together with its corresponding *95%* HPD as a red band. According to the boundary decision, the $\boldsymbol{x}$ values (sepal length in this case) to the left corresponds to the *class 0* (setosa), and the values to the right to the *class 1* (versicolor). This decision boundary is defined as the value of $x_i$, for which $y = 0.5$. And it turns out to be $-\frac{\alpha}{\beta}$, as we can derive as follows:

From the definition of the model we have the relationship:

$$\theta = logistic\left(\alpha + \beta \boldsymbol{X}\right)$$

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

$$0.5 = logistic\left(\alpha + \beta x_i\right) \Leftrightarrow 0 = \alpha + \beta x_i$$

Reordering the preceding equation, we find that the value $x_i$, for which, $\theta = 0.5$ corresponds to the expression:

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

It is worth mentioning that the boundary decision is a scalar, that is, a single number. This makes sense since we are working with unidimensional data. Therefore, we just need a scalar to divide the data into two groups or classes. It is very important to remark that while this boundary decision may sound reasonable, there is nothing special about the value *0.5*, other than that it is just the number in the middle between *0* and *1*. We may argue this boundary is only reasonable if we are OK about making a mistake either in 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 to the misclassification does not need to be symmetrical.

# Making predictions

Once we have the $\alpha$ and β parameters we can use them to classify new data. We can create a function that, given the sepal length, returns the probability of a flower being versicolor instead of setosa. A very rudimentary function, just as a proof of concept, is:

```
def classify(n, threshold):
    """
    A simple classifying function
    """
    n = np.array(n)
    mu = trace_0['alpha'].mean() + trace_0['beta'].mean() * n
    prob = 1 / (1 + np.exp(-mu))
    return prob, prob > threshold
classify([5, 5.5, 6], 0.4)
```

It is clear from the previous example that it is not possible to unambiguously classify the classes `setosa` and `versicolor` based on the sepal length feature alone. This is not surprising; in fact it becomes evident if we paid attention to the `joinplot` we did previously. Observe that according to our data, versicolor flowers can have sepals as short as ~*4.9* and setosa flowers as large as ~*5.8*. In other words, there is an overlap of the sepal length values for setosa and versicolor in the range ~*4.9* to ~*5.8*.

What will happen if we use one of the other variables? Check exercise 1 to explore this question.

# Multiple logistic regression

In a similar fashion as with the multiple linear regression, the multiple logistic regression is about using more than one independent variable. Let us try combining the sepal length and the sepal width. Remember that 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 if you are not much interested in how we can derive the boundary decision.

From the model, we have the following:

$$\theta = logistic\left(\alpha + \beta_0 \boldsymbol{x}_0 + \beta_1 \boldsymbol{x}_1\right)$$

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

$$0.5 = logistic\left(\alpha + \beta_0 \boldsymbol{x}_0 + \beta_1 \boldsymbol{x}_1\right) \Leftrightarrow 0 = \alpha + \beta_0 \boldsymbol{x}_0 + \beta_1 \boldsymbol{x}_1$$

Reordering, we find the value of $x_1$ for which $\theta = 0.5$ corresponds to the following expression:

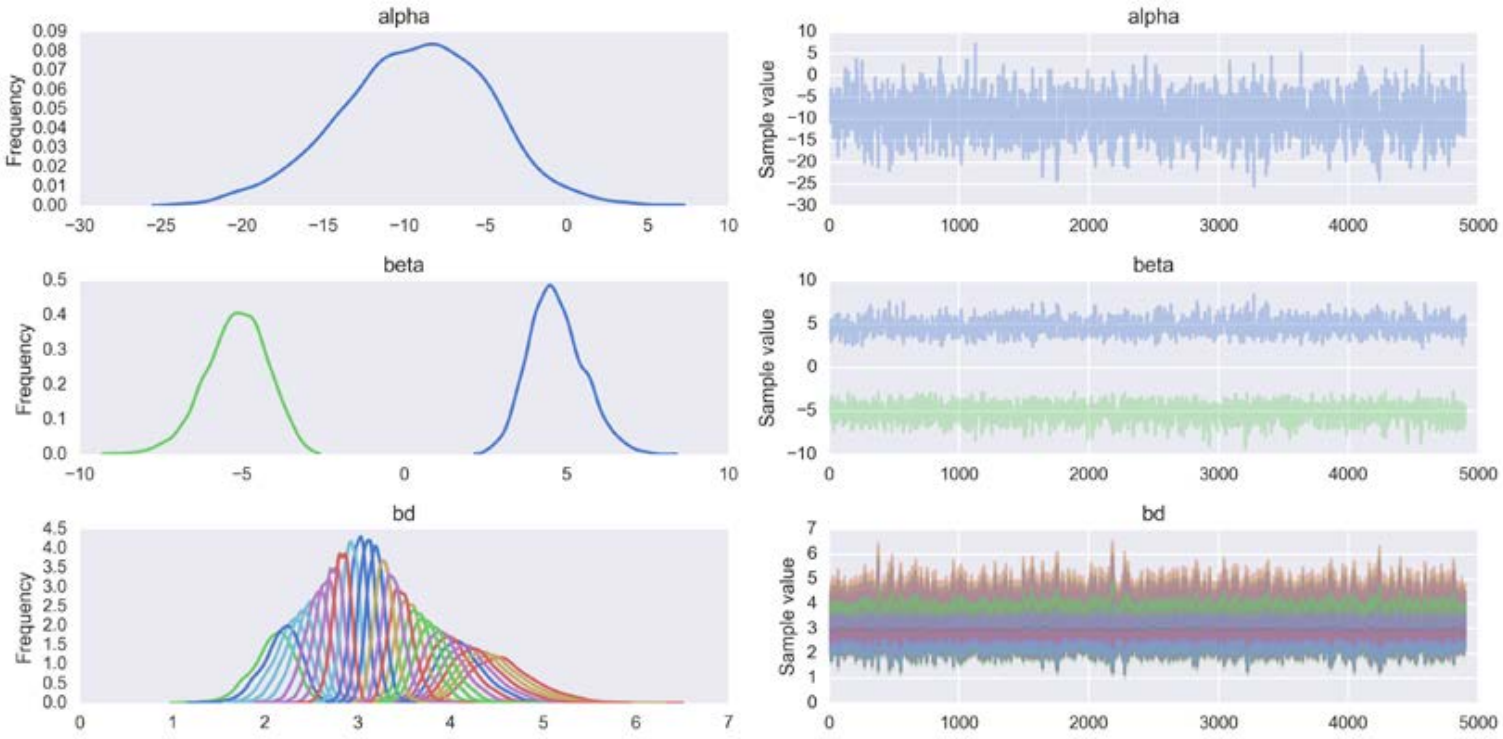
$$\boldsymbol{x}_1 = -\frac{\alpha}{\beta_1} + \left(-\frac{\beta_0}{\beta_1}\boldsymbol{x}_0\right)$$

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 wish. The boundary being a line is totally reasonable, isn't it? If we have one feature, we have uni-dimensional 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:

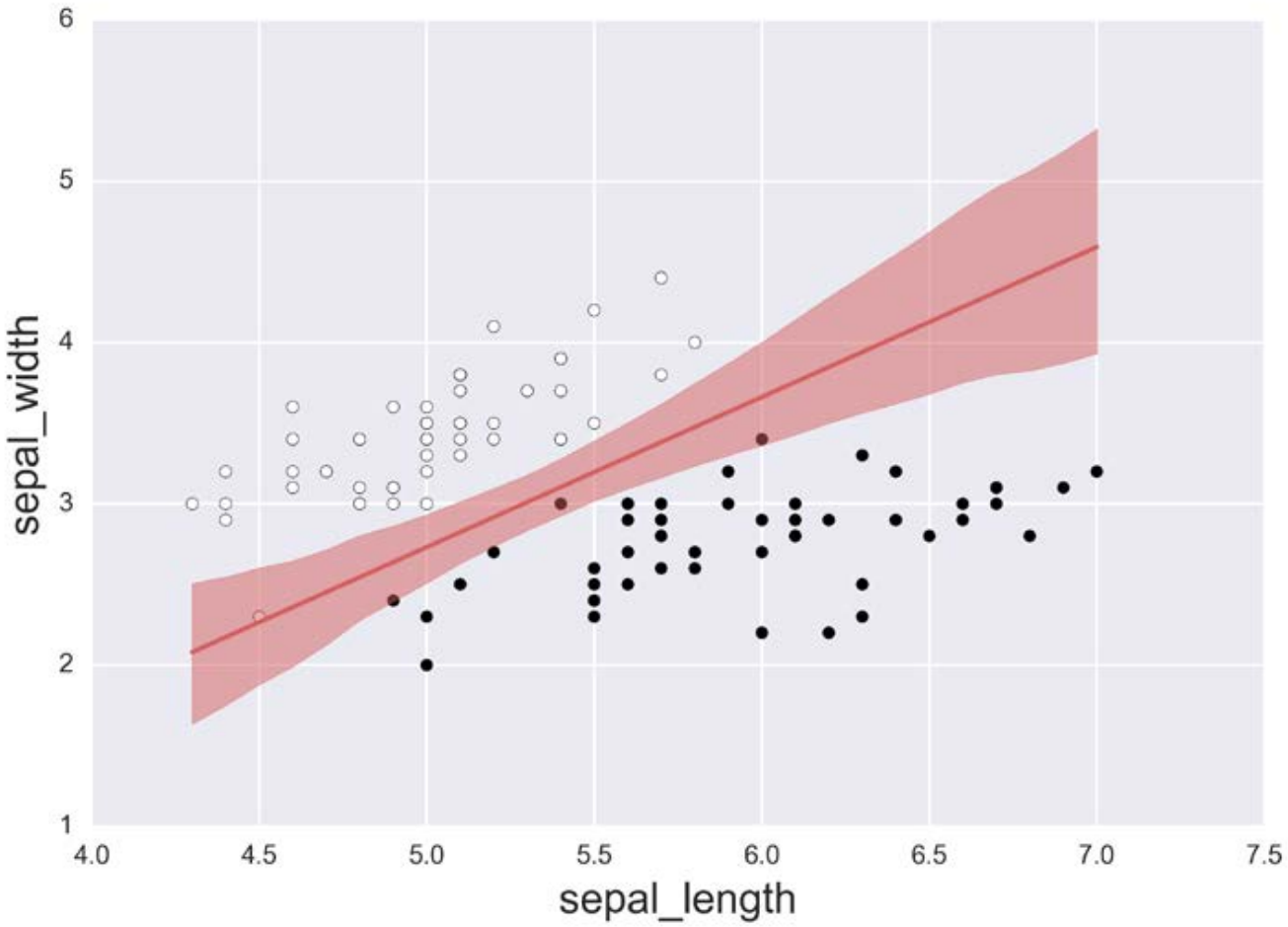
```
with pm.Model() as model_1:
    alpha = pm.Normal('alpha', mu=0, sd=10)
    beta = pm.Normal('beta', mu=0, sd=2, shape=len(x_n))

    mu = alpha + pm.math.dot(x_1, beta)
    theta = 1 / (1 + pm.math.exp(-mu))
    bd = pm.Deterministic('bd', -alpha/beta[1] -
  beta[0]/beta[1] * x_1[:,0])

    yl = pm.Bernoulli('yl', p=theta, observed=y_1)

    trace_1 = pm.sample(5000)
chain_1 = trace_1[100:]
varnames = ['alpha', 'beta']
pm.traceplot(chain_1)
```




Notice in the preceding figure that now we do not get a single curve for the boundary decision as we did in the previous example. Now we get 100 curves, one for each data point.

As we did for a single predictor variable, we are going to plot the data and the decision boundary. In the following code we are going to omit plotting the sigmoid curve (that is now a 2D curve). If you want, go ahead and make a 3D plot with the sigmoid curve embedded:

```
idx = np.argsort(x_1[:,0])
bd = chain_1['bd'].mean(0)[idx]
plt.scatter(x_1[:,0], x_1[:,1], c=y_0)
plt.plot(x_1[:,0][idx], bd, color='r');

bd_hpd = pm.hpd(chain_1['bd'])[idx]
plt.fill_between(x_1[:,0][idx], bd_hpd[:,0], bd_hpd[:,1],
  color='r', alpha=0.5);

plt.xlabel(x_n[0], fontsize=16)
plt.ylabel(x_n[1], fontsize=16)
```



The boundary decision is a straight line, as we have already seen. Do not get confused by the curved aspect of the *95%* HPD band. The apparent curvature is the result of having multiple lines pivoting around a central region, which is roughly the narrower part of the HPD band. Remember we saw something similar in the previous chapter, although probably less obvious.

# Dealing with correlated variables

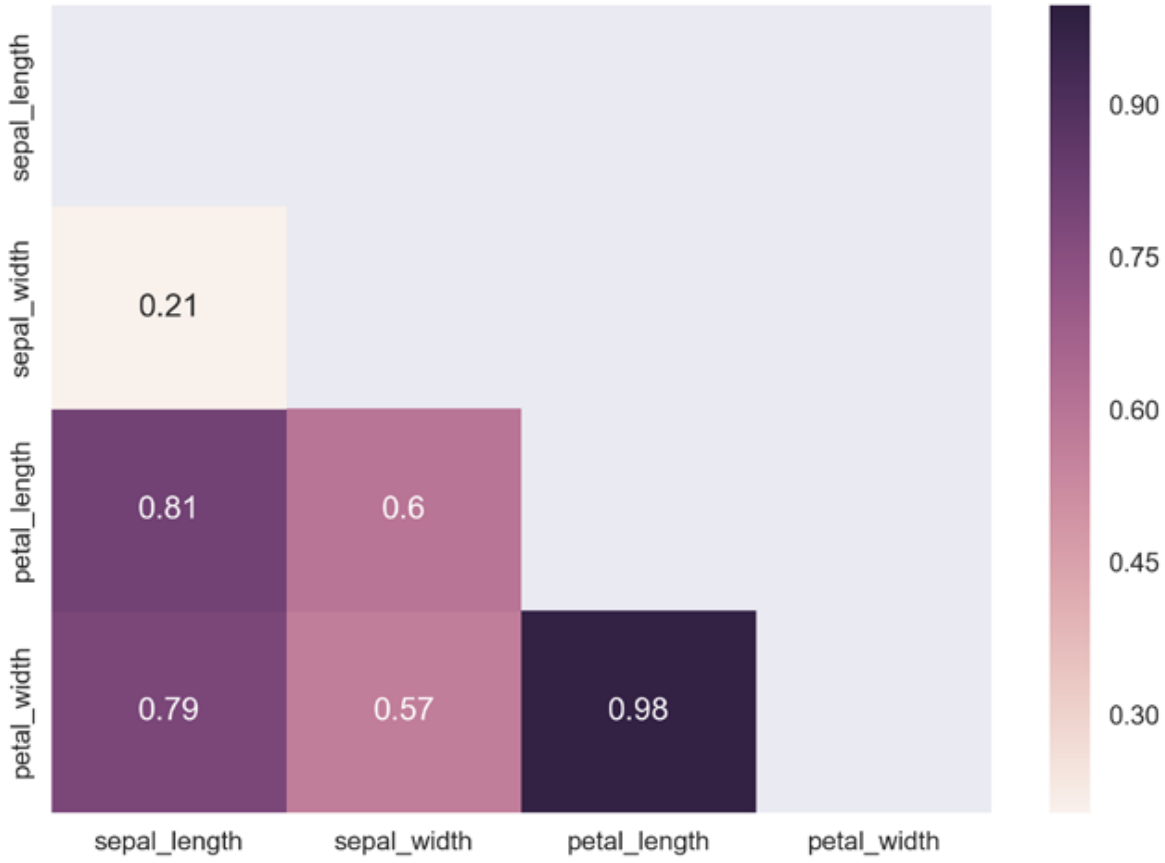
We know from the previous chapter that tricky things await us when we deal with (highly) correlated variables. For example, what will be the result of running the previous model but this time using the variables petal width and petal length?

If you did the previous exercise, you may have noticed the beta coefficients are broader now than before and also the *95%* HPD (red band in the previous plot) is now much wider. The following heat map shows 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). As we saw, correlated variables translate into wider combinations of coefficients that are able to explain the data, or from the 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. As we saw, one solution is to avoid using correlated variables, but this solution may not be adequate. Another option is to put more information into the prior; this can be archived using informative priors if we have useful information, or more generally, using weakly informative priors. Andrew Gelman and the Stan Team recommend using the following prior when performing logistic regression:

$$\beta \sim Student\ t\left(0, \nu, s\right)$$

Here $s$ should be chosen in order to weakly inform about the expected values for the scale. The normality parameter $\nu$ is suggested to be around *3-7*. What this prior is saying is that we expect the coefficient to be small, but we put fat tails because this leads us to a more robust model than using a Gaussian distribution. Remember our discussion about robust models in *Chapter 3, Juggling with Multi-Parametric and Hierarchical Models* and *Chapter 4, Understanding and Predicting Data with Linear Regression Models*:

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




In the preceding plot, we have used a mask to remove the upper triangle and the diagonal elements of the heat map, because these are uninformative or redundant. Also notice 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.

# Dealing with unbalanced classes

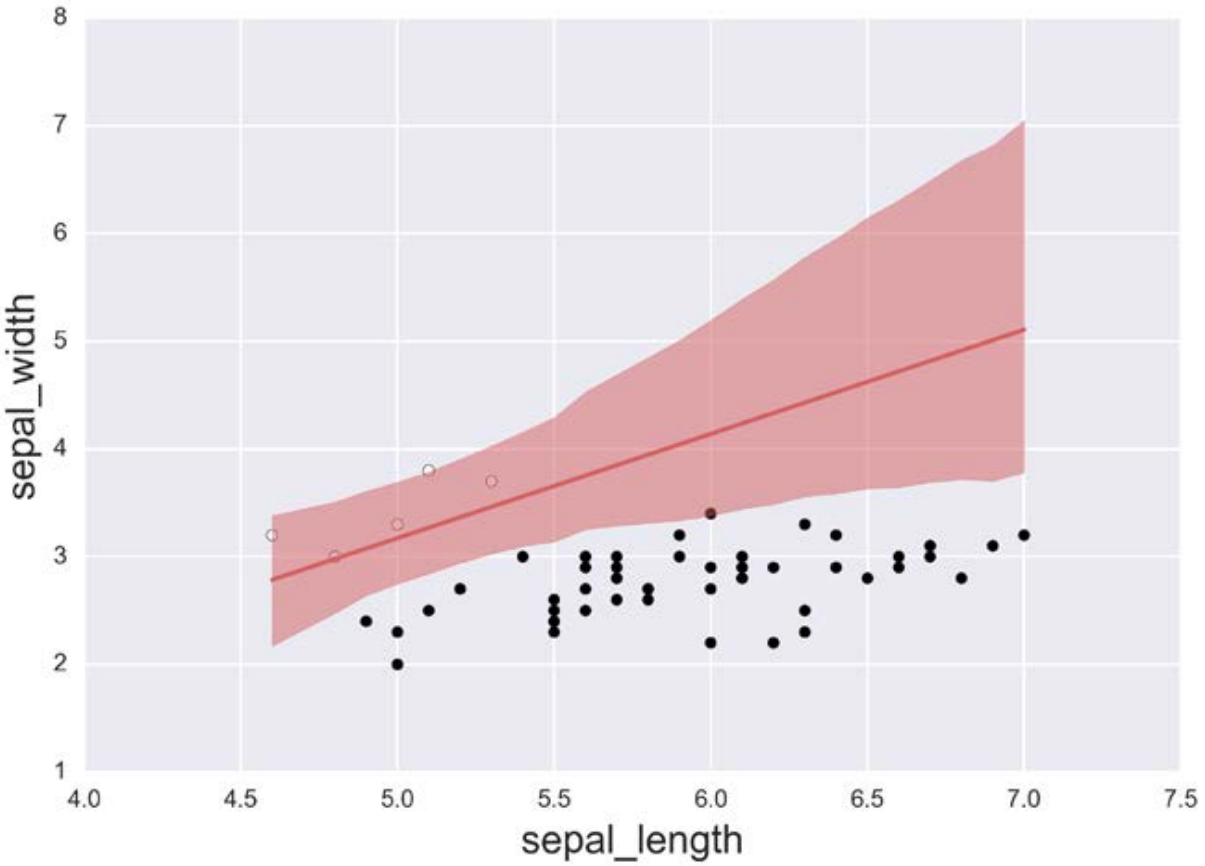
One of the nice features of the iris dataset is that it is completely balanced, in the sense that each category has exactly the same number of subjects (or instances). We have 50 setosas, 50 versicolors, and 50 virginicas. This is something to thank Fisher for, unlike his dedication to popularizing the use of p-values ;-). In practice, many datasets consist of unbalanced data, that is, there are many more data points from one class than from the other. 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 just as before. You can actually do it with your computer. Instead I am just going to plot the results here:

```
idx = np.argsort(x_3[:,0])
bd = trace_3['bd'].mean(0)[idx]
plt.scatter(x_3[:,0], x_3[:,1], c=y_3)
plt.plot(x_3[:,0][idx], bd, color='r');

bd_hpd = pm.hpd(trace_3['bd'])[idx]
plt.fill_between(x_3[:,0][idx], bd_hpd[:,0], bd_hpd[:,1], color='r',
alpha=0.5);

plt.xlabel(x_n[0], fontsize=16)
plt.ylabel(x_n[1], fontsize=16)
```



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. It can be even worse when the classes do not have a nice gap like in this example and there is more overlap between them. But wait a minute! You may argue that I am cheating here since the wider uncertainty could be the product of having less total data and not just less setosas than versicolors! That could be a valid point; try doing exercise two to verify that what explains this plot is the unbalanced data.

# How do we solve this problem?

Well, the obvious solution is to get a dataset with roughly the same number of data points per class. So this can be important to have in mind if you are collecting or generating the data. If you have no control over the dataset, then be careful when interpreting the result for unbalanced data. Check the uncertainty of the model and run some posterior predictive checks to see if the results are useful to you. Another option will be to put more prior information, if available, and/or run an alternative model as explained later in this chapter.

# Interpreting the coefficients of a logistic regression

We must be careful when interpreting the coefficient of a logistic regression because the logistic inverse link function introduce non-linearities.

$$\theta = logistic\left(\alpha + \beta \boldsymbol{X}\right)$$

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

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

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

$$logit\left(\theta\right) = \alpha + \beta \boldsymbol{X}$$

Or equivalently:

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

Remember that $\theta$ in our model was the probability of *y=1*, thus:

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

The quantity $\frac{p(y=1)}{1-p(y=1)}$ is known as the **odds**. The odds are an alternative way to represent probabilities. While the probability of getting *2* by rolling a fair die is *1/6*, the odds for the same event are *1* to *5* (or equivalently *0.2*), that is, we have one favorable event and five unfavorable events. Besides, odds are often used by gamblers because odds provide a more intuitive tool than raw probabilities when thinking about the proper way to bet.

Going back to the logistic regression, we can see the coefficient $\beta$ indicates the increase in the log odds by unit increase of the $\boldsymbol{x}$ variable. It is important to remark that $\beta$ does not indicate how much *p(y=1)* will change with an increase of *x* since the relationship between $\boldsymbol{x}$ and *p(y=1)* is not linear. If $\beta$ is positive, increasing *x* will increase *p(y=1)* by some amount, but the amount will depend on the current value of *x*. This is reflected by the S-shaped line in the first plot we did; the slope of $\boldsymbol{y}$ versus $\boldsymbol{x}$ changes with *x*, while the slope of log odds versus $\boldsymbol{x}$ is a linear function.

# Generalized linear models

So let's summarize what we have done in this chapter so far and how it is related to the linear regression model we saw in the previous chapter. What we have done is extend the model to deal with categorical data instead of quantitative data. We have done this by introducing the concept of the inverse link function and by replacing the Gaussian distribution with another one (the Bernoulli distribution). In summary, we have adapted the simple linear model from the previous chapter to a different data/problem by introducing changes into the likelihood, priors, and the inverse link function connecting both.

The logistic model is not the only possible extension of the linear regression model. In fact, there is a whole family of models that can be considered generalizations of the linear model and these are known as **generalized linear models** (**GLMs**). Some GLMs very commonly used in statistics are:

- The softmax regression (that we will see next), an extension of the logistic regression for more than two classes.
- The **ANalysis Of VAriance** (**ANOVA**), where we have a quantitative predicted variable and more than two categorical predictors. The ANOVA is a model used to compare between groups in a similar fashion as what we saw in *Chapter 3*, *Juggling with Multi-Parametric and Hierarchical Models*, but using a model framed as a linear regression.
- The Poisson regression and other models for counting data. We will see a variation of the Poisson regression model in *Chapter 7*, *Mixture Models*.

If you want to learn more about this, specially the ANOVA model, which we will not cover in this book, I highly recommend the book *Doing Bayesian Data Analysis* by *John Kruschke*, where he does an excellent job describing how to build many Bayesian models from the GLM family.

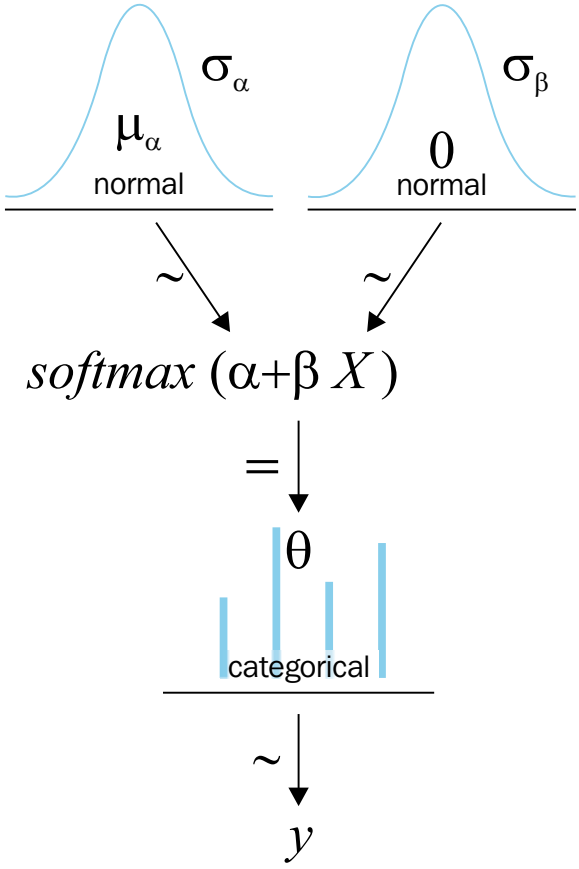
# Softmax regression or multinomial logistic regression

We saw how to classify outcomes when we have two classes, and now we are going to generalize what we learned to more than two classes. One way of doing it is by creating a multinomial (instead of binomial) logistic regression. This model is also known as **softmax regression**, since we use the *softmax* function instead of the logistic. The *softmax* function is as follows:

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

Notice that to obtain the output of the *softmax* function for the *i-esim* element of a vector $\boldsymbol{\mu}$, we take the exponential of the *i-esim* value divided by the sum of all the exponentiated values in the $\boldsymbol{\mu}$ vector. The *softmax* guarantees that we will get positive values that sum up to *1*. The *softmax* function is reduced to the logistic function when *k*=2. As a side note, the *softmax* function has the same form as the Boltzmann distribution used in statistical mechanics, a very powerful branch of physics dealing with the probabilistic description of molecular systems. The Boltzmann distribution (and the *softmax* in some fields) has a parameter call temperature (*T*) dividing $\boldsymbol{\mu}$ in the preceding equation; when $T \to \infty$ the probability distribution becomes flat and all states are equally likely, and when $T \to 0$ only the most probable state gets populated, and hence the *softmax* behaves like a max function, which clarifies its name.

The other difference between the softmax regression model and logistic regression 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!

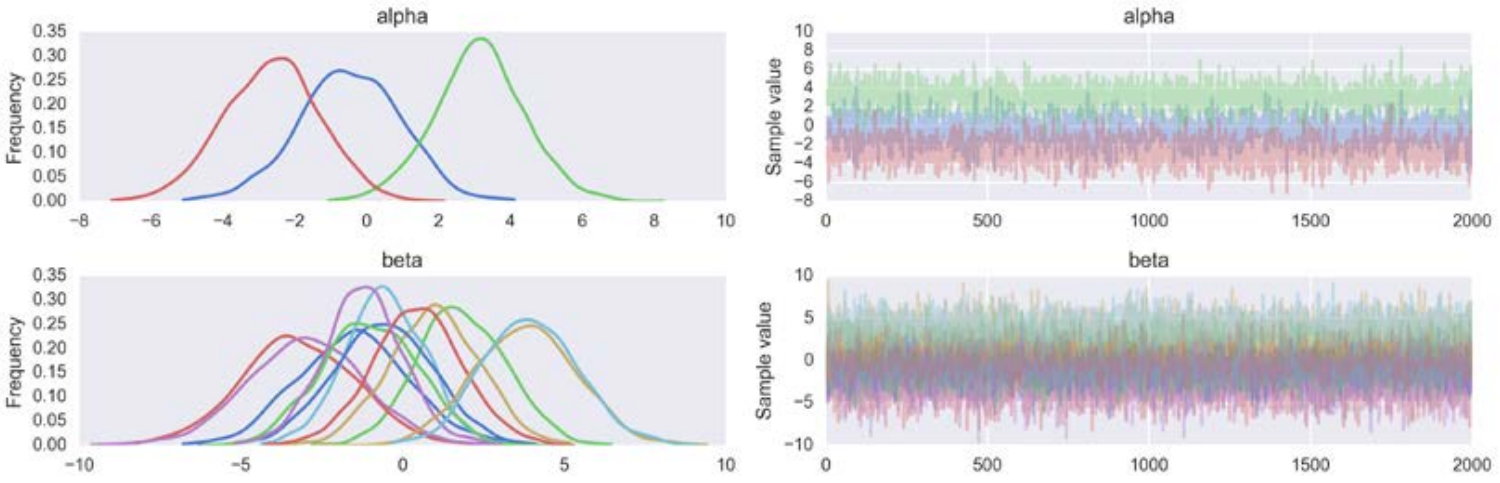


We are going to continue working with the `iris` dataset, only this time we are going to use its three classes (`setosa`, `versicolor`, and `virginica`) and its four features (sepal length, sepal width, petal length, and petal width). We are also going to standardize the data, since this will help the 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 few changes between the logistic and softmax model. Notice the shapes of the `alpha` and `beta` coefficients. Here we have used the softmax function from Theano; we have used the idiom `import theano.tensor as tt`, which is the convention used by the PyMC3 developers:

```
with pm.Model() as model_s:
    alpha = pm.Normal('alpha', mu=0, sd=2, shape=3)
    beta = pm.Normal('beta', mu=0, sd=2, shape=(4,3))

    mu = alpha + pm.dot(x_s, beta)

    theta = tt.nnet.softmax(mu)

    yl = pm.Categorical('yl', p=theta, observed=y_s)
    start = pm.find_MAP()
    step = pm.NUTS()
    trace_s = pm.sample(2000, step, start)
pm.traceplot(trace_s)
```



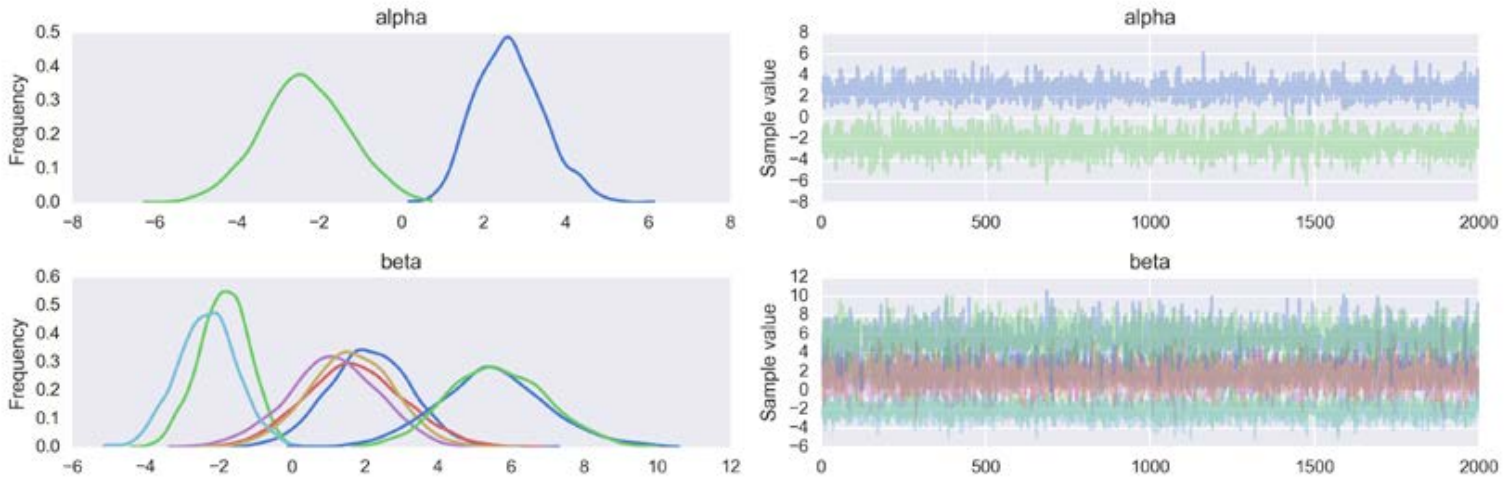
How well does our model perform? Let us find out by checking how many cases we can predict correctly. In the following code, we just use the `mean` of the parameters to compute the probability of each data point to belong to each of the three classes, then we assign the class by using the `argmax` function. And we compare the result with the observed values:

```
data_pred = trace_s['alpha'].mean(axis=0) + np.dot(x_s,
  trace_s['beta'].mean(axis=0))
y_pred = []
for point in data_pred:
    y_pred.append(np.exp(point)/np.sum(np.exp(point), axis=0))
np.sum(y_s == np.argmax(y_pred, axis=1))/len(y_s)
```

The result is that we classify correctly *~98%* of the data points, that is, we miss only three cases. That is really a very good job. Nevertheless, a true test to evaluate the performance of our model will be to test it on data not fed to the model. Otherwise, we will be most probably overestimate its abilities to generalize to other data. We will discuss this subject in detail in the next chapter. 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 have already encountered for correlated data in linear/logistic regression or with perfectly separable classes. In this case, the wide posterior is due to the condition that all probabilities must sum to *1*. Given this condition, we are using more parameters than we need to fully specify the model. In simple terms, if you have *10* numbers that sum to *1*, you just need to give me *9* 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:
    alpha = pm.Normal('alpha', mu=0, sd=2, shape=2)
    beta = pm.Normal('beta', mu=0, sd=2, shape=(4,2))

    alpha_f = tt.concatenate([[0] , alpha])
    beta_f = tt.concatenate([np.zeros((4,1)) , beta], axis=1)

    mu = alpha_f + pm.math.dot(x_s, beta_f)
    theta = tt.nnet.softmax(mu)

    yl = pm.Categorical('yl', p=theta, observed=y_s)
    start = pm.find_MAP()
    step = pm.NUTS()
    trace_sf = pm.sample(5000, step, start)
```




# 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 $p(\boldsymbol{x} \mid \boldsymbol{y})$, that is, the probability of a given class knowing $\boldsymbol{x}$, which is some feature we measured to members of that class. 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 boundary that allows us to assign classes.

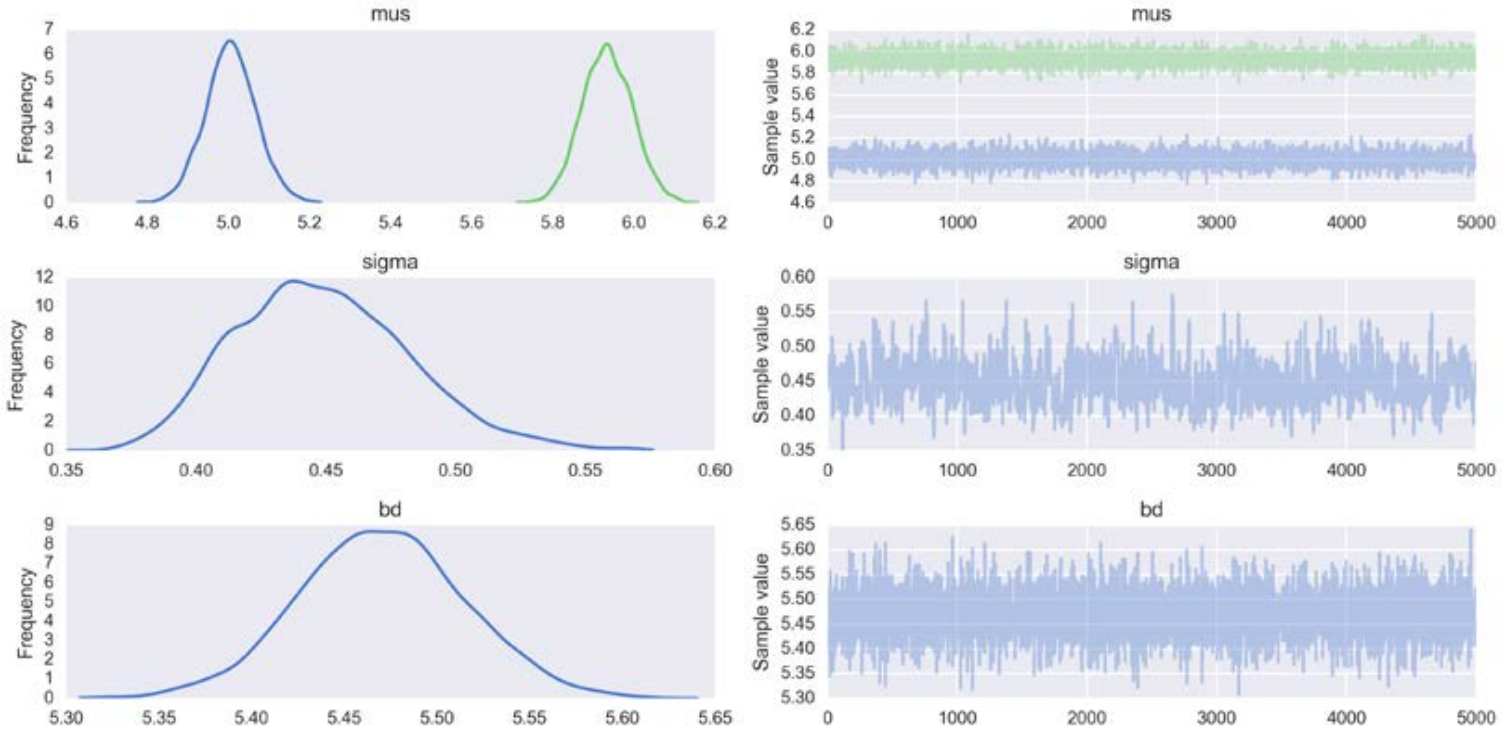
This approach is not unique. One alternative is to model first $p(\boldsymbol{x} \mid \boldsymbol{y})$, that is, the distribution of $\boldsymbol{x}$ for each class, and then assign the classes. This kind of model is called a **generative classifier** because we are creating a model from which we can generate samples from each class. On the contrary, logistic regression is a type of discriminative classifier since it tries to classify by discriminating classes but we cannot generate examples from each class.

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, using the same data.

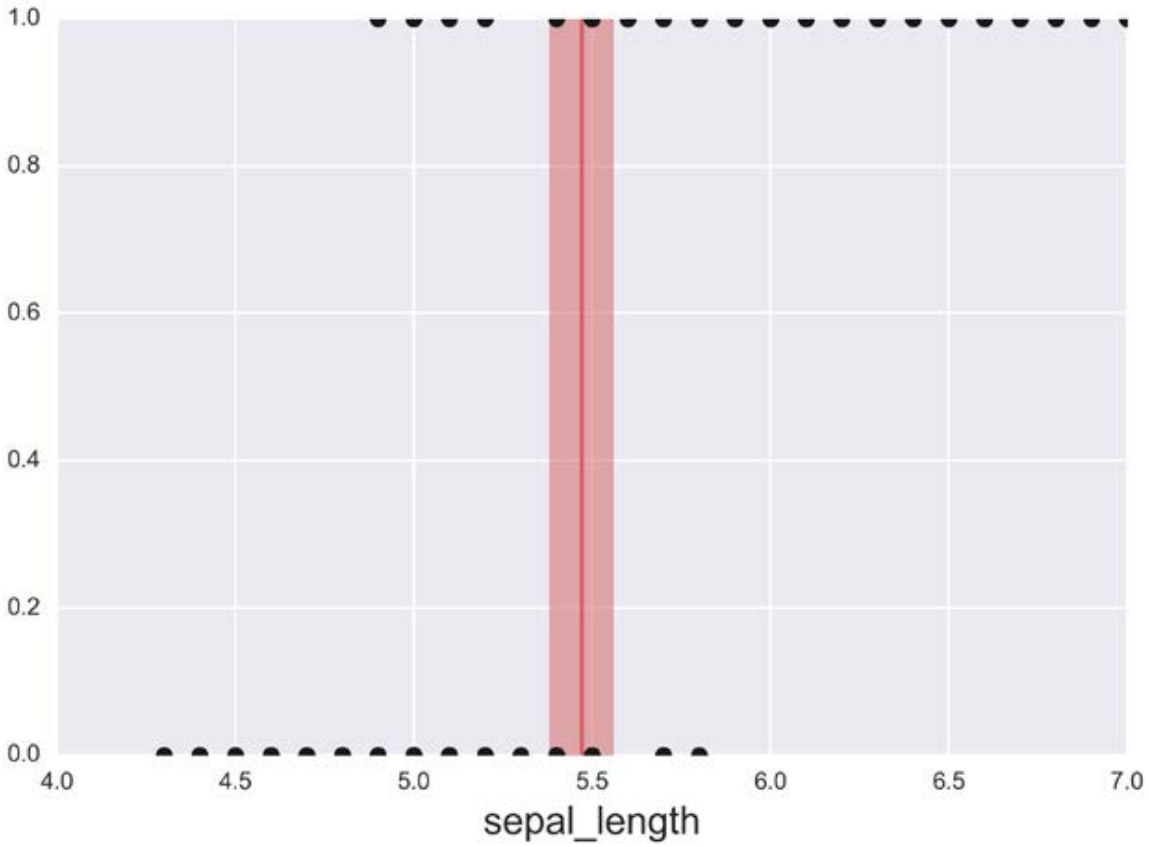
Following is a PyMC3 implementation of a generative classifier. From the code, you can see that now the boundary decision is defined as the average between both estimated Gaussian means. This is the correct boundary decision when the distributions are normal and their standard deviations are equal. These 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:

    mus = pm.Normal('mus', mu=0, sd=10, shape=2)
```

```
    sigmas = pm.Uniform('sigmas', 0, 10)

    setosa = pm.Normal('setosa', mu=mus[0], sd=sigmas[0],
observed=x_0[:50])
    versicolor = pm.Normal('setosa', mu=mus[1], sd=sigmas[1],
observed=x_0[50:])

    bd = pm.Deterministic('bd', (mus[0]+mus[1])/2)

    start = pm.find_MAP()
    step = pm.NUTS()
    trace = pm.sample(5000, step, start)
```




Now we are going to plot a figure showing the two classes (`setosa = 0` and `versicolor = 1`) against the values for sepal length, and also the boundary decision as a red line and the *95%* HPD interval for it as a semitransparent red band.




As you may have noticed, the preceding figure is pretty similar to the one we plotted at the beginning of this chapter. Also check the values of the boundary decision in the following summary:

```
pm.df_summary(trace_lda)
```

| | **mean** | **sd** | **mc_error** | **hpd_2.5** | **hpd_97.5** |
|---|---|---|---|---|---|
| mus__0 | 5.01 | 0.06 | 8.16e-04 | 4.88 | 5.13 |
| mus__1 | 5.93 | 0.06 | 6.28e-04 | 5.81 | 6.06 |
| sigma | 0.45 | 0.03 | 1.52e-03 | 0.38 | 0.51 |
| bd | 5.47 | 0.05 | 5.36e-04 | 5.38 | 5.56 |

Both the LDA model and the logistic regression gave 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 common covariance matrices when working with more than one feature). This leads to a model known as **quadratic linear discriminant** (**QDA**), since now the decision boundary is not linear but quadratic.

In general, an LDA or QDA model will work better than a logistic regression when the features we are using are more or less Gaussian distributed and the logistic regression will perform better in the opposite case. One advantage of the discriminative 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 in 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 used 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. Notice that in the preceding model we just did that but in a more Bayesian way. We estimate the parameters of the two Gaussians and then we plug those estimates into a formula. Where do such formulae come from? Well, without entering into details, 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 where we want to relax the normality assumption, such as, for example using a Student's t-distribution (or a multivariate Student's t-distribution, or something else). 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.

# Summary

In this chapter, we learned how to extend the simple linear regression model to deal with categorical predicted data and how to perform Bayesian classification using either logistic regression when we have two classes or softmax regression for more than two classes. We learned what an inverse link function is and how it is used to build **Generalized Linear Models (GLM)**, which extends the range of problems that can be solved by linear models. We also learned about some precautions we have to take, for example, when dealing with correlated variables, perfectly separable classes or unbalanced classes. While we focused on discriminative models for classification, we also learned about generative models and some of the main differences between both types of models.

# Keep reading

- *Doing Bayesian Data Analysis, Second Edition* by *John Kruschke*. Chapters 21 and 22.
- *Statistical Rethinking* by *Richard McElreath*. Chapter 10.
- *Bayesian Data Analysis, Third Edition* by *Andrew Gelman* and others. Chapter 16.

- *An Introduction to Statistical Learning* by *Gareth James* and others (second edition). Chapter 4.
- Check the PyMC3 example of logistic regression at `https://pymc-devs.github.io/pymc3/notebooks/GLM-logistic.html`. This example also includes model comparison techniques—a topic we will see in the next chapter.

# Exercises

1. Rerun the first model using the variables petal length and then petal width. 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 weakly informative prior. Try different value of $\nu$.
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 the previous chapter. How useful is a linear regression compared to the logistic regression? Can the result be interpreted as a probability? Hint: check if the values of $y$ are restricted to the *[0, 1]* interval.
4. 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?
5. In the example for dealing with unbalanced data, change `df = df[45:]` to `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?
6. Compare the likelihood of the logistic model versus the likelihood of the LDA model. Use the function `sample_ppc` to generate predicted data and compare the type of data you get for both cases. Be sure to understand the difference between the types of data the model predicts.

# 6
# Model Comparison

"*All models are wrong, but some are useful.*"

— *George Box*

We have already discussed the idea that models are wrong in the sense that they are just approximations used in an attempt to understand a problem through data and not a verbatim copy of the real world. While every model is wrong, not every model is equally wrong; some models will be worse than others at describing the same data. In the previous chapters, we focused our attention on the inference problem, that is, how to learn the value of parameters from the data. In this chapter, we are going to focus on a different problem: how to compare two or more models 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 the present chapter, we will explore the following topics:

- Occam's razor, simplicity and accuracy overfitting, and underfitting
- Regularizing priors
- Information criteria
- Bayes factors

# Occam's razor – simplicity and accuracy

Suppose we have two models for the same data/problem and both seem to explain the data equally as well. Which model should we choose? There is a guiding principle or heuristic known as **Occam's razor** that loosely states that 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 since simpler models are easier to understand than more complex models, and another justification is based on Bayesian statistics. Without getting into the details of these justifications, we are going to accept this criterion as a useful rule of thumb for the moment, 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. If you do not remember very well what the coefficient of determination is, go back to *Chapter 4*, *Understanding and Predicting Data with Linear Regression Models*. Continuing with our discussion, 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? Well, maybe; remember that we also said, in the previous paragraph, that we tend to prefer simpler models.

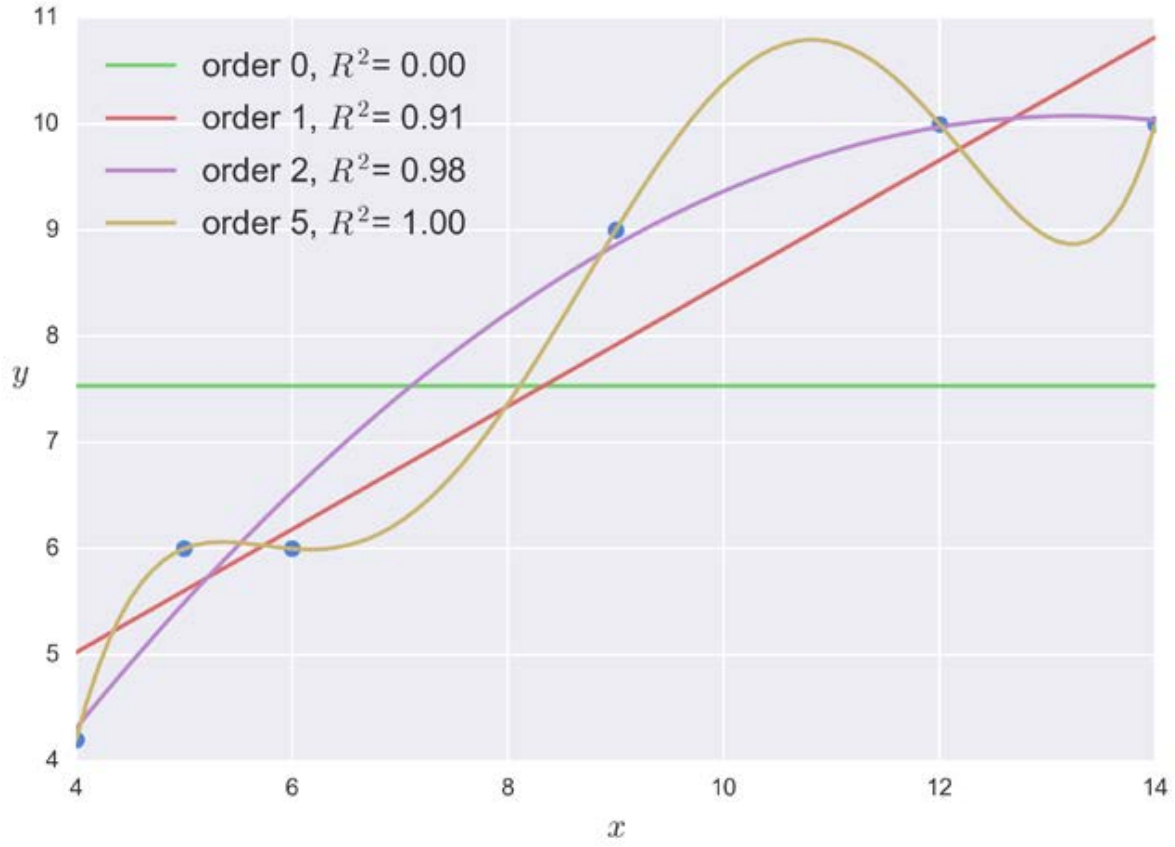
Intuitively, it seems that when comparing models, we tend to like those that have high accuracy and those that are simple. During the rest of this chapter, we are going to discuss this idea of balancing between these two features.

This chapter is more theoretical than previous chapters (even when we are just scratching the surface of this topic), so to make things easier, let us introduce an example that will help us move from this (correct) intuition of balancing accuracy versus complexity to a more theoretical (or at least empirical) grounded justification.

In this example, we are going to fit 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 shortcut:

```
x = np.array([4.,5.,6.,9.,12, 14.])
y = np.array([4.2, 6., 6., 9., 10, 10.])
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='order {}, $R^2$= {:.2f}'.format(i,
    r2))
plt.legend(loc=2, fontsize=14)
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$y$', fontsize=16, rotation=0)
```

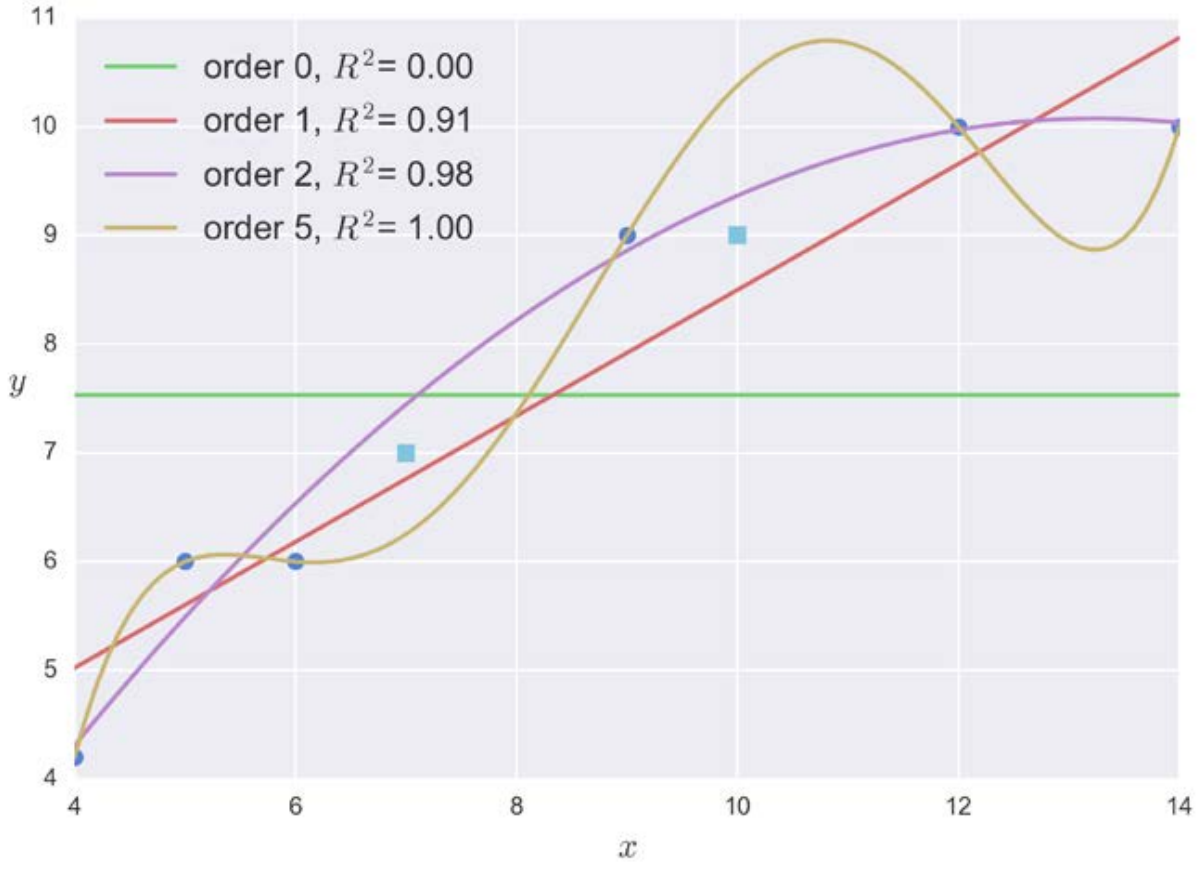



# Too many parameters leads to overfitting

From the preceding figure, we can see that increasing the complexity of the model is accompanied by an increasing accuracy 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 we briefly discussed this behavior of polynomials in *Chapter 4, Understanding and Predicting Data with Linear Regression Models* 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 degree 5 able to capture the data without missing a single data point? The reason is that we have the same number of parameters, 6, as the number of data points, 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! Another hint indicating that we are not really learning anything comes from the predictions of the model *between* the data points. The model predicts what seems to be a very weird behavior of the data. Notice that the best line according to the model of order 5 first goes up, then stays more or less at constant values (or may go slightly downward), then up again and finally down. Compare this wiggly behavior against the predictions of the model of order 1 or 2. They predict a line and a parabola respectively, together with some Gaussian noise. Intuitively, this seems to be a much more simple and plausible explanation than a line fitting every single data point but wiggling *between* data points. From this example, we can see that a model with higher accuracy is not always what we really want.

The next example will lead us to another important insight. 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 the following figure). How well does the order 5 model explain those points compared to the order 1 or 2 models? Not very well indeed. 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.



The lack of generalization is due to the fact that we have a very flexible model, one with so many parameters that it has overfitted the data. **Overfitting** is a general problem in statistics and machine learning and occurs when a model starts learning the noise in the data, effectively hiding the interesting pattern. We are of course assuming there is an interesting pattern in the first place. In general, a model with more parameters has more ways to accommodate the data, and hence it has a tendency to overfit the data. This is a practical concern with overly complex models and a practical justification for Occam's razor.

This example shows us that focusing only on how well the model explains the data used to fit the model can be misleading. The reason is that, at least in principle, we can always improve the accuracy by adding more parameters to the model. Let's introduce some vocabulary to help clarify this discussion. The accuracy measured with the data used to fit a model is referred as the **within-sample accuracy**. But a more informative and useful measure of the behavior of our model is the predictive accuracy of the model measured on data not used for fitting the model; this is usually referred to as **out-of-sample accuracy**.

# 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. In this model, all the beta parameters are set exactly to zero and hence we have reduced the linear model relating two variables to a simpler Gaussian model of the dependent variable alone. Notice that for the order 0 model, the independent variable does not matter and the model can only capture the average of the dependent variable. In other words, this model is saying the data can be explained by the mean of the dependent variable and some Gaussian noise. We say this model has underfitted the data; it is so simple that it is unable to capture the interesting pattern in the data, it can only capture a very simplified version of what is really going on. In general, a model with too few parameters will tend to underfit the data.

# 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. Like in a healthy diet, when modeling we have to keep a balance. 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. There are different ways of thinking about this idea of balance. For example, we may think that the purpose of fitting a model (or learning models) is to obtain a compressed representation of the data; we want to simplify the data in order to understand it and/or make predictions. If the model represents the data in a very compressed way, we lose important details and we end up getting very simple summaries such as a mean value; on the contrary, we get too much noise, or we can push things to the extreme and get the data encoded in a different way with no compression at all.

The overfitting/underfitting balance can also be discussed in terms of the bias-variance trade-off. Let me introduce this concept using an example. Suppose we have a model able to exactly pass through every point in our dataset, just like the order 5 model. Imagine that we refit the model with a new sample of six points and then with another and we keep doing this for several different sets of six points. Each time, we will obtain different curves that will accommodate the new six data points exactly. We could get a wiggly line, then a straight line, another time a parabola, and so on. Because the model can adapt to every single detail in the data our predictions will have a lot of variation and we would say the model has **high variance**. On the contrary, if we have a more restricted model, like a straight line, we have a more biased model in the sense that it will always try to accommodate a straight line. A model with **high bias** is a model with more *prejudices* (if you will excuse the anthropomorphization) or more *inertia*.

The order 1 model, from our example, has higher bias and lower variance than the order 2 model . The second can produce different curved lines including a straight line as a special case. In summary, we find that:

- **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 with 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 general, when we increase one of these terms we decrease the other and that is why people talk about the bias-variance trade-off. Once again, the main idea is that we want a balanced model.

# Regularizing priors

Using informative and weakly informative priors is a way of introducing bias in a model and, if done properly, can be a good thing because it helps to prevent overfitting.

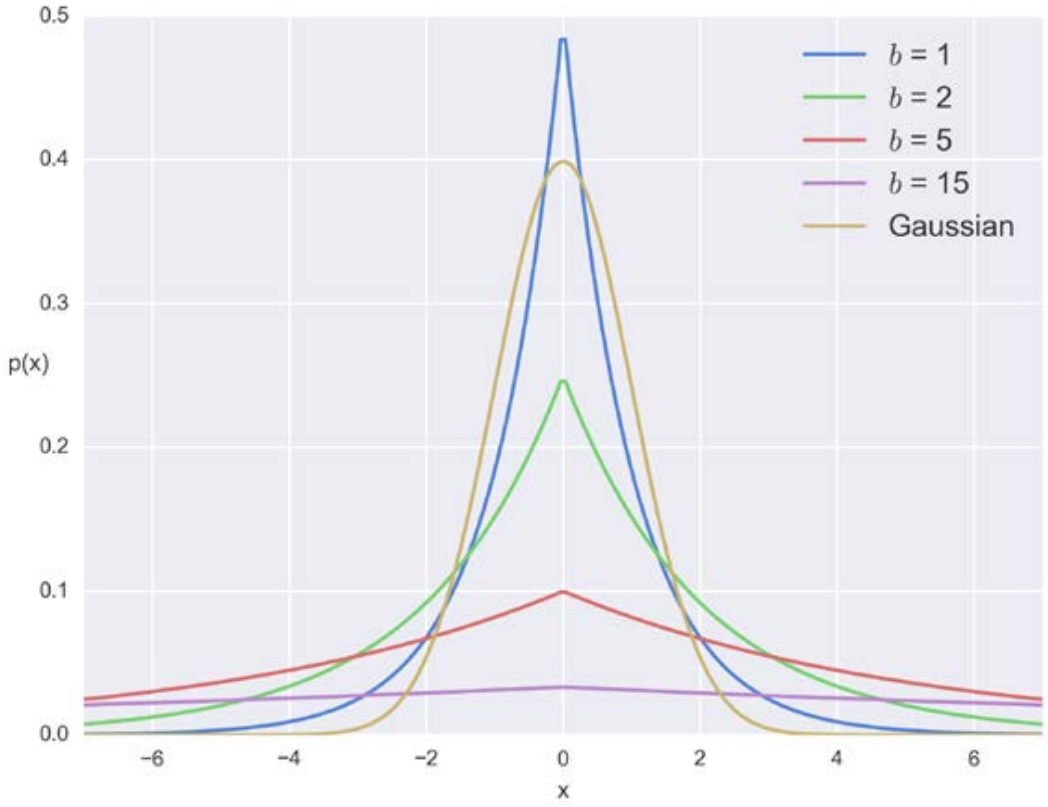
The regularization idea is so powerful and useful that it has been discovered several times, including outside the Bayesian framework. In some fields, this idea is known as the **Tikhonov regularization**. In non-Bayesian statistics, this regularization idea takes the form of two modifications on the 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 small standard deviation that pushes the coefficients towards zero, while the Lasso regression can be interpreted from a Bayesian point of view as using Laplace priors instead of Gaussian for the beta coefficients. The standard versions of ridge and lasso regressions corresponds to single point estimates; we do not get a posterior like when we perform a fully Bayesian analysis.

Before moving on, let us take a moment to discuss the Laplace distribution. This distribution is similar to the Gaussian distribution, but its first derivate is undefined at zero because it has a very sharp peak at zero (see the following figure). The Laplace distribution concentrates its probability mass much closer to zero compared to the normal distribution; the resulting effect when we use it as a prior is that it has the tendency to make some parameters zero. Thus it is said that lasso (or its Bayesian analog) can be used to regularize and also to perform variable selection (removing some terms or variables from a model).

The following code generates a figure showing the Laplace distribution centered on zero and with four different values for its scale parameter. A Gaussian with a mean zero and a standard deviation one is also shown for comparison.

```
plt.figure(figsize=(8, 6))
x_values = np.linspace(-10, 10, 300)
for df in [1, 2, 5, 15]:
    distri = stats.laplace(scale=df)
    x_pdf = distri.pdf(x_values)
    plt.plot(x_values, x_pdf, label='$b$ = {}'.format(df))

x_pdf = stats.norm.pdf(x_values)
```

```
plt.plot(x_values, x_pdf, label='Gaussian')
plt.xlabel('x')
plt.ylabel('p(x)', rotation=0)
plt.legend(loc=0, fontsize=14)
plt.xlim(-7, 7);
plt.savefig('B04958_06_03.png', dpi=300, figsize=[5.5, 5.5])
```




It is something very remarkable that these widely accepted ideas of regularization fit so naturally in the Bayesian paradigm. Some people may even say that because everyone agrees on regularization being a *honking great idea (and we should do more of those!)* everyone is at least a bit Bayesian, even when they do not know it or reject that label.

# Regularizing priors and hierarchical models

In the light of what we have been discussing, hierarchical models can also be thought of as a regularization method. Think of hierarchical models as a way of learning the prior from the data, by means of introducing hyper-priors, of course. So, in a sense, because we are learning the prior from the data, we are performing regularization and letting the data tell us the strength of the regularization. This is another and probably insightful way of thinking about hierarchical models and shrinkage. Think about how the concept of regularization can be used to interpret the result from *Chapter 4, Understanding and Predicting Data with Linear Regression Models*, when we use a hierarchical model to fit a line to a single data point.

# Predictive accuracy measures

In the previous example, it is more or less easy to see that the order 0 model is very simple and the order 5 model is too complex, but what about the other two? How we can distinguish between those options? We need a more principled way of taking into account the accuracy on one side and the simplicity on the other. Two methods to estimate the out-of-sample predictive accuracy using only the within-sample data 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

On average, the accuracy of a model will be higher for the within-sample than for the out-of-sample accuracy. As we need data to fit the model and data to test it, one simple solution is to split our data into two parts:

- The training set which we use to fit a model
- The test set which we use to measure how well the model fits the data

This is a very good solution if we have a lot of data; for example, crystallographers have been doing this to solve and validate the structures of molecules for decades. Otherwise, the aforementioned procedure will be detrimental since we are reducing the information available to fit the model and also to assess its accuracy.

To circumvent this problem of lack of data, a simple and in most cases effective solution is to do cross-validation. We take our data and we partition it into *K* portions, for example, five 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 (four portions in this example) and the remaining to validate it. Then we repeat this procedure systematically by leaving a different portion out of the training set and using that portion as the validation set until we have done *K* rounds. The results are then averaged along the *K* runs. This is known as **K-fold cross-validation**, where *K* is the number of partitions; in this example, we would say five-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.

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 can be beyond our possibilities. Many people have tried to come up with simpler-to-compute quantities that approximate the results obtained with cross-validation or that work in scenarios where cross-validation can be not that straightforward to perform. And that is the subject of the next section.

# Information criteria

Information criteria are a collection of different and somehow related tools used to compare models in terms of how well they fit the data while taking into account their complexity through a penalization term. In others words, information criteria formalize the intuition we developed at the beginning of the 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 out of the scope of this book, so we are going to limit ourselves to understand them from a practical point of view.

## The log-likelihood and the 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:

$$\frac{1}{n}\sum\left(y_i - E\left(y_i \mid \theta\right)\right)^2$$

$E\left(y_i \mid \theta\right)$ is just the predicted value given the estimated parameters. Notice this is essentially the average of the difference between 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.

This is a very simple-to-compute quantity and is useful under certain constraints, such as data being normally distributed. Instead, a more general measure is to compute the log-likelihood:

$$log\ p\left(y \mid \theta\right)$$

Under certain circumstances, this turns out to be proportional to the quadratic mean error, for example, in the simple linear regression case.

In practice and for historical reasons, people usually do not use the log-likelihood directly; instead, they use a quantity known as **deviance**, which is:

$$-2log\ p(y \mid \theta)$$

The deviance is used for Bayesians and non-Bayesians alike; the difference is that under a Bayesian framework, $\theta$ is estimated from the posterior and, like any quantity derived from a posterior, it has a distribution. On the contrary, in non-Bayesian settings, $\theta$ is a point estimate. To learn how to use the 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. So 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 in common the use of the deviance and a penalization term. And 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:

$$AIC = -2log\ p\left(y \mid \hat{\theta}_{mle}\right) + 2p_{AIC}$$

Here, $p_{AIC}$ is the number of parameters and $\hat{\theta}_{mle}$ is the maximum likelihood estimation of $\theta$. Maximum likelihood estimation is common practice for non-Bayesians and, in general, is equivalent to the Bayesian **maximum a posteriori** (**MAP**) estimation when using flat priors. Notice that the $\hat{\theta}_{mle}$ is a point estimation and not a distribution.

Alternatively, we can represent the previous formula as:

$$AIC = -2\left(\log p\left(y \mid \hat{\theta}_{mle}\right) - p_{AIC}\right)$$

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 we should choose the one with the fewer parameters.

AIC works for non-Bayesian approaches but is problematic for Bayesian ones. 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 a model using non-flat priors. When using non-flat priors, we cannot simply count the number of parameters in the model. Properly used non-flat priors have the property for regularizing the model, that is they reduce the tendency to overfit. It is the same as saying that the effective number of parameters of a model with regularizing priors is lower than the real number of parameters. Something similar occurs when we have a hierarchical model; after all, hierarchical models can be thought of as effective ways to learn the strength of the prior from the data (without cheating).

# Deviance information criterion

A way to get a Bayesian version of AIC is to include information from the posterior and also estimate the number of parameters from the model and the data. This is exactly what the **deviance information criterion** (**DIC**) does:

$$-2 \times log\ p\left(y \mid \hat{\theta}_{post}\right) + 2p_{dic}$$

We can see that DIC and AIC are analogous, the difference being that now we are computing the deviance from $\hat{\theta}_{post}$, that is, the posterior mean of $\theta$, and we are using $p_{dic}$ as the effective number of parameters. $p_{dic}$ is obtained as:

$$\hat{D}(\theta) - D\left(\hat{\theta}\right)$$

That is, the mean deviance minus the deviance at the mean. If we have a peaked posterior, that is, a posterior for $\theta$ concentrated around $\hat{\theta}$, both terms in the preceding equation will be similar and $p_{dic}$ will be small. Instead, if we have a broad posterior, we will have more values of $\theta$ away from $\hat{\theta}$ and hence $\hat{D}(\theta)$ will be large and $p_{dic}$ will be larger.

So far so good; we can say that DIC is a more Bayesian version of AIC. Nevertheless, the story does not end here. DIC does not use the whole posterior and the way the effective number of parameters are computed for DIC has some problems for weak priors. Alternatives for this second problem have been suggested. Nevertheless, this is the one implemented in PyMC3 and is enough for our current discussion.

# Widely available information criterion

This is similar to DIC but is more Bayesian because it uses the whole posterior distribution. Like with AIC and DIC, we can see that the **widely available information criterion** (**WAIC**) has two terms, one that measures how well the data fits the model and one penalizing complex models:

$$WAIC = 2lppd + 2\,p_{waic}$$

Here, *lppd* is the log point-wise predictive density and can be approximated as:

$$lppd = \sum \log\left(\frac{1}{S}\sum p\left(y_i \mid \theta^s\right)\right)$$

First we compute the average of the likelihood along *S* samples from the posterior. We do that for every data point and then we sum over the *N* data points. And the effective number of parameters is computed as:

$$p_{waic} = \sum v_{s=1}^{s}\left(log\ p\left(y_i \mid \theta^s\right)\right)$$

That is, we compute the variance of the log-likelihood along *S* samples from the posterior. We do that for every data point and then we sum over the *N* data points. The intuition for this way of computing the effective number of parameters is similar to the one used for DIC. A more flexible model will tend to result in a more broad or diffusive posterior, as we already discussed at the end of the AIC section.

# Pareto smoothed importance sampling leave-one-out cross-validation

This is a method to approximate the LOOCV results but without actually performing LOOCV. Without going into too much detail, the main idea is that is possible to approximate LOOCV by re-weighting the likelihoods appropriately. This can be done using a technique known as importance sampling. The problem is that the results are instable. 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 LOOCV. The interpretation is similar to the other measures we have seen; the lower the value, the higher the estimated predictive accuracy of the model.

# Bayesian information criterion

Just like the logistic regression and my mother's sopa seca, this name can be misleading. The **Bayesian information criterion** (**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 like AIC. It also assumes flat priors and uses a maximum likelihood estimate.

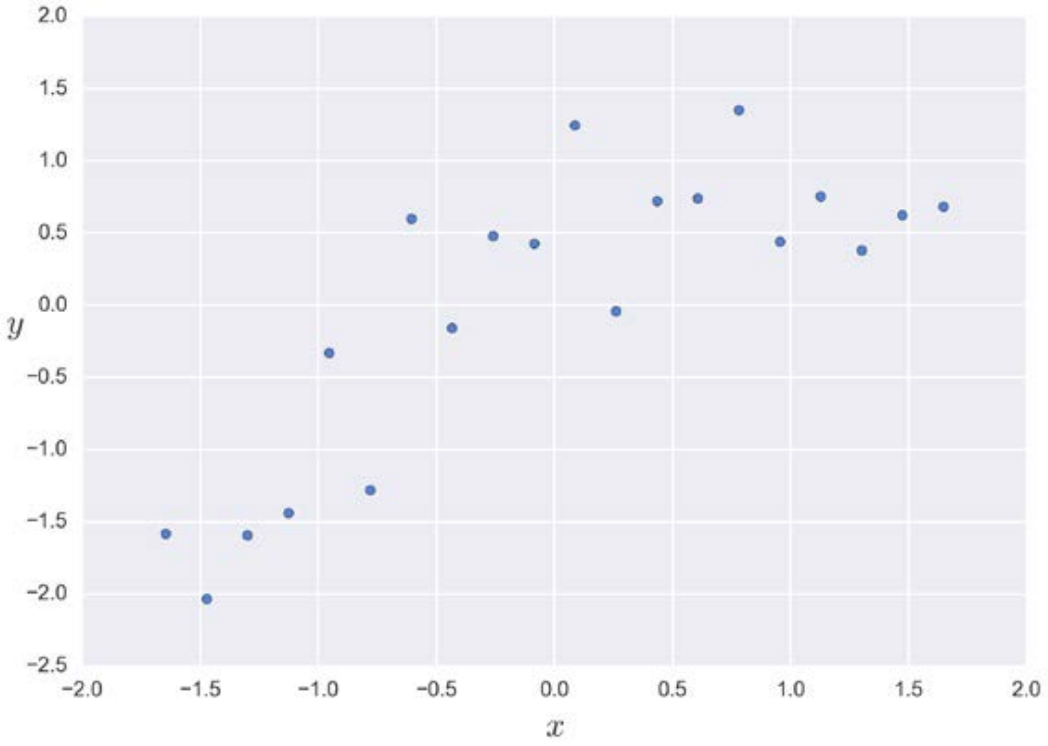
Even more importantly, BIC is different to the other information criteria we have seen and is more related to the Bayes factors that we will discuss later in this chapter. For the previous reasons and following Gelman's recommendations, we are not going to further discuss or use BIC.

# Computing information criteria with PyMC3

Information criteria can be easily computed with PyMC3. We just need to call a function. To exemplify their use, we are going to build a simple example. First we define some data and we standardize it:

```
real_alpha = 4.25
real_beta = [8.7, -1.2]
data_size = 20
noise = np.random.normal(0, 2, size=data_size)
x_1 = np.linspace(0, 5, data_size)
y_1 = real_alpha + real_beta[0] * x_1 + real_beta[1] * x_1**2 +
  noise
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('$x$', fontsize=14)
plt.ylabel('$y$', fontsize=14, rotation=0);
```




As we can see from the code, and a little bit less clearly from the plot, we have data that can be fitted using a polynomial of order 2. But suppose we have reasons to think a simple linear regression is also a good model. We are going to fit both models and then use the information criteria to compare them. We are going to start with the linear one:

```
with pm.Model() as model_l:
    alpha = pm.Normal('alpha', mu=0, sd=10)
    beta = pm.Normal('beta', mu=0, sd=10)
    epsilon = pm.HalfCauchy('epsilon', 5)
    mu = alpha + beta * x_1s[0]
    y_pred = pm.Normal('y_pred', mu=mu, sd=epsilon, observed=y_1s)
    trace_l = pm.sample(2000)
chain_l = trace_l[100:]
```

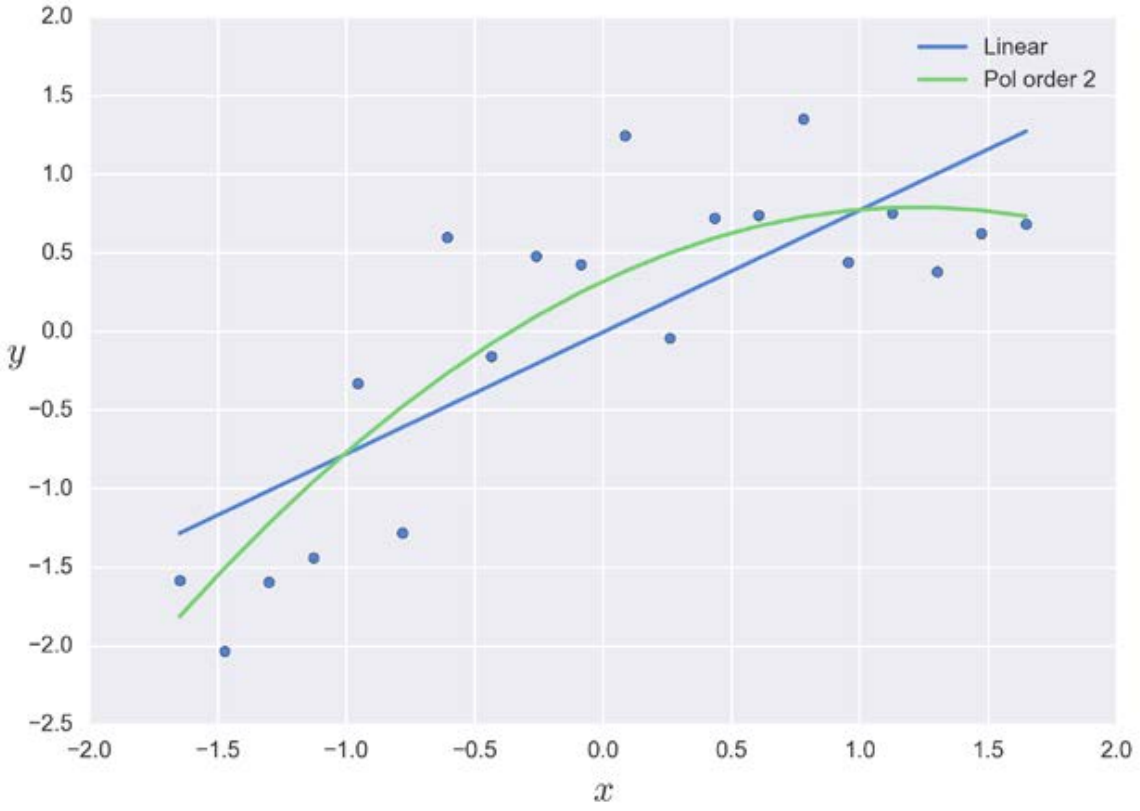
To save space, we are going to omit the `traceplot` and other plot tests. And we are going to continue with the second order polynomial model. You should not omit those tests and plots:

```
with pm.Model() as model_p:
    alpha = pm.Normal('alpha', mu=0, sd=10)
    beta = pm.Normal('beta', mu=0, sd=10, shape=x_1s.shape[0])
    epsilon = pm.HalfCauchy(''epsilon', 5)
```

```
    mu = alpha + pm.math.dot(beta, x_1s)
    y_pred = pm.Normal('y_pred', mu=mu, sd=epsilon, observed=y_1s)
    trace_p = pm.sample(1000)
chain_p = trace_p[100:]
```

Now we are going to plot the results as the best-fitted lines, using the following code:

```
alpha_l_post = chain_l['alpha'].mean()
betas_l_post = chain_l['beta'].mean(axis=0)
idx = np.argsort(x_1s[0])
y_l_post = alpha_l_post + betas_l_post * x_1s[0]
plt.plot(x_1s[0][idx], y_l_post[idx], label='Linear')
alpha_p_post = chain_p['alpha'].mean()
betas_p_post = chain_p['beta'].mean(axis=0)
y_p_post = alpha_p_post + np.dot(betas_p_post, x_1s)
plt.plot(x_1s[0][idx], y_p_post[idx], label='Pol order
  {}'.format(order))
plt.scatter(x_1s[0], y_1s)
plt.legend()
```

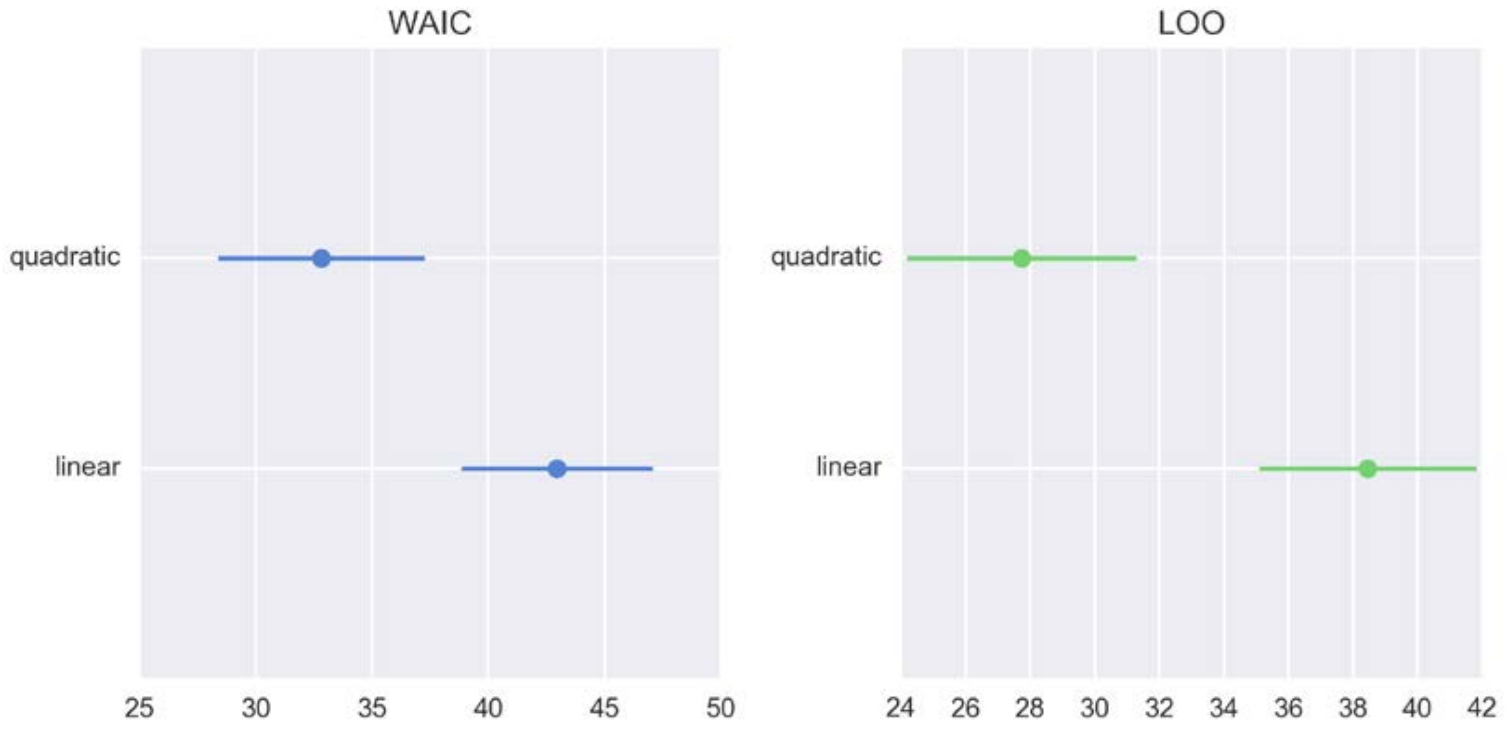



To get the value of DIC using PyMC3, we need to call a function with a `trace` as an argument; the model will be guessed from the context if we are inside a proper `with` statement, or we can just pass the model as an additional argument:

```
pm.dic(trace=trace_l, model=model_l)
```

The same goes for the calculation of WAIC using the function `pm.waic()` and for LOO using the function `pm.loo()`. For WAIC and LOO PyMC3 reports a point estimate together with an estimation of their respective standard errors. We can use a standard error to assess the uncertainty of the WAIC (or LOO) estimates. Nevertheless, caution needs to be taken because the estimation of the standard error assumes normality and hence it may not be very reliable when the sample size is low.

```
plt.figure(figsize=(8, 4))
plt.subplot(121)
for idx, ic in enumerate((waic_l, waic_p)):
    plt.errorbar(ic[0], idx, xerr=ic[1], fmt='bo')
plt.title('WAIC')
plt.yticks([0, 1], ['linear', 'cuadratic'])
plt.ylim(-1, 2)

plt.subplot(122)
for idx, ic in enumerate((loo_l, loo_p)):
    plt.errorbar(ic[0], idx, xerr=ic[1], fmt='go')
plt.title('LOO')
plt.yticks([0, 1], ['linear', 'cuadratic'])
plt.ylim(-1, 2)
plt.tight_layout()
```



# 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 indicating 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 you have enough samples, that you have a well-mixed chain, and that you have used a proper burn-in value. 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 think about using another method such as directly performing K-fold cross-validation.

# Interpreting and using information criteria measures

The simpler way to use information criteria is to perform **model selection**. Simply choose the model with the lower **Information Criterion (IC)** value and forget about the other models. Thus we can interpret these plots as saying that the two methods agree that the best model is the order 2 model.

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 of what we really know.

One possible alternative is to perform a model selection but report and discuss the different models together with the computed information criteria values and also posterior predictive checks. It is important to put all 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 our methods. 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 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 this formula:

$$w_i = \frac{\exp\left(-1/2 dIC_i\right)}{\sum_j^M \exp\left(-1/2 dIC_j\right)}$$

Where $dIC_i$ is the difference between the i-esim information criterion value and the lowest one.

We can use any information criterion we want to compute a set of weights, but, of course, we cannot mix them. This formula is a heuristic way to compute the relative probability of each model (given a fixed set of models) from the information criteria values. Look how the denominator is just a normalization term to ensure that the weights sum up to one.

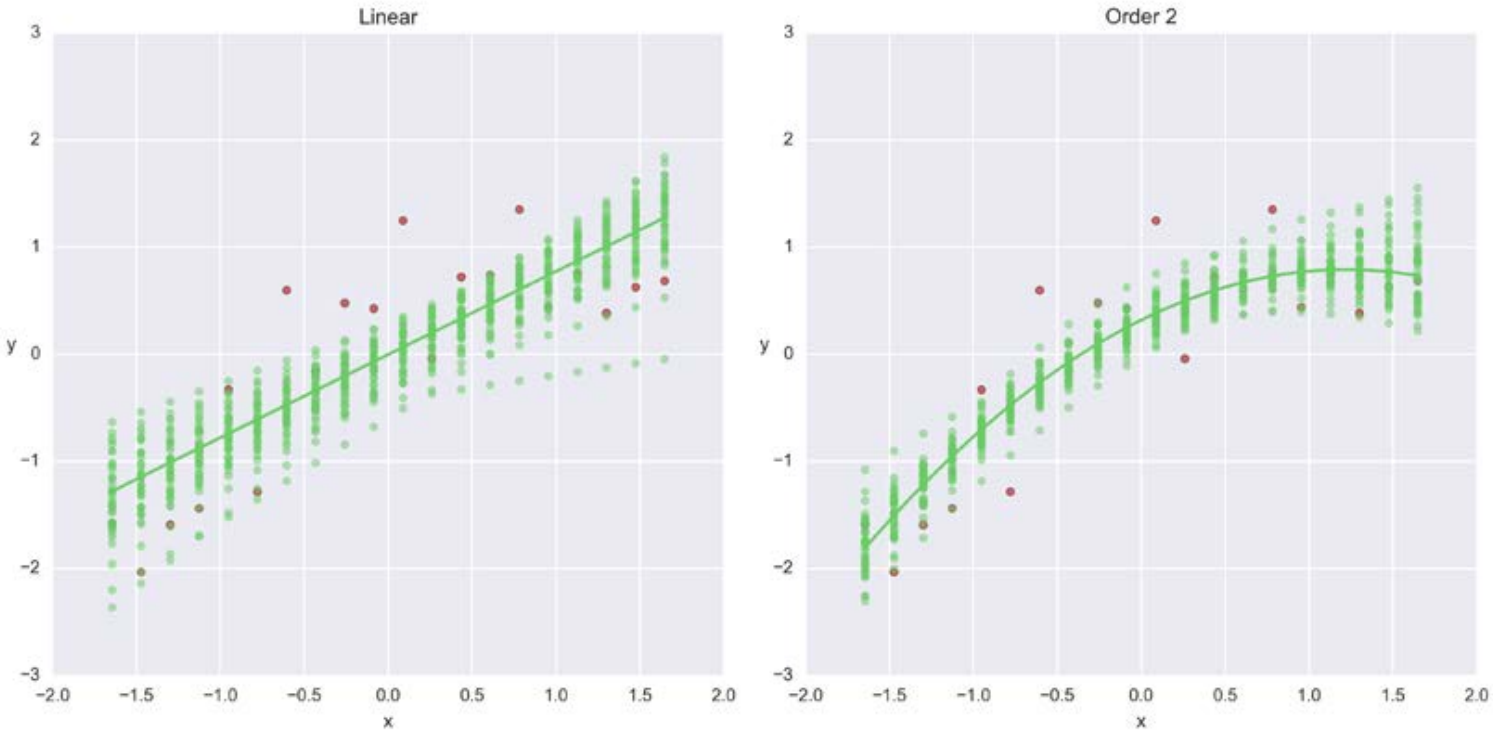
There are other ways to average models such as, for example, explicitly building a super-model that includes all the models we have. We then perform parameter inference while jumping between the models. Later, in the Bayes factor section, we will discuss a form of this.

Besides averaging discrete models we can sometimes think of continuous versions of them. A toy example is to imagine we have the coin-flipping problem and we have two different models, one with a prior biased towards heads and one towards tails. We could fit both separate models and then average them using, for example, IC-derived weights. As an alternative, we could build a hierarchical model to estimate the prior distribution, and notice that instead of contemplating two discrete models we will be computing a continuous model that includes these two both discrete models as particular cases. Which approach is better? Once again, that depends on our concrete problem. Do we have good reasons to think about two discrete models, or is our problem better represented with a continuous model?

# Posterior predictive checks

In past chapters, we introduced the concept of posterior predictive checks as a way to evaluate how well a model explains the same data used to fit the model. This was a kind of consistency check, we said. The purpose of posterior predictive checks was not to say that the model is wrong; we already know that. The idea was to try to understand which parts of the data do not get very well modeled in order to have a better grasp of the limitations of a model or to try to improve it. We will revisit this topic here, as a reminder that posterior predictive checks can be used to compare models and get a better grasp of how they are different:

```
plt.subplot(121)
plt.scatter(x_1s[0], y_1s, c='r');
plt.ylim(-3, 3)
plt.xlabel('x')
plt.ylabel('y', rotation=0)
plt.title('Linear')
for i in range(0, len(chain_l['alpha']), 50):
    plt.scatter(x_1s[0], chain_l['alpha'][i] + chain_l['beta']
[i]*x_1s[0], c='g', edgecolors='g', alpha=0.05);
plt.plot(x_1s[0], chain_l['alpha'].mean() + chain_l['beta'].
mean()*x_1s[0], c='g', alpha=1)

plt.subplot(122)
plt.scatter(x_1s[0], y_1s, c='r');
plt.ylim(-3, 3)
plt.xlabel('x')
plt.ylabel('y', rotation=0)
plt.title('Order {}'.format(order))
for i in range(0, len(chain_p['alpha']), 50):
    plt.scatter(x_1s[0], chain_p['alpha'][i] + np.dot(chain_p['beta']
[i], x_1s), c='g', edgecolors='g', alpha=0.1)
idx = np.argsort(x_1)
plt.plot(x_1s[0][idx], alpha_p_post + np.dot(betas_p_post,
  x_1s)[idx], c='g', alpha=1)
```




# Bayes factors

A common alternative to evaluate and compare models in the Bayesian world (at least in some of its countries) are the Bayes factors.

One problem with Bayes factors is that their computation can be highly sensitive to aspects of the priors that have no practical effect on the posterior distribution of individual models. You may have noticed in 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 generally affected by these kind of changes in the model. Another problem with Bayes factors is that their computations can be more difficult than inference. One final criticism is that Bayes factors can be used as a Bayesian way of doing hypothesis testing; there is nothing wrong in this per se, but many authors have pointed out that an inference or modeling approach, similar to the one used in this book, is better suited to most problems than the generally taught approach of hypothesis testing (whether Bayesian or not Bayesian). Having said all this, let's see what Bayes factors are and how to compute them. To understand what Bayes factors are, let us write Bayes theorem one more time (we have not done so for a while!):

$$p(\theta \mid y) = \frac{p(y \mid \theta)\, p(\theta)}{p(y)}$$

Where, $y$ represents the data and $\theta$ the parameters. Alternatively, we could write it like this:

$$p(\theta \mid y, M) = \frac{p(y \mid \theta, M)\, p(\theta \mid M)}{p(y \mid M)}$$

The only difference is that now we are making explicit the dependency of the inference on a given model *M*. The term in the denominator is known as the evidence or marginal likelihood, as you may remember from the first chapter. So far, we have omitted this term from our computations, thanks to our inference engines such as Metropolis and NUTS. We can write the evidence as:

$$p(y, M) = \int p(y \mid \theta, M)\, p(\theta \mid M)\, d\theta_m$$

That is, to compute the evidence $p(y \mid M)$, we need to marginalize (by summation or integration) over all the possible values of $p(\theta \mid M)$, that is, over all the prior values of $\theta$ for a given model.

The quantity $p(y \mid M)$ does not tell us too much on its own; like with the information criteria, what matters are the relative values. So when we want to compare two models, we take the ratio of their evidence, and that is a Bayes factor:

$$BF = \frac{p(y \mid M_0)}{p(y \mid M_1)}$$

Then when $BF > 1$, model 0 explains data better than model 1.

Some authors have proposed tables with ranges to ease BF interpretation like the following, indicating how much evidence favors 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. But 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 necessary to make a claim is not the same in particle physics, or a court, or to evacuate a town to prevent hundreds of deaths.

# Analogy with information criteria

Notice that if we take the logarithm of Bayes factors, we can turn the ratio of the marginal likelihood into a difference, and comparing differences in marginal likelihoods is similar to comparing differences in information criteria. But where is the fitting term and where the penalizing term? Well, 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 diffuse the prior and hence the greater the penalty when computing the evidence. This is the reason people say the Bayesian theorem leads to a natural penalization of complex models or that the Bayes theorem comes with a built-in Occam's razor.

# Computing Bayes factors

The computation of Bayes factors can be framed as a hierarchical model, where the high-level parameter is an index 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 $p(M_x \mid y)$. To compute the Bayes factors, we do:

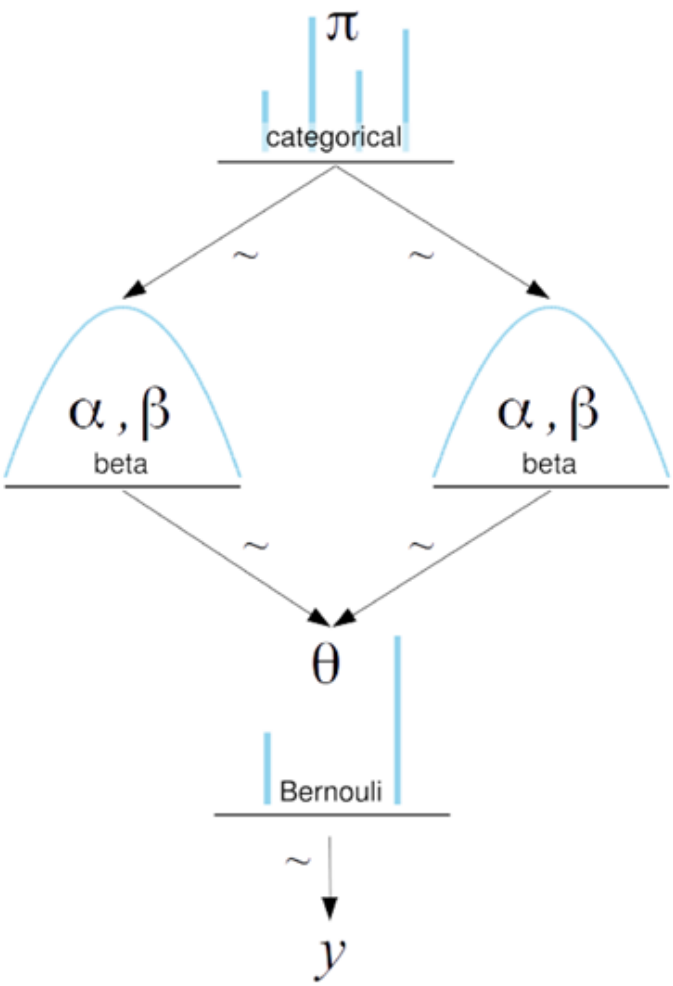
$$\frac{p(y \mid M_0)}{p(y \mid M_1)} = \frac{p(M_0 \mid y)\, p(M_1)}{p(M_1 \mid y)\, p(M_0)}$$

The first term to the right of the equality is known as the posterior odds and the second is the prior odds. Remember, we already talked about the definition of odds in *Chapter 5*, *Classifying Outcomes with Logistic Regression*. If you are wondering where this equation comes from, it is just the consequence of writing the Bayes theorem for the ratio of $p(y \mid M_0)$ and $p(y \mid M_1)$.

To exemplify the computation of the Bayes factors, we are going to flip coins one more time:

```
coins = 30
heads =  9
y = np.repeat([0, 1], [coins-heads, heads])
```

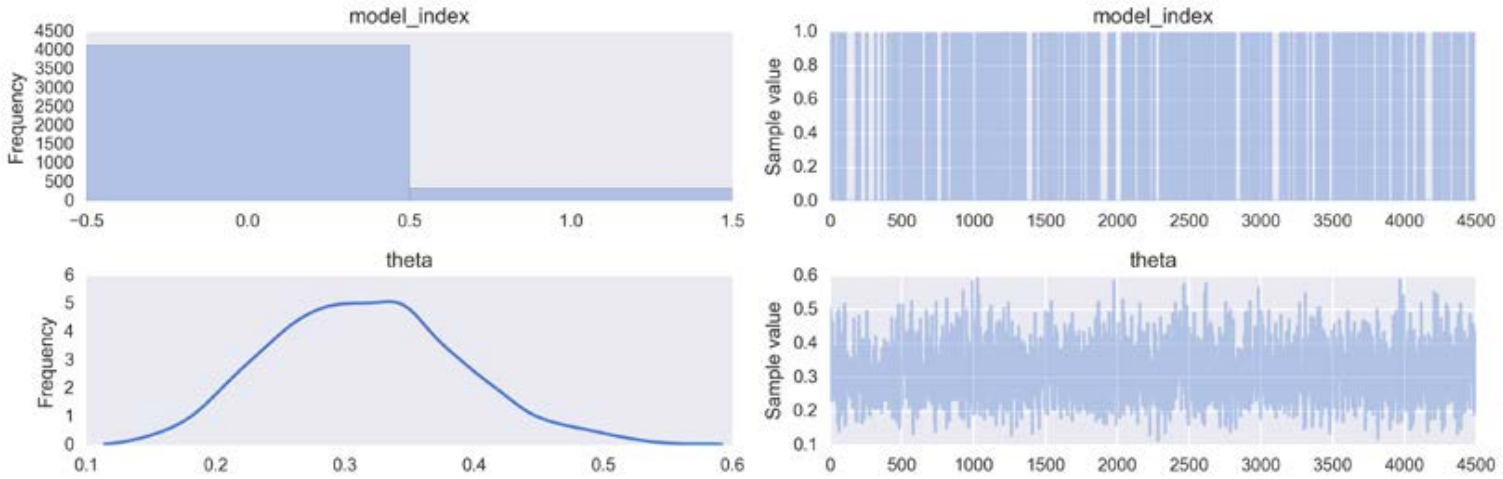
A Kruschke diagram of the model we are going to use is illustrated as follows. In this example, we are choosing between two beta priors: one biased towards 1 and the other towards 0:




Notice that now while we are computing Bayes factors between models that differ only on the prior, the models could differ on the likelihood or even both. The idea is the same.

Now the PyMC3 model. To switch between priors, we are using the `pm.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 we are also using the `pm.math.eq()` function to check if 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.switch(pm.math.eq(model_index, 0), m_0, m_1)

    theta = pm.Beta('theta', m[0], m[1])
    y = pm.Bernoulli('y', theta, observed=y)
    trace_BF = pm.sample(5000)
chain_BF = trace_BF[500:]
pm.traceplot(chain_BF)
```




And now we compute the Bayes factor by counting the variable `model_index`. Notice that we have included the values of the priors for each model:

```
pM1 = chain_BF['model_index'].mean()
pM0 = 1 - pM1
BF = (pM0/pM1)*(p[1]/p[0])
```

As a result, we get a value of ~*11*, that is, according to the Bayes factor we compute, model 0 is favored over model 1. This makes total sense since the data has fewer values of heads than expected for $\theta = 5$ 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 model 1 is more compatible with $\theta > 0.5$ (more heads than tails).

## Common problems 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. And this could be problematic because we can under-sample 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, 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 find these problems, we can do 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`

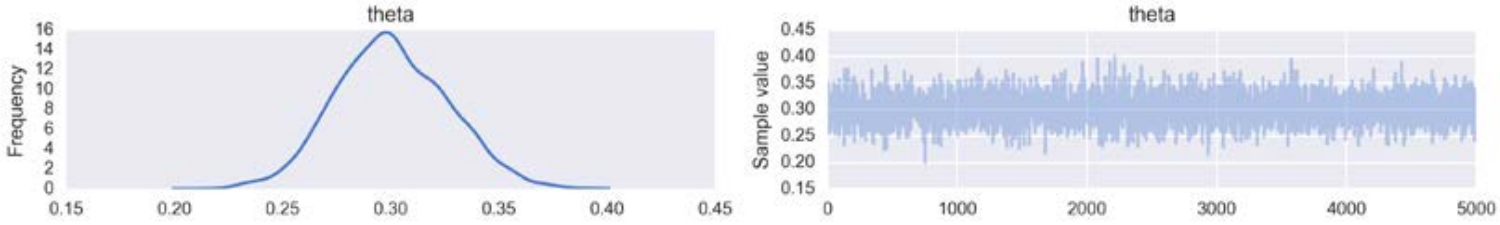
# Bayes factors and information criteria

We have already said that Bayes factors are more sensitive to priors than many people like. It is like having differences that are practically irrelevant when doing inference but that turn out to be important when computing Bayes factors. And this is one of the reasons many Bayesians do not like Bayes factors.

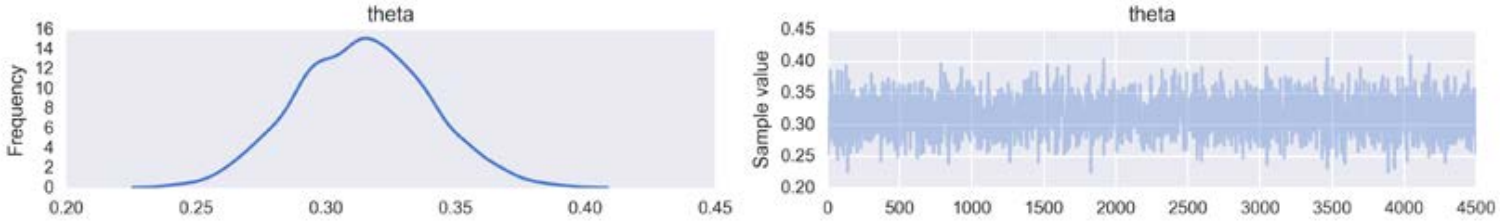
Now we are going to see an example that will help clarify what Bayes factors are doing and what information criteria are doing. Go back to the definition of the data for the coin flip example and now set *300* coins and *90* heads; this is the same proportion as before but we have *10* times more data. Then run each model separately:

```
with pm.Model() as model_BF_0:
    theta = pm.Beta('theta', 4, 8)
```

```
    y = pm.Bernoulli('y', theta, observed=y)

    trace_BF_0 = pm.sample(5000)
chain_BF_0 = trace_BF_0[500:]
pm.traceplot(trace_BF_0);
```



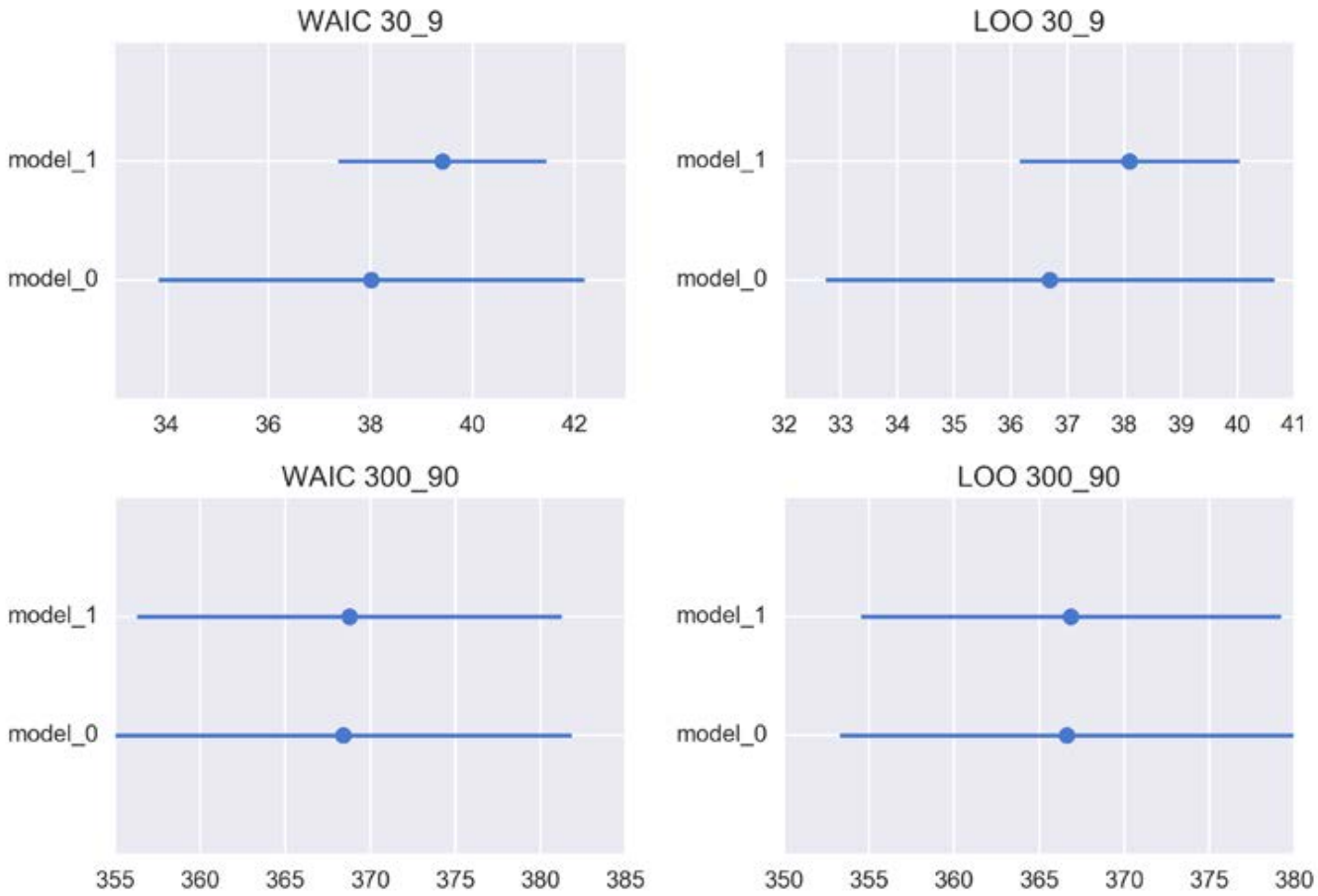

```
with pm.Model() as model_BF_1:
    theta = pm.Beta('theta', 8, 4)
    y = pm.Bernoulli('y', theta, observed=y)
    trace_BF_1 = pm.sample(5000)
pm.traceplot(trace_BF_1);
```




If you check the posterior, you will see that both models make similar predictions, even when having different priors; the reason is that we have so much data that the effect of the prior, while still there, has been reduced. So now we are going to compute the Bayes factor; we get a value of ~*25* (of course you can compute it by yourself). The Bayes factor is saying that model 0 is favored over model 1 even more than when we had **30** coins and **9** heads. We increase the data and the decision between models becomes more clear. This makes total sense because now we are more certain that model 1 has a prior that does not agree with the data. But notice that as we increase the amount of data, both models will tend to agree on the value of $\theta$; in fact, we get ~*0.3* with both models. And hence if we decide to use $\theta$ to predict new outcomes, it will hardly make any difference if we choose the value of $\theta$ estimated with one model or the value estimated with the other model. So in this example, the Bayes factor is telling us that one model is better than the other, so in some sense it is helping us to detect something like the true model, while the predictions made with the parameters estimated from both models are more or less the same.

Compare now what WAIC and LOO are telling us (see the following figure); WAIC is ~*368.4* for model zero and ~*368.6* for model one and LOO is ~*366.4* and ~*366.7*. This intuitively sounds like a small difference. And what is more important than the actual difference is that if you compute the information criteria again for the data, **30** coins and **9** heads, you will get something like ~*38.1* for model zero and ~*39.4* for model one with WAIC and ~*36.6* for model zero and ~*38.0* for model one with LOO. That is, the relative difference when increasing the data becomes smaller, the more similar the estimation of $\theta$, the most similar the values for the predictive accuracy estimated by the information criteria. This example should help to clarify the differences between Bayes factors and information criteria.

The following figure shows the values for WAIC and LOO and their standard errors for the coin-flipping example using **30** tosses and **9** heads for the first row and **300** tosses and **90** heads for the second row.



# Summary

We began this chapter with the intuition that a good model is one that effectively explains the data and is also simple. Using this intuition, we discussed the problem of overfitting and underfitting that pervades statistical and machine learning practices. We then formalized our intuitions by introducing the concept of deviance and information criteria. We started with the rather unsophisticated AIC, and its more Bayesian cousin known as DIC. Then we learned about an improved version of both, WAIC. We also discussed briefly the empirical cross-validation method and a way to approximate its results using the LOO method. We briefly discussed priors and hierarchical models in the light of the new ideas exposed in this chapter. Finally, we ended with a discussion of Bayes factors, how to compute them, and how to solve some usual sampling problems associated with them. We finished with an example to clarify the different aims of Bayes factors and information criteria.

# Keep reading

- Chapter 6, *Statistical Rethinking*, *Richard McElreath*.
- Chapter 10, *Doing Bayesian Data Analysis*, *Second Edition*, *John Kruschke*.
- Chapter 7, *Bayesian Data Analysis*, *Third Edition*, *Gelman* and others.
- A nice post by *Jake VanderPlas* about model selection (part of a series): `http://jakevdp.github.io/blog/2015/08/07/frequentism-and-bayesianism-5-model-selection/`
- Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC: `http://arxiv.org/abs/1507.04544`.

# 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 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 it also with `sd=np.array([10, 0.1, 0.1, 0.1, 0.1])`.
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_ppc()` to re-run 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. 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 get in the first chapter.
7. Repeat the last example where we compare Bayes factors and IC but now reducing the sample size.

# 7
# Mixture Models

One common approach to model building is about combining or mixing simpler models to obtain more complex ones. In statistic, this type of models are generically known as mixture models. Mixture models are used for different purposes such as directly modeling subpopulations or as a useful trick to handle complicated distributions that cannot be described with simpler distributions. In this chapter we are going to learn how to build them. We are are also going to find that some models from previous chapters were mixture models in disguise, now we are going to uncover them using a mixture-model perspective.

In this chapter, we will learn:

- Finite mixture models,
- Zero-Inflated Poisson distribution
- Zero-Inflated Poisson regression
- Robust logistic regression
- Model-based clustering
- Continuous mixture models

# Mixture models

Sometimes a process or phenomenon under study cannot be properly described using a single distribution like a Gaussian or a binomial, or any other canonical/pure distribution, but it can be described as a mixture of such distributions. Models that assume the data comes from a mixture of distributions are know as **mixture models**.

One kind of situation where mixture models arise naturally is when we have a dataset that is better described as a combination of real subpopulations. For example, it makes perfect sense to describe the distribution of heights, in an adult human population, as a mixture of female and male subpopulations. Even more, if we have to deal also with non-adults, we may find it useful to include a third group describing children, probably without needing to make a gender distinction inside this group. Another classical example of a mixture model approach is used to describe a group of handwritten digits. In this case, it also makes perfect sense to use *10* subpopulation to model the data, at least in a base 10 system!

In other cases, we may choose to use a mixture model out of mathematical/ computational convenience and not because we are really trying to describe subpopulations in the data. 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 multi-modal or skewed distribution? Can we use Gaussian distributions to model them? We can, if we use a mixture of Gaussians. In this **Gaussian mixture model**, each component will be a Gaussian with a different mean and the same or 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, no matter how complex or weird they are, 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. This idea taken to an extreme (and performed in a non-Bayesian way) gives the **kernel density estimation** (**KDE**) technique we have being using to plot data (instead of using histograms). This non-Bayesian method puts a distribution (the SciPy implementation uses Gaussians) at each data point and then it sums all the individual Gaussians to approximate the empirical distribution of the data. A KDE is a non-parametric method. We will discuss non-parametric methods in *Chapter 8*, *Gaussian Processes*. For now, what we care about is that we have already seen examples of mixing Gaussians to approximate arbitrary distributions.

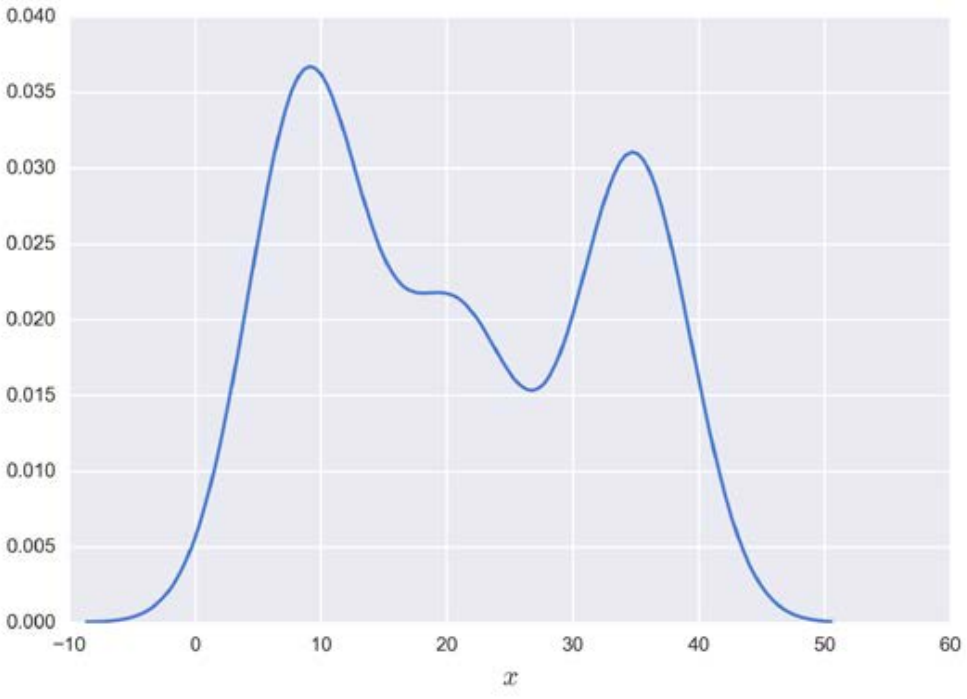
Whether we really believe in subpopulations or we use them for mathematical convenience (or even something in the middle), mixture models are a useful way to add flexibility to our models by using a mixture of distributions to describe the data.

# How to build mixture models

The general idea when building a finite mixture model is that we have a certain number of subpopulations, each one represented by some distribution, and we have data points that belong to those distribution but we do not know to which distribution each point belongs. Thus we need to assign the points properly. We can do that by building a hierarchical model. At the top level of the model, we have a random variable, often referred as a **latent variable**, which is a variable that is not really observable. The function of this latent variable is to specify to which component distribution a particular observation is assigned to. That is, the latent variable *decides* which component distribution we are going to use to model a given data point. In the literature, people often use the letter $z$ to indicate latent variables.

Let us start building mixture models with a very simple example. We have a dataset that we want to describe as being composed of three Gaussians.

```
clusters = 3
n_cluster = [90, 50, 75]
n_total = sum(n_cluster)
means = [9, 21, 35]
std_devs = [2, 2, 2]

mix = np.random.normal(np.repeat(means, n_cluster),
  np.repeat(std_devs, n_cluster))



sns.kdeplot(np.array(mix))
plt.xlabel('$x$', fontsize=14)
```



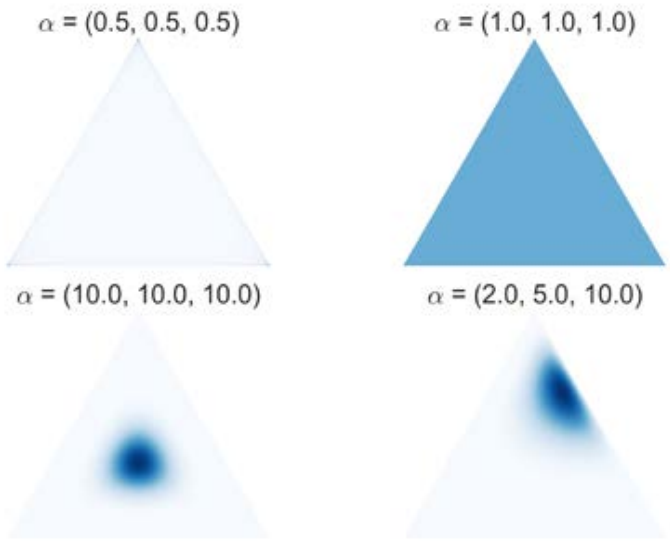
In many real situations, when we wish to build models, it is often more easy, effective and productive to begin with simpler models and then add complexity, even if we know from the beginning that we need something more complex. This approach has several advantages, such as getting familiar with the data and problem, developing intuition, and avoiding choking us with complex models/codes that are difficult to debug.

So, we are going to begin by supposing that we know that our data can be described using three Gaussians (or in general, k-Gaussians), maybe because we have enough previous experimental or theoretical knowledge to reasonably assume this, or maybe we come to that conclusion by eyeballing the data. We are also going to assume we know the mean and standard deviation of each Gaussian.

Given this assumptions the problem is reduced to assigning each point to one of the three possible known Gaussians. There are many methods to solve this task. We of course are going to take the Bayesian track and we are going to build a probabilistic model.

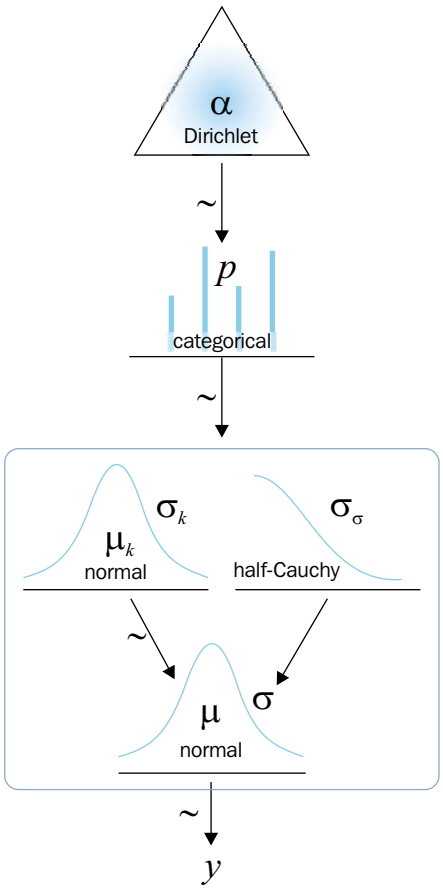
To develop our model, we can get ideas from the coin-flipping problem. Remember that we have had two possible outcomes and we used the Bernoulli distribution to describe them. Since we did not know the probability of getting heads or tails, we use a beta prior distribution. Our current problem with the Gaussians mixtures is similar, except that we now have k-Gaussian outcomes.

The generalization of the Bernoulli distribution to k-outcomes is the categorical distribution and the generalization of the beta distribution is the **Dirichlet distribution**. This distribution may look a little bit weird at first because it 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 positive and sum up to one. To understand how the Dirichlet generalize the beta, let us first refresh a couple of features of the beta distribution. We use the beta for 2-outcome problems, one with probability p and the other *1-p*. In this sense we can think that the beta returns a two-element vector, *[p, 1-p]*. Of course, in practice, we omit *1-p* because it is fully determined by *p*. Another feature of the beta distribution is that it is parameterized using two scalars $\alpha$ and $\beta$. How does these features compare to the Dirichlet distribution? Let us think of the simplest Dirichlet distribution, one we could use to model a three-outcome problem. We get a Dirichlet distribution that returns a three element vector *[p, q , r]*, where *r=1 – (p+q)*. We could use three scalars to parameterize such Dirichlet and we may call them $\alpha$, $\beta$, and $\gamma$; however, it does not scale well to higher dimensions, so we just use a vector named $\alpha$ with lenght *k*, where k is the number of outcomes. Note that we can think of the beta and Dirichlet as distributions over probabilities. To get an idea about this distribution pay attention to the following figure and try to relate each *triangular* subplot to a beta distribution with similar parameters.




The preceding figure is the output of the code written by Thomas Boggs with just a few minor tweaks. You can find the code in the accompanying text; also check the *Keep reading* sections for details.

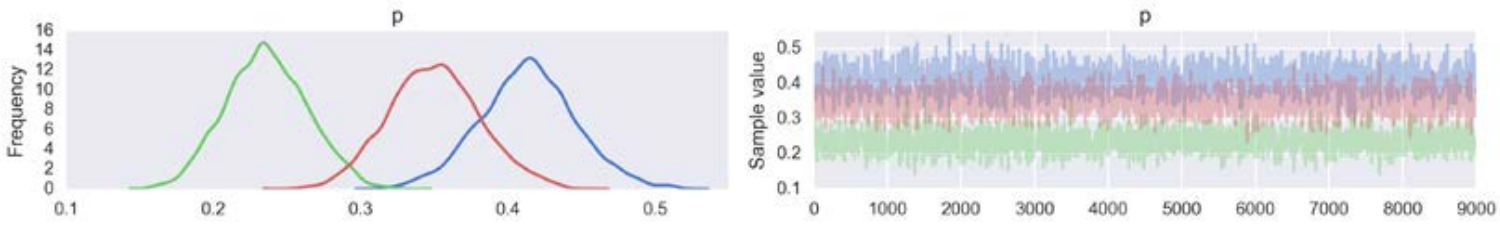
Now that we have a better grasp of the Dirichlet distribution we have all the elements to build our mixture model. One way to visualize it, is as a k-side coin flip model on top of a Gaussian estimation model. Of course, instead of k-sided coins you may prefer to think in terms of a k-side dice. Using Kruschke-style diagrams, we can visualize this model as:



The rounded-corner box is indicating that we have k-Gaussian likelihoods (with their corresponding priors) and the categorical variables decide which of them we use to describe a given data point.

Remember, we are assuming we know the means and standard deviations of the Gaussians; we just need to assign each data point to one Gaussian. One detail of the following model is that we have used two samplers, `Metropolis` and `ElemwiseCategorical`, which is specially designed to sample discrete variables.

```
with pm.Model() as model_kg:
    p = pm.Dirichlet('p', a=np.ones(clusters))
    category = pm.Categorical('category', p=p, shape=n_total)

    means = pm.math.constant([10, 20, 35])

    y = pm.Normal('y', mu=means[category], sd=2, observed=mix)

    step1 = pm.ElemwiseCategorical(vars=[category],
values=range(clusters))
    step2 = pm.Metropolis(vars=[p])
    trace_kg = pm.sample(10000, step=[step1, step2])
      chain_kg = trace_kg[1000:]
       varnames_kg = ['p']
       pm.traceplot(chain_kg, varnames_kg)
```



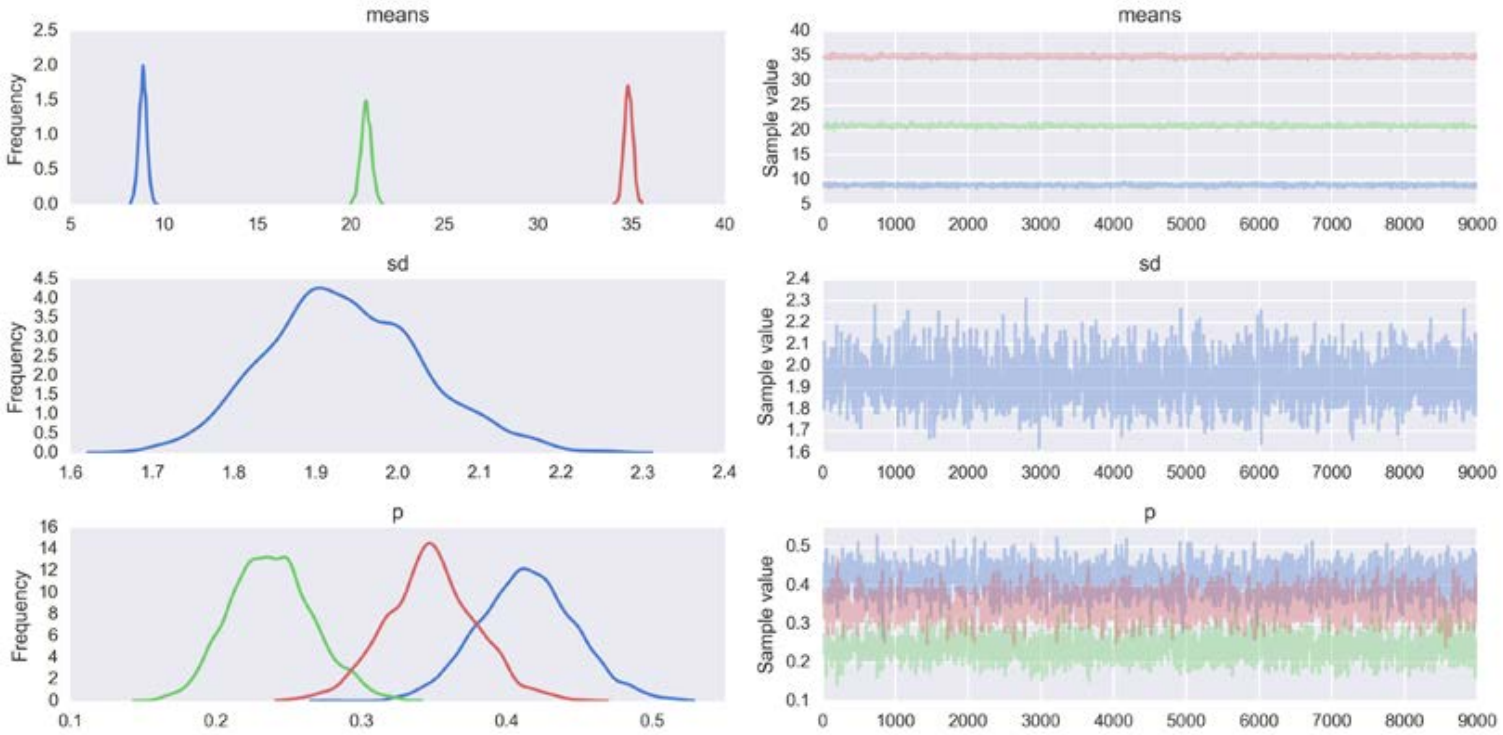

Now that we know the skeleton of a Gaussian mixture model, we are going to add a complexity layer and we are going to estimate the parameters of the Gaussians. We are going to assume three different means and a single shared standard deviation.

As usual, the model translates easily to the PyMC3 syntax.

```
with pm.Model() as model_ug:
    p = pm.Dirichlet('p', a=np.ones(clusters))
    category = pm.Categorical('category', p=p, shape=n_total)

    means = pm.Normal('means', mu=[10, 20, 35], sd=2, shape=clusters)
    sd = pm.HalfCauchy('sd', 5)

    y = pm.Normal('y', mu=means[category], sd=sd, observed=mix)

    step1 = pm.ElemwiseCategorical(vars=[category],
values=range(clusters))
    step2 = pm.Metropolis(vars=[means, sd, p])
    trace_ug = pm.sample(10000, step=[step1, step2])
```

Now we explore the trace we got:

```
chain = trace[1000:]
varnames = ['means', 'sd', 'p']
pm.traceplot(chain, varnames)
```



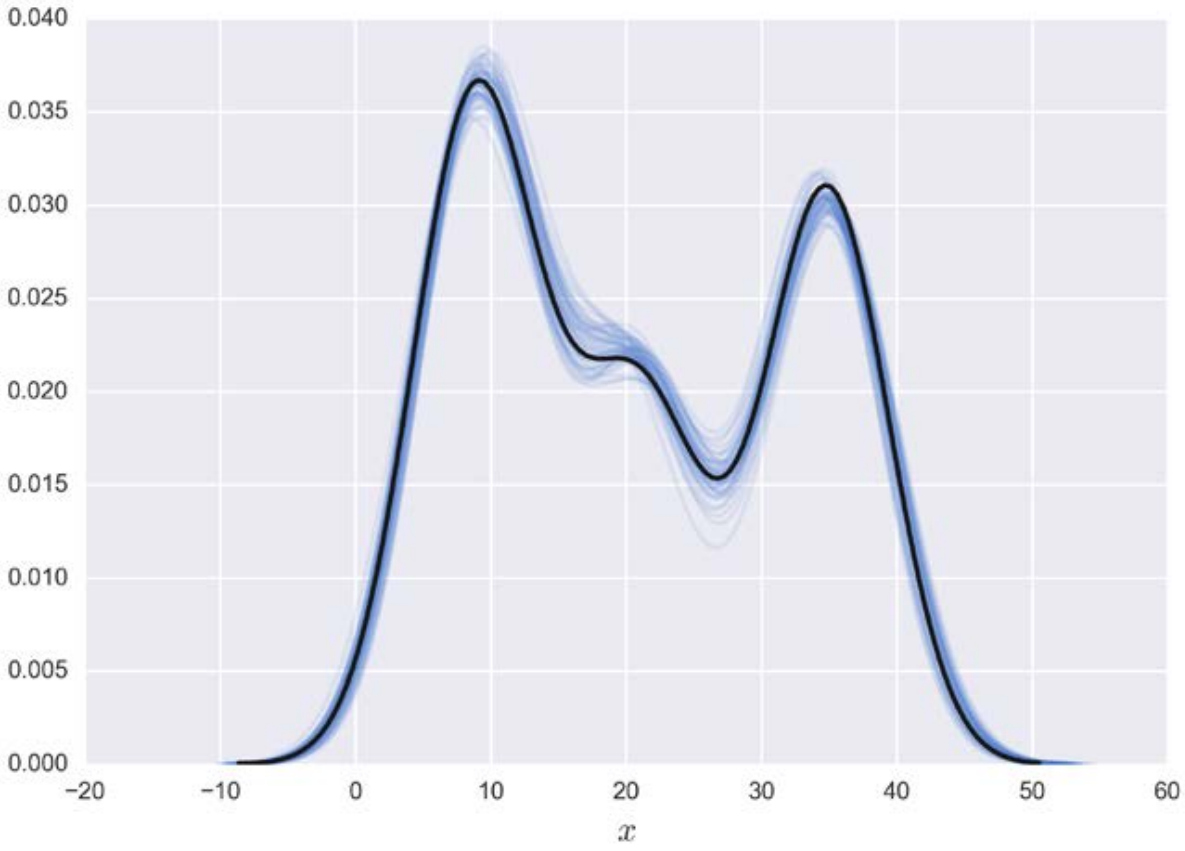
And a tabulated summary of the inference:

```
pm.df_summary(chain, varnames)
```

| | mean | sd | mc_error | hpd_2.5 | hpd_97.5 |
|---|---|---|---|---|---|
| means__0 | 21.053935 | 0.310447 | 0.012280 | 20.495889 | 21.735211 |
| means__1 | 35.291631 | 0.246817 | 0.008159 | 34.831048 | 35.781825 |
| means__2 | 8.956950 | 0.235121 | 0.005993 | 8.516094 | 9.429345 |
| sd | 2.156459 | 0.107277 | 0.002710 | 1.948067 | 2.368482 |
| p__0 | 0.235553 | 0.030201 | 0.000793 | 0.179247 | 0.297747 |
| p__1 | 0.349896 | 0.033905 | 0.000957 | 0.281977 | 0.412592 |
| p__2 | 0.347436 | 0.032414 | 0.000942 | 0.286669 | 0.410189 |

Now we are going to do a predictive posterior check to see what our model learned from the data:

```
ppc = pm.sample_ppc(chain, 50, model)
for i in ppc['y']:
    sns.kdeplot(i, alpha=0.1, color='b')

sns.kdeplot(np.array(mix), lw=2, color='k')
plt.xlabel('$x$', fontsize=14)
```



Notice how the uncertainty, represented by the lighter blue lines, is smaller for the smaller and larger values of $x$ and is higher around the central Gaussian. This makes intuitive sense since the regions of higher uncertainty correspond to the regions where the Gaussian overlaps and hence it is harder to tell if a point belongs to one or the other Gaussian. I agree that this is a very simple problem and not that much of a challenge, but it is a problem that contributes to our intuition and a model that can be easily applied or extended to more complex problems.

# Marginalized Gaussian mixture model

In the preceding models, we have explicitly included the latent variable $z$ in the model. One problem with this model is that sampling the discrete latent variable $z$ usually leads to slow mixing and ineffective exploration of the tails of the distribution. Using a specific tailored sampler for the discrete latent variable can help improve the sampling. An alternative is to write an equivalent model but using a different parametrization. Note that in a mixture model the observed variable $y$ is modeled conditionally on the latent variable $z$. That is:

$$p(y \mid z, \theta)$$

We may think of the $z$ latent variable as a nuisance variable that we can marginalize and get:

$$p(y \mid \theta)$$

Luckily for us, PyMC3 includes a distribution that effectively does this, without us needing to explicitly do the math. So we can write a Gaussian mixture model in the following way.

```
with pm.Model() as model_mg:
    p = pm.Dirichlet('p', a=np.ones(clusters))

    means = pm.Normal('means', mu=[10, 20, 35], sd=2, shape=clusters)
    sd = pm.HalfCauchy('sd', 5, shape=clusters)

    y = pm.NormalMixture('y', w=p, mu=means, sd=sd, observed=mix)

    step = pm.Metropolis()
    trace_mg = pm.sample(2000, step)
```

It is left as an exercise for the reader to run this model and made all the necessary plots to explore the results.

# Mixture models and count data

Sometimes we use data that results from 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** and one common distribution used to model it is the Poisson distribution.

# The Poisson distribution

Imagine we are counting the number of red cars passing through an avenue per hour. We could use the Poisson distribution to describe this data. The 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, $\lambda$ (the rate) that corresponds to the mean and also the variance of the distribution. The probability mass function is:

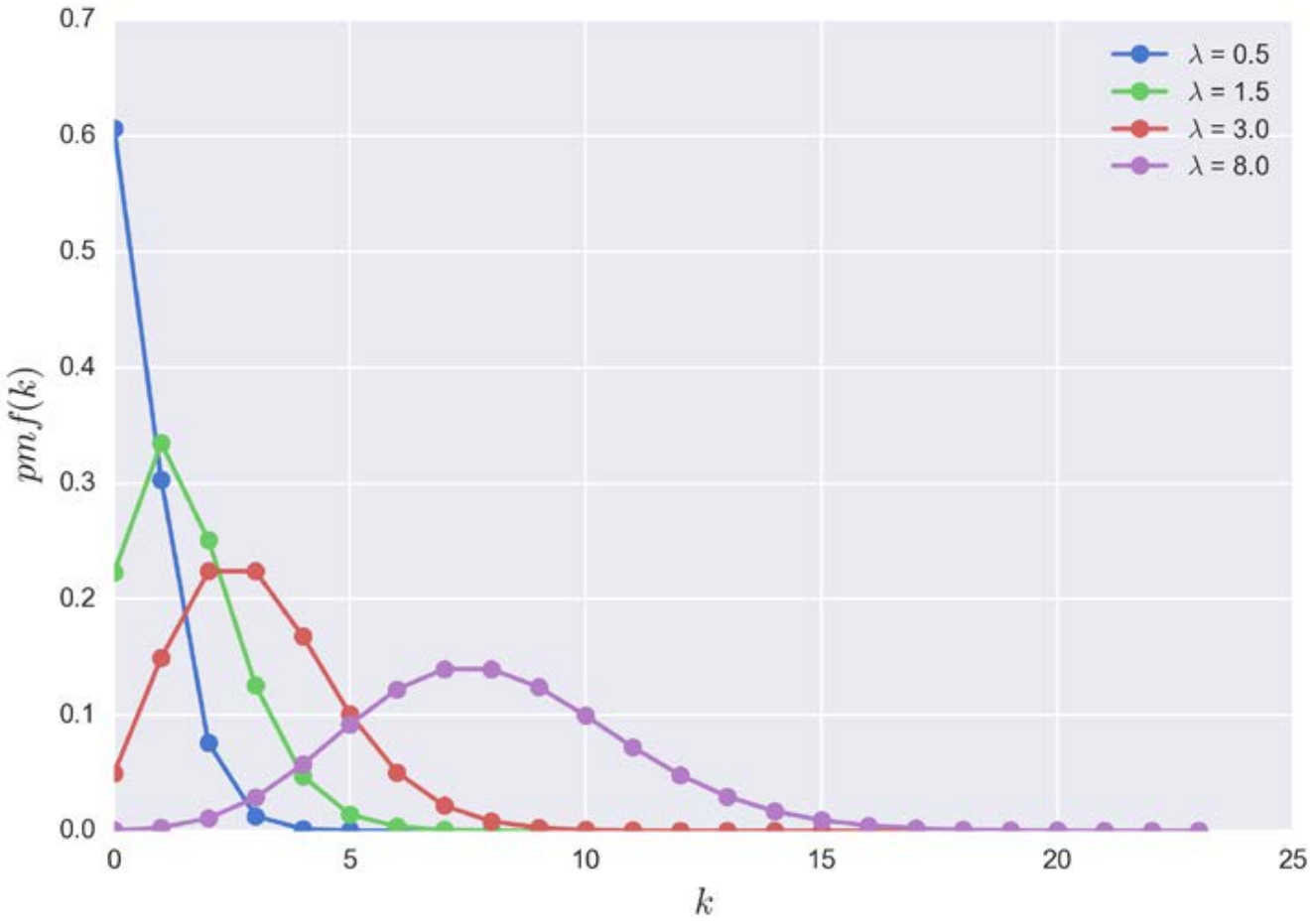
$$pmf\left(k\right)=\frac{\lambda^{k}e^{-\lambda}}{k!}$$

where:

- $\lambda$ is the average number of events per unit of time/space
- $k$ is a positive integer value *0, 1, 2, …*
- *k!* is the factorial of $k$, $k!=k\times\left(k-1\right)\times\left(k-2\right)\times\ldots\times2\times1$

In the following plot, we can see some examples of the Poisson distribution family, for different values of $\lambda$.

```
lam_params = [0.5, 1.5, 3, 8]
k = np.arange(0, max(lam_params) * 3)
for lam in lam_params:
    y = stats.poisson(lam).pmf(k)
    plt.plot(k, y, 'o-', label="$\\lambda$ = {:3.1f}".format(lam))
plt.legend()
plt.xlabel('$k$', fontsize=14)
plt.ylabel('$pmf(k)$', fontsize=14)
```




Note that $\lambda$ can be a float, but the output of the distribution is always an integer. In the previous plot 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 the preceding statement. Since we either see the red car or we do not, we can use a binomial distribution to model the number of red cars. In that case we have:

$$x \sim Bin(n, p)$$

Then, the mean of the binomial distribution is:

$$E[x] = np$$

And the variance is given by:

$$Var\left[x\right] = np\left(1-p\right)$$

But notice 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:

$$n \gg p \Rightarrow np \simeq np\left(1-p\right)$$

So, we can make the following approximation:

$$Var\left[x\right] = np$$

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:

$$x \sim Pois\left(\lambda = np\right)$$

# The Zero-Inflated Poisson model

When we are counting things, it often happens that the number zero occurs for more than one reason; we either have 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 we do not get a nice fit especially because we are seeing more zeros than expected if the data was really Poisson-distributed.

How do we fix that? We may try to address the exact cause of our model predicting less 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 model composed of a number coming from a Poisson distribution with probability $\psi$ and extra zeros with probability $1-\psi$. If we opt for the mixture model, we get what is known as a **Zero-Inflated Poisson** (**ZIP**) model. In some texts, you will find that $\psi$ 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 in a concrete example.

Basically a ZIP distribution tells us that:

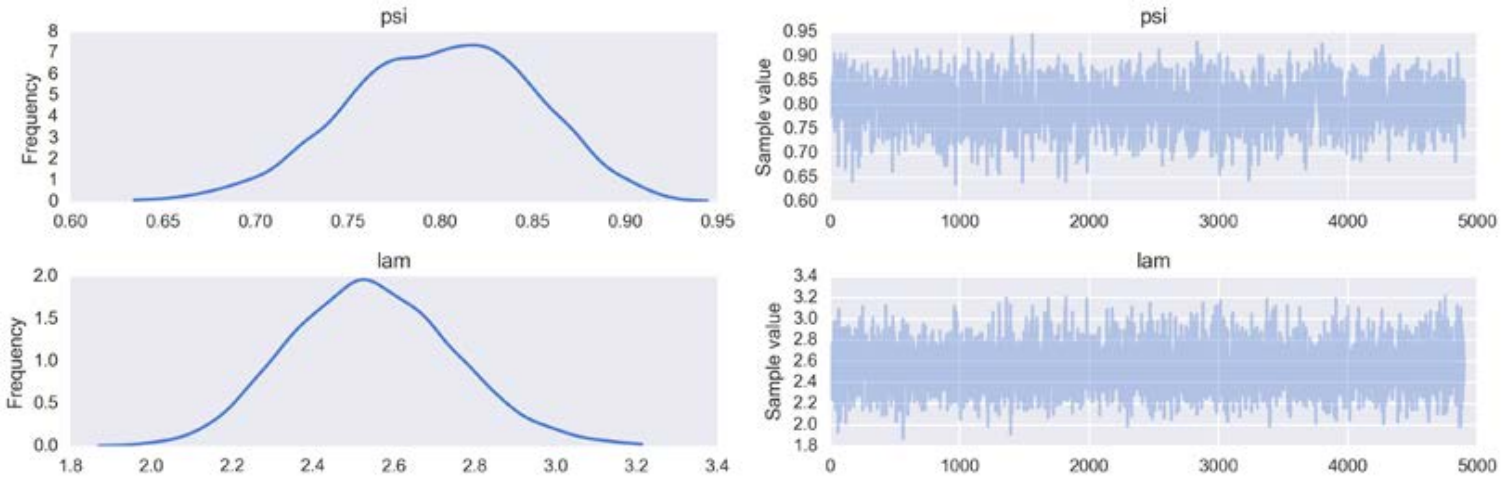
$$p\left(y_j = 0\right) = 1 - \psi + \left(\psi\right) e^{-\lambda}$$

$$p\left(y_j = k_i\right) = \psi \frac{\lambda^{k_i} e^{-\lambda}}{k_i!}$$

Where $1-\psi$ is the probability of extra zeros.

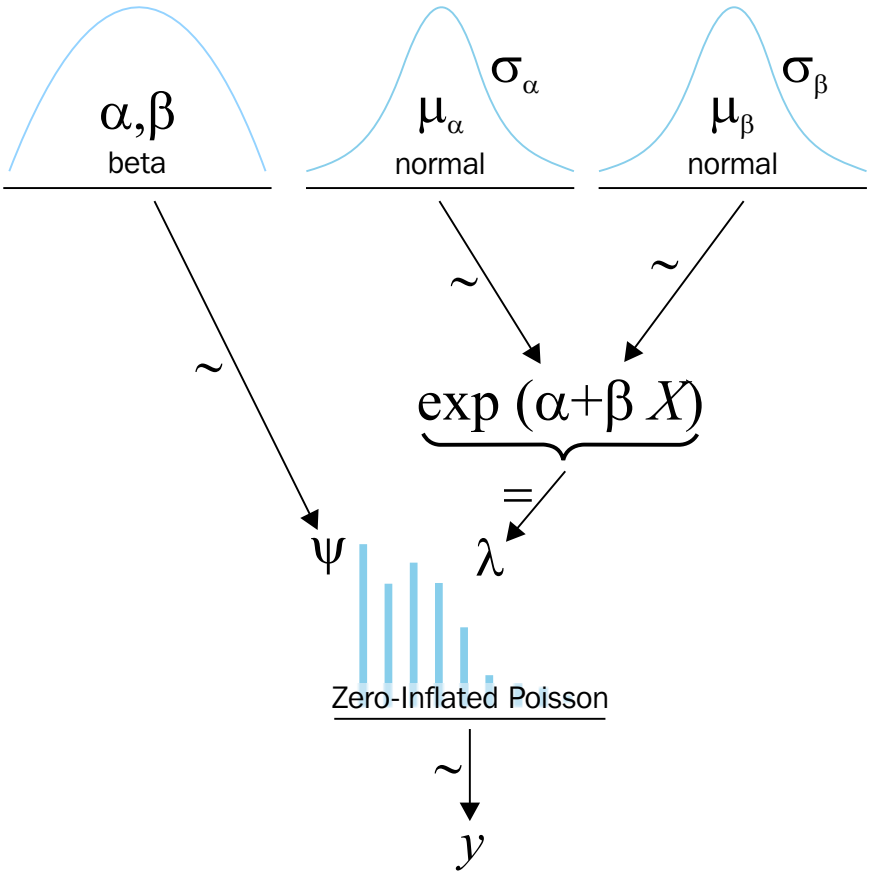
We could implement these equations into a PyMC3 model. However, PyMC3 is equipped with a ZIP distribution, so we can write the model with more ease. Since Python has already *call dibs* on lambda, we instead use the variable name `lam`. Using PyMC3, we can express the `ZIP` model as:

```
with pm.Model() as ZIP:
    psi = pm.Beta('p', 1, 1)
    lam = pm.Gamma('lam', 2, 0.1)

    y = pm.ZeroInflatedPoisson('y', lam, psi, observed=counts)
    trace = pm.sample(1000)
pm.traceplot(trace[:]);
```



# Poisson regression and ZIP regression

The ZIP model may look a little dull, but sometimes we need to estimate simple distributions like this or even ones like the Poisson or Gaussian distributions. Besides, we can use the Poisson or ZIP distributions as part of a linear model. As we saw in *Chapter 4*, *Understanding and Predicting Data with Linear Regression Models*, a simple linear regression model can be built by using a linear model (with the identity inverse link function) and a Gaussian distribution (or Student's t) as a noise or error distribution. In *Chapter 5*, *Classifying Outcomes with Logistic Regression*, we saw how to adapt this model to perform classification. We used the logistic or softmax as the inverse link function and the Bernoulli or categorical, respectively, to model the output variable. Following the same idea, we can now perform a regression analysis when the output variable is a count variable using a Poisson or a ZIP distribution. In the following figure, we see one possible implementation of a ZIP regression. The Poisson regression will be similar, but without the need to include $\psi$ because we will not be modeling an excess of zeros. Notice that now we use the exponential function as an inverse link function. This choice guarantees the values returned by the linear model are positive.




To exemplify a ZIP-regression model implementation, we are going to work with a data set taken from `http://www.ats.ucla.edu/stat/data/fish.csv`. The data is also distributed with the accompanying code.

The problem is as follows: We work at a park administration and we want to improve the experience for the visitors. Thus we decide to take a short survey to 250 groups visiting the park. Part of the data we collected (at the group-level) consists of:

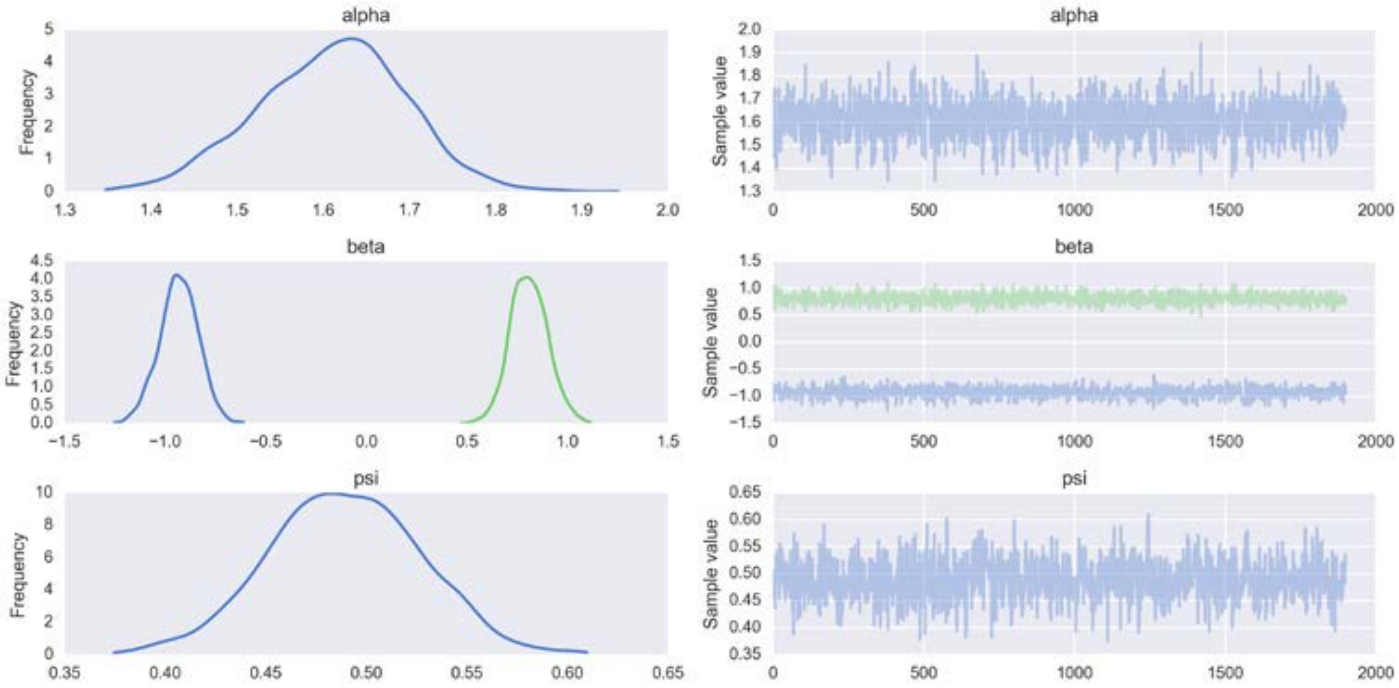
- The number of fish they caught (`count`)
- How many children were in the group (`child`)
- Whether or not they brought a camper to the park (`camper`).

Using this data we are now going to build a model that predicts the number of catched fishes as a function of the variables `child` and `camper`. We can use Pandas to load the data:

```
fish_data = pd.read_csv('fish.csv')
```

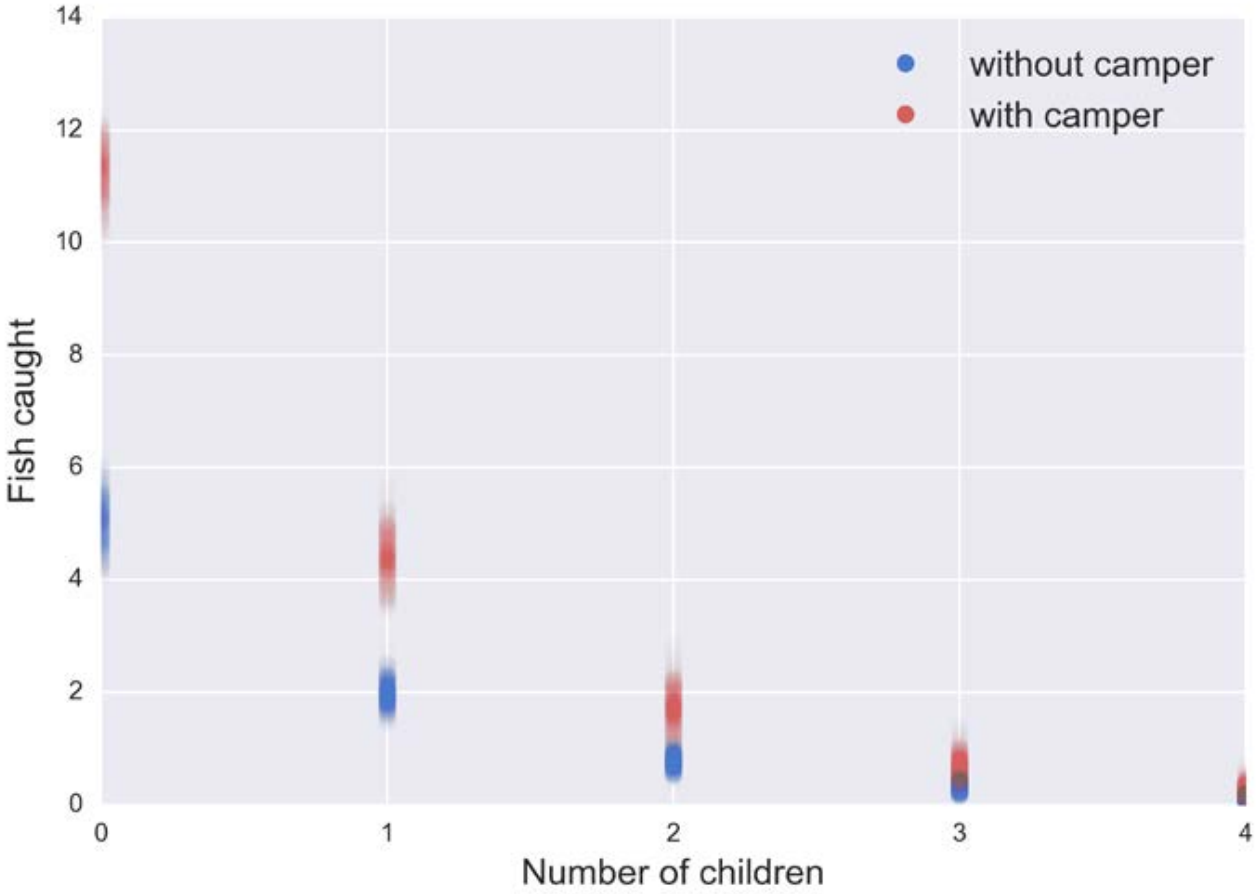
I leave as an exercise for you to explore the dataset using plots and/or a Pandas function, such as `describe()`. For now we are going to continue by translating the above Kruschke diagram to PyMC3:

```
with pm.Model() as ZIP_reg:
    psi = pm.Beta('psi', 1, 1)

    alpha = pm.Normal('alpha', 0, 10)
    beta = pm.Normal('beta', 0, 10, shape=2)
    lam = pm.math.exp(alpha + beta[0] * fish_data['child'] + beta[1] *
fish_data['camper'])

    y = pm.ZeroInflatedPoisson('y', lam, psi, observed=fish_
data['count'])
    trace_ZIP_reg = pm.sample(1000)
chain_ZIP_reg = trace_ZIP_reg[100:]
pm.traceplot(chain_ZIP_reg)
```



As usual, your duty is to do a sanity-check for the sampling process. To better understand the results of our inference, let's do a plot.

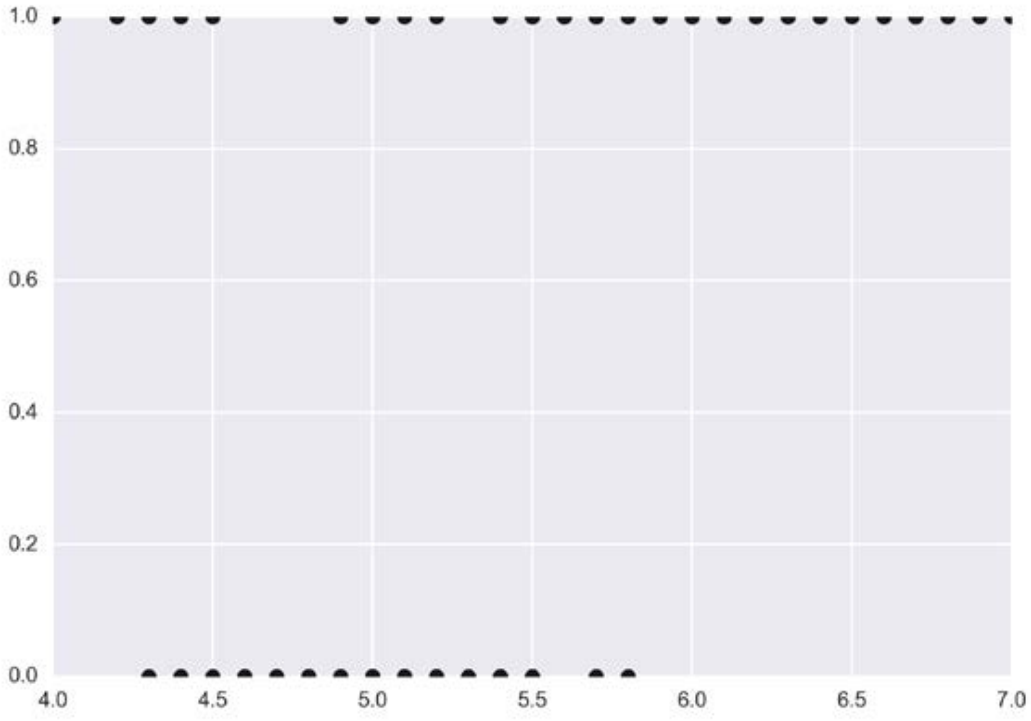
```
children =  [0, 1, 2, 3, 4]
fish_count_pred_0 = []
fish_count_pred_1 = []
thin = 5
for n in children:
    without_camper = chain_ZIP_reg['alpha'][::thin] + chain_ZIP_
reg['beta'][:,0][::thin] * n
    with_camper = without_camper + chain_ZIP_reg['beta'][:,1][::thin]
    fish_count_pred_0.append(np.exp(without_camper))
    fish_count_pred_1.append(np.exp(with_camper))
```



# 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 have already seen, but with some *intruders* added:

```
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)))
x_0 = np.concatenate((x_0, [4.2, 4.5, 4.0, 4.3, 4.2, 4.4]))
x_0_m = x_0 - x_0.mean()
plt.plot(x_0, y_0, 'o', color='k')
```



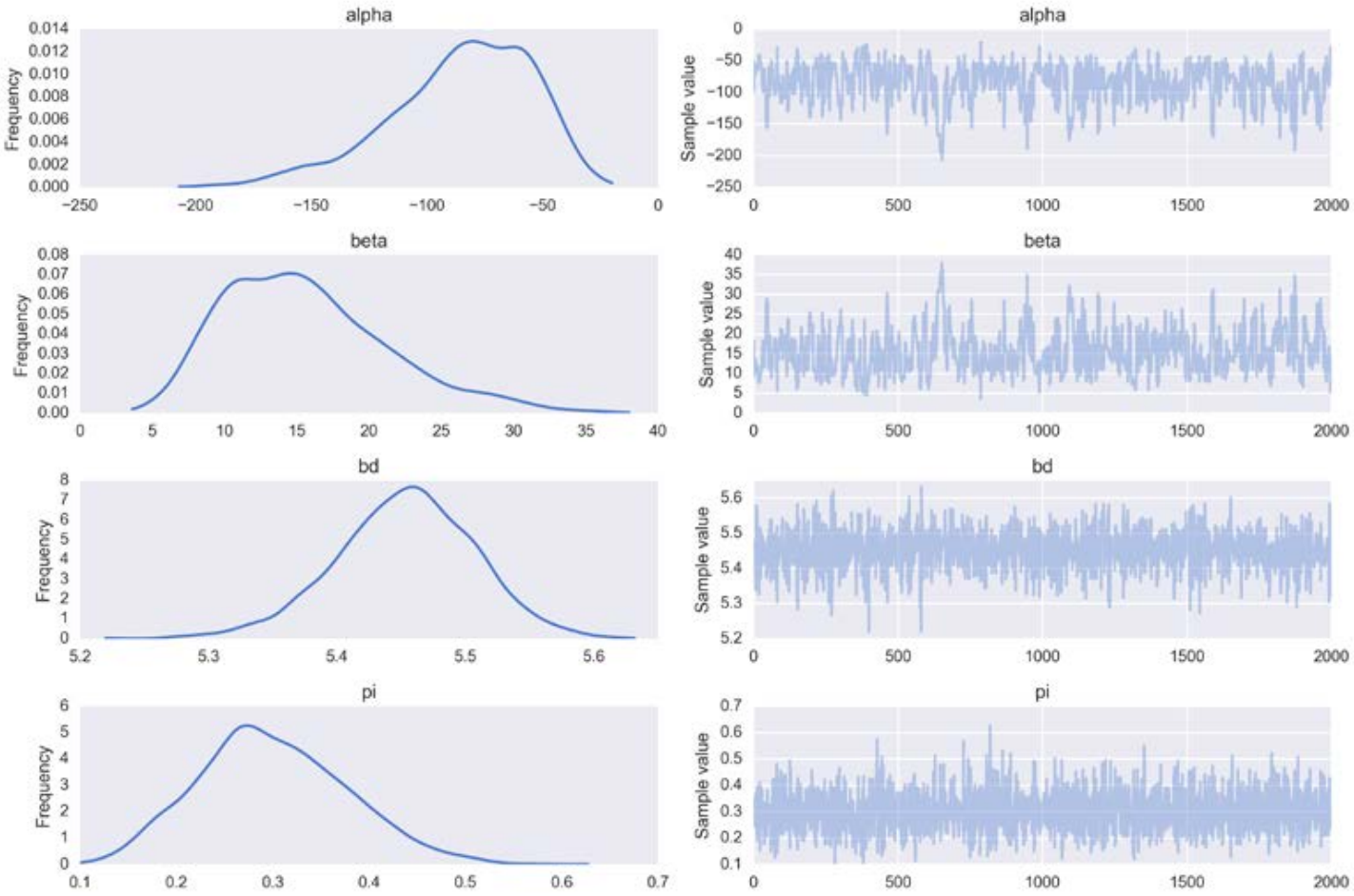

Here we have some versicolors (1s) with an unusually short sepal length. We can fix this with a mixture model. We are going to say that the output variable comes with probability $\pi$ by random guessing or with probability $1-\pi$ from a logistic regression model. Mathematically, we have:

$$p = \pi 0.5 + (1-\pi)\, logistic(\alpha + \beta \boldsymbol{X})$$

Notice that when $\pi = 1$ we get $p = 0.5$, and when $\pi = 2$ we recover the expression for logistic regression.

Implementing this model is a straightforward modification of the one we used in *Chapter 5, Classifying Outcomes with Logistic Regression*.

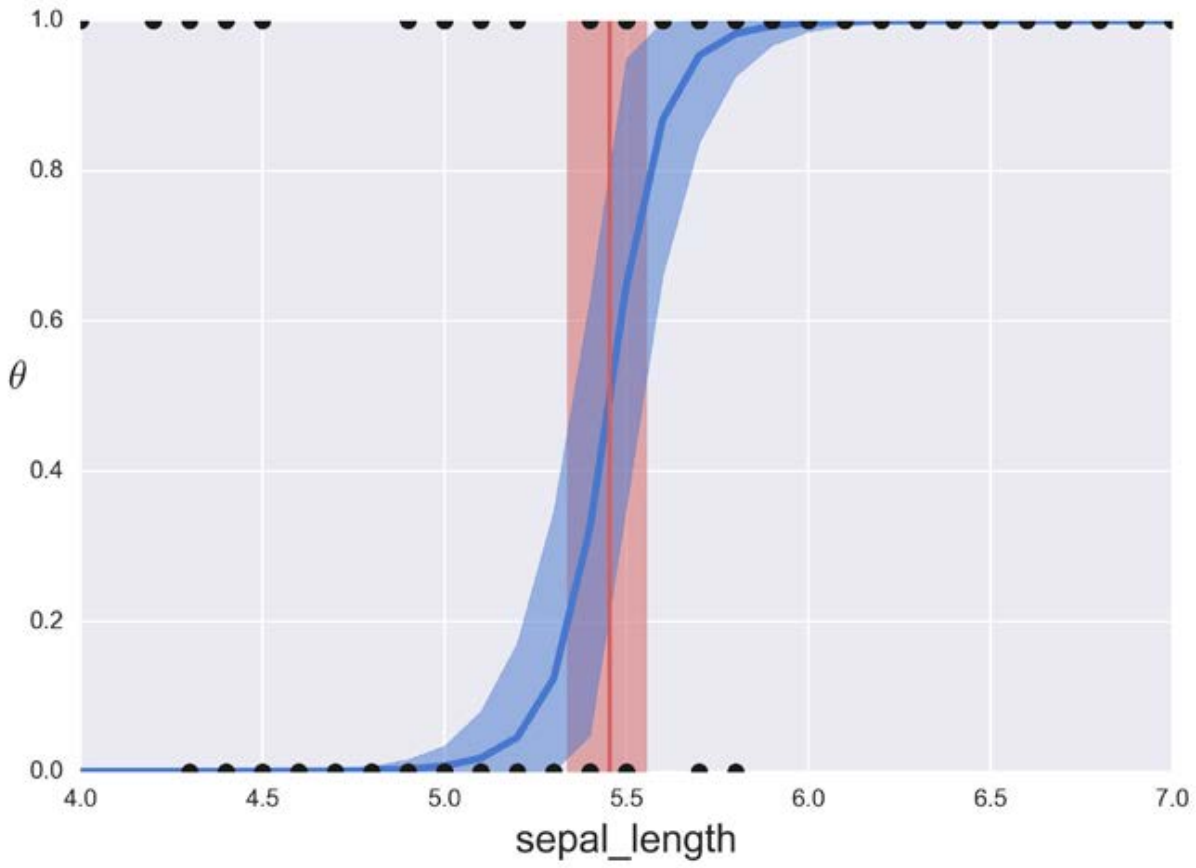
```
with pm.Model() as model_rlg:
    alpha_tmp = pm.Normal('alpha_tmp', mu=0, sd=100)
    beta = pm.Normal('beta', mu=0, sd=10)

    mu = alpha_tmp + beta * x_0_m
    theta = pm.Deterministic('theta', 1 / (1 + pm.math.exp(-mu)))

    pi = pm.Beta('pi', 1, 1)
    p = pi * 0.5 + (1 - pi) * theta

    alpha = pm.Deterministic('alpha', alpha_tmp - beta *
      x_0.mean())
    bd = pm.Deterministic('bd', -alpha/beta)

    yl = pm.Bernoulli('yl', p=p, observed=y_0)

    trace_rlg = pm.sample(2000, start=pm.find_MAP())
varnames = ['alpha', 'beta', 'bd', 'pi']
pm.traceplot(trace_rlg, varnames)
```



If we compare these results we will see that we get more or less the same outcome as in *Chapter 5, Classifying Outcomes with Logistic Regression*:

```
pm.df_summary(trace_rlg, varnames)
```

| | mean | sd | mc_error | hpd_2.5 | hpd_97.5 |
|---|---|---|---|---|---|
| alpha | -88.04 | 32.06 | 1.99 | -151.35 | -32.68 |
| beta | 16.15 | 5.87 | 0.36 | 6.21 | 27.95 |
| bd | 5.45 | 0.05 | 0.00 | 5.34 | 5.55 |
| pi | 0.30 | 0.07 | 0.00 | 0.15 | 0.45 |




# Model-based clustering

Clustering is part of the unsupervised family of statistical/machine learning tasks and is similar to classification, but a little bit more difficult since we do not know the correct labels!

If we do not know the correct labels we can try grouping data points together. Loosely speaking, points that are closer between themselves, under some metric, are defined as belonging to the same group and separated from the other groups. Clustering has many, many applications; for example, **phylogenetics**, a branch of biology studying the evolutionary relationships among biological entities, can be framed as clustering techniques applied to and guided by an evolutionary question. A more capitalist-driven application of clustering is determining which movie/book/song/you-name-the-product we may be interested in. We can try to guess this based on our consumption-record and how this record clusters with those of other users. As with other unsupervised learning tasks, we may be interested in performing clustering by itself or we may want to use it as part of an exploratory data analysis. Now that we have a general idea of what clustering is, let us continue by checking some criteria used to define if two data points need to be grouped together or not.

Often we define how close or similar two data points are by measuring the Euclidean distance between them. The Euclidean distance is, roughly speaking, the straight line between two points. It is important to realize that even when we are not measuring an actual physical distance, let's say in meters or light years, we can define a Euclidean distance. For every two data points $q_i$ and $p_i$ in an n-dimensional feature/variable space, we can compute the Euclidean distance as:

$$d(p,q) = d(q,p) = \sqrt{\sum (q_i - p_i)^2}$$

An algorithm that, in its more general incarnation, uses the Euclidean distance is k-means clustering. We are not going to see it now, but I recommend you to read about it because it is a nice introductory example to the clustering problem and one that is easy to understand, interpret, and implement by yourself. You can find more about it in the book, *Python Machine Learning* by *Sebastian Raschka*.

Computing Euclidean distances or some other measure of closeness is not the only way to cluster data. An alternative is to take a probabilistic approach and assume the observed data was generated from some probability distribution, and then build a fully probabilistic model . This approach is often known as model-based clustering.

A mixture model is a natural candidate when it comes to solving the clustering problem under the Bayesian framework. Indeed, the first example in this chapter was about using a mixture model as a way to model (unlabeled/unobserved) subpopulations that combine to give the larger observed population. 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 probability *0* or *1*. Of course, it is possible to turn soft-clustering into hard-clustering by introducing some rule or boundary, like we did with logistics regression. A reasonable choice is to compute for each point the probability of belonging to each cluster and assign it to the cluster with the highest probability.

# Fixed component clustering

The first example in this chapter was an example of a mixture model applied to a clustering problem. We had an empirical distribution of data points and we used three Gaussians to describe subpopulations in the data. Notice that instead of thinking on a similarity/dissimilarity metric we performed inference on our data, assuming a mixture of Gaussians.

# Non-fixed component clustering

For some problems, such as trying to clusterize handwritten digits, it is easy to justify the number of clusters we should find in the data. For other problems we can have good guesses; for example, for a dataset similar to the iris one we may know that the samples were taken from a region where only three Iris species grow. For other problems, choosing a priori the number of clusters can be a shortcoming. One Bayesian solution to this type of problem is using a non-parametric method to estimate the number of clusters. We can do this using a Dirichlet Process. In *Chapter 8*, *Gaussian Processes*, we are going to learn about non-parametric statistics; nevertheless, we are not covering the Dirichlet process in this book. I recommend that after reading *Chapter 8*, *Gaussian Processes*, you read and run the following notebook (`https://pymc-devs.github.io/pymc3/notebooks/dp_mix.html`). This great introduction to *Dirichlet Processes* was written by *Austin Rochford* (PyMC3 contributor and reviewer of this book).

# 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. One example of a continuous mixture model is the robust logistic regression model that we saw earlier. This is a mixture of two components: a logistic on one hand and a random guessing on the other. Note that the parameter $\pi$ 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 $\pi$ 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 it own slope or that all the groups share the same slope. Alternatively, instead of framing our problem as two extreme discrete options, when we build a hierarchical model, we are effectively modeling a continuous mixture of these extreme options. So 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 $y$ in an $n$ number of Bernoulli trials when the probability of success $p$ at each trial is unknown and assumed to follow a beta distribution with parameters $\alpha$ and $\beta$.

$$BetaBinonial\left(y \mid n, \alpha, \beta\right) = \int_0^1 Bin\left(y \mid p, n\right) Beta\left(p \mid \alpha, \beta\right) dp$$

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 being 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-binomal distribution.

In a similar fashion, we have the negative-binomial distribution, which can be understood as a **gamma-Poisson mixture** continuous model, that is, by averaging (integrating) a Poisson probability over continuous values of rates coming from a gamma distribution. 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:

$$t_v\left(y \mid \mu, \sigma\right) = \int_0^\infty N\left(y \mid \mu, \sigma\right) Inv\chi^2\left(\sigma \mid v\right) dv$$

Notice this is similar to the previous expression for the negative-binomial except that now we have a **Normal distribution** with the parameters $\mu$ and $\sigma$ and the $Inv\chi^2$ distribution with the parameter $\nu$ from which we sample the values of $\sigma$ is the parameter known as a degree of freedom, or as we prefer to call it, the **normality parameter**. The parameter $\nu$, 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, winch may lead to an easier to sample model, as we already saw with the marginalized mixture model example.

# Summary

In this chapter, we learned about mixture models, a type of hybrid model useful to solve a large collection of problems. Creating a finite mixture model is a relatively easy task given what we have learned from previous chapters. A very handy application of this type of model is dealing with an excess of zeros in count data or for example to expand a Poisson model if we observe over-dispersion. Another application we explored was about extending logistic regression to handle outliers. We also briefly discussed the central elements of performing Bayesian (or model-based) clustering. Lastly, we presented a more theoretical discussion about continuous mixture models and how these types of models are connected to concepts we already learned in previous chapters, such as hierarchical models and the Student's t-distribution for robust models.

# Keep reading

- Chapter 11, *Statistical Rethinking*, *Richard McElreath*
- Chapter 21, *Doing Bayesian Data Analysis, Second Edition*. *John Kruschke*
- Chapter 22, *Bayesian Data Analysis, Third Edition Gelman et al*

# Exercises

1. For the first example, modify the synthetic data to make it harder for the model to recover the true parameters; try increasing the overlap of the 3 Gaussians by changing the means and standard deviations. Try changing the number of points per cluster, and think of ways to improve the model for the harder data you come up with.
2. Using the `fish` data, extend the model in the book 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 with 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 $\psi$ variable. For the second case you will need a logistic inverse link!

3. 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.
4. Read and run all the PyMC3 notebook examples about Mixture models (`https://pymc-devs.github.io/pymc3/examples.html#mixture-models`).
5. Use a mixture model to cluster the three iris species, using two features. Assume for a moment that you do not know the correct species/labels. You will need to define three multivariate Gaussians, each one with six means. As a first approach, use a single shared covariance matrix.

# 8

# Gaussian Processes

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. Another type of models are those known as **non-parametric models**. Non-parametric models are models where the number of parameters increases with the data, in other words, models with a potentially infinite number of parameters that we somehow manage to reduce to a finite number, just those necessary to describe the data. We will began the chapter, by learning about the concept of a kernel, and how to rethink problems in terms of kernels. Gaussians are the workhorse of statistics and this is not only true for classical methods, but also Bayesian statistics and machine learning. We are going to see a clear example of this as we explore how to extend the notion of Gaussian distribution to infinitely large dimensions and how to learn distributions over functions. Even though this will seem really weird at first, it will allow us to infer functions through the use of parameterized kernels.

In this chapter, we will learn about:

- Non-parametric statistics
- Kernels
- Kernelized regression
- Gaussian processes and prior over functions

# Non-parametric statistics

Non-parametric statistics is often described as the set of statistical tools/models that do not rely on parameterized families of probability distributions. From this definition, it may sound as if Bayesian non-parametric is not possible since we have learned that the first step in doing Bayesian statistics is precisely combining probability distributions in a full probabilistic model. We said in *Chapter 1*, *Thinking Probabilistically - A Bayesian Inference Primer*, that probability distributions are the building blocks of probabilistic models. Under the Bayesian paradigm, non-parametric models refer to models with an infinite number of parameters. So, we will define parametric models as those models for which the number of parameters is allowed to grow with the size of the data. For these models, the theoretical numbers of parameters is infinite and we use the data to *collapse it* to a finite number, thus we allow the data to effectively determine the number of parameters.

# Kernel-based models

The study of kernel-based methods is a very productive and active area of research, with entire books dedicated to the subject. Their popularity relies on some interesting mathematical properties of kernels. For the sake of our current introduction to kernels, we are just going to say that we can use kernels as the basis of flexible non-linear models that also are relatively easy to compute. Two popular kernel-based methods are the **support vector machine** (**SVM**) and the Gaussian processes. The later is a probabilistic method and is the topic of this chapter while the former is a non-probabilistic method that we are not going to discuss it here, you can read more about it in the following books *Python Data Science Handbook*, *Jake Vanderplas* and *Python Machine Learning*, *Sebastian Raschka*. Before discussing Gaussian Processes, let's explore what kernels are and how we can use them.

You may find more than one definition of kernel in the statistical literature, and according to those definitions kernels will have 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 that is always positive. If these conditions are met, we can interpret the output of a kernel function as a measure of similarity between the two inputs.

There are several useful kernels, some of them are specifically tailored to problems such as image recognition or document analysis, others are better suited to modeling periodic functions, and so on. A popular kernel used in many statistical and machine-learning methods is the Gaussian Kernel, also known as the Gaussian radial basis function.

# The Gaussian kernel

The Gaussian kernel is defined as:

$$K\left(x, x'\right) = \exp\left(-\frac{\left\|\boldsymbol{x} - \boldsymbol{x}'\right\|^2}{w}\right)$$

Where $\left\|\boldsymbol{x} - \boldsymbol{x}'\right\|^2$ is the **squared Euclidean distance** (**SED**). For an n-dimensional space we have:

$$\left\|\boldsymbol{x} - \boldsymbol{x}'\right\|^2 = \left(x_1 - x_1'\right)^2 + \left(x_2 - x_2'\right)^2 + \ldots + \left(x_n - x_n'\right)^2$$

Notice that if we were computing the Euclidean distance we should have taken the square distance. The SED does not satisfy the triangle inequality and thus is not a true distance. Nevertheless, it is commonly used in many problems in which we only need to compare SED: for example, finding the minimum Euclidean distance between a set of points is the same as finding the minimum SED.

It may not be obvious at first sight, but the Gaussian kernel has the a similar formula as the Gaussian distribution (see *Chapter 4*, *Understanding and Predicting Data with Linear Regression Models*) where we have dropped the normalization constant and we have defined $w = 2\sigma^2$. Thus, the term $w$ is proportional to the variance and controls the width of the kernel, and is sometimes known as the **bandwidth**.

# Kernelized linear regression

We have learned that the basic motif of the linear regression model has the form:

$$y = f\left(\boldsymbol{x}\right) + \in$$

Where $\in$ is the error or noise and is generally Gaussian distributed.

$$f(\boldsymbol{x}) = \boldsymbol{\mu} = \sum_i^n \boldsymbol{\gamma x}$$

Here we are using *f(.)* to represent the linear function (without noise) with the identity link function. If we are using other inverse link functions, as we did, for example, in *Chapter 5*, *Classifying Outcomes with Logistic Regression*, we will include it inside this *f(.)*. The vector $\gamma$ is the vector of coefficients, in general, an intercept and one or more slopes.

In *Chapter 4*, *Understanding and Predicting Data with Linear Regression Models*, we introduced the concept of polynomial regression and we warned that probably the only practical use of polynomial regression (at least of order greater than two or three) is to introduce statistical concepts in books. We saw how polynomial regression can be used to model non-linear data using a linear model.

Notice we can write the polynomial regression as:

$$\mu = \sum \gamma_i \phi_i(x)$$

Where the $\phi$ function represents a series of polynomials of increasing order. By transforming our input vector **x** into a matrix where each column is a power of that vector, we are effectively projecting our data into a higher dimensional space. And we are finding a straight line in that higher dimensional space that fits the data. When projected back to the original lower dimensional space, that straight line is not necessarily a straight line, but a curve. This is usually referred as projecting the inputs into the feature space.

The function $\phi$ does not need to be a polynomial, there are other functions we can use $\phi$ to map the input vector into a (higher dimensional) feature space. It turns out that under certain conditions, instead of using the $\phi$ function, we can replace it by a kernel. While being mathematically equivalent it is often more computationally convenient to use kernels, this is known as the kernel trick. And this is the main reason why the kernel is the central object in many statistical and machine learning methods and the feature space concept is less important in practice, although it is a useful concept for intuitive insights when learning about kernel-based methods.

Continuing with our discussion, and without getting into the mathematical details, we are going to replace $\phi$ with a kernel *K*. We are going to use let *K* be the Gaussian kernel . We get the following:

$$\boldsymbol{\mu} = \sum_i^N \gamma_i K_i\left(\boldsymbol{x}, \boldsymbol{x}'\right)$$

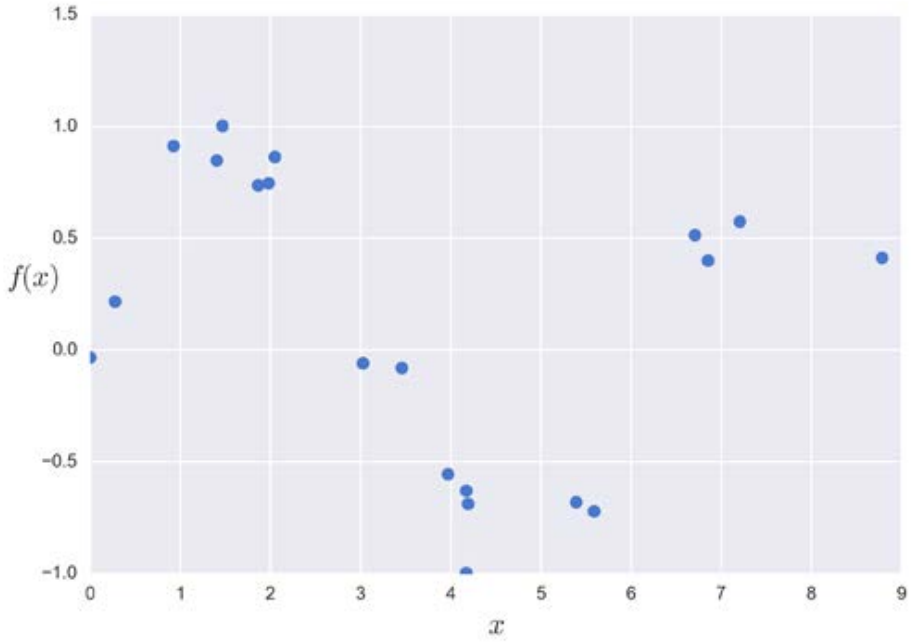
Notice that now we have our data $\boldsymbol{x}$ and also a new vector named $\boldsymbol{x}'$ . The later is a vector of points, usually referred to as **knots** or **centroids** which are distributed somehow, for example, uniformly on the range of the data. As a special case we could have $\boldsymbol{x} = \boldsymbol{x}'$; in other words nothing prevents us from choosing the data points as the knots.

Why are we adding these extra points? Instead of answering that question directly, let's see how we are going to use the knots. Notice that the closer the knots are to a data point, the larger the value returned by the kernel function. To simplify the analysis, assume $w = 1$, then if we have $\boldsymbol{x}_i = \boldsymbol{x}'_j$ we will have $k\left(x_i, x'_j\right) = 1$ and if $x_i$ and $x'_j$ are far away from each other we will have $k\left(x_i, x'_j\right) \approx 0$. In other words, since the output of the Gaussian kernel is a measure of similarity, we are saying thats if $x_i$ and $x'_j$ are similar, the mean value of the function at those points should also be similar, $\mu_i \sim \mu_j$. If we vary $x_i$ a little bit we expect $\boldsymbol{\mu}$ to vary a little bit, if we take a larger step on $\boldsymbol{x}$ we expect a larger change on $\boldsymbol{\mu}$. If we think about this for a moment, we will recognize this as a reasonable feature of our model, our experience tells us that many functions behave this way; in fact, we have a name for this type of function, we call them **smooth functions**. Exceptions arise, of course, but a whole array of problems can be approximated using smooth functions.

OK, keep going with yet another interpretation of what we are doing. We are effectively trying to approximate a smooth unknown function by using the grid approach we saw in *Chapter 1, Thinking Probabilistically - A Bayesian Inference Primer* and *Chapter 2, Programming Probabilistically – A PyMC3 Primer*. The grid points are the $\boldsymbol{x}'$ points. At each knot we are placing a Gaussian function and weighting up or down (through $\gamma$) each of those Gaussians according to the data points. If we sum up the Gaussians, we will obtain a smooth curve that approximates the value of $\boldsymbol{\mu}$. This interpretation is known as the **Weight-space view**. Later in the *Gaussian processes* section we are going to take a look at an alternative way of formulating the problem (and getting the same results) known as the **Function-space view**.

Let's put all these ideas in action by using a simple synthetic dataset, where the dependent variable is a `sin` function and the independent variable is just a set of points from a uniform distribution:

```
np.random.seed(1)
x = np.random.uniform(0, 10, size=20)
y = np.sin(x)
plt.plot(x, y, 'o')
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$f(x)$', fontsize=16, rotation=0)
```




Now we are going to write a convenient simple function to compute a Gaussian kernel in our model:

```
def gauss_kernel(x, n_knots=5, w=2):
    """
    Simple Gaussian radial kernel
    """
    knots = np.linspace(np.floor(x.min()), np.ceil(x.max()),
      n_knots)
    return np.array([np.exp(-(x-k)**2/w) for k in knots])
```

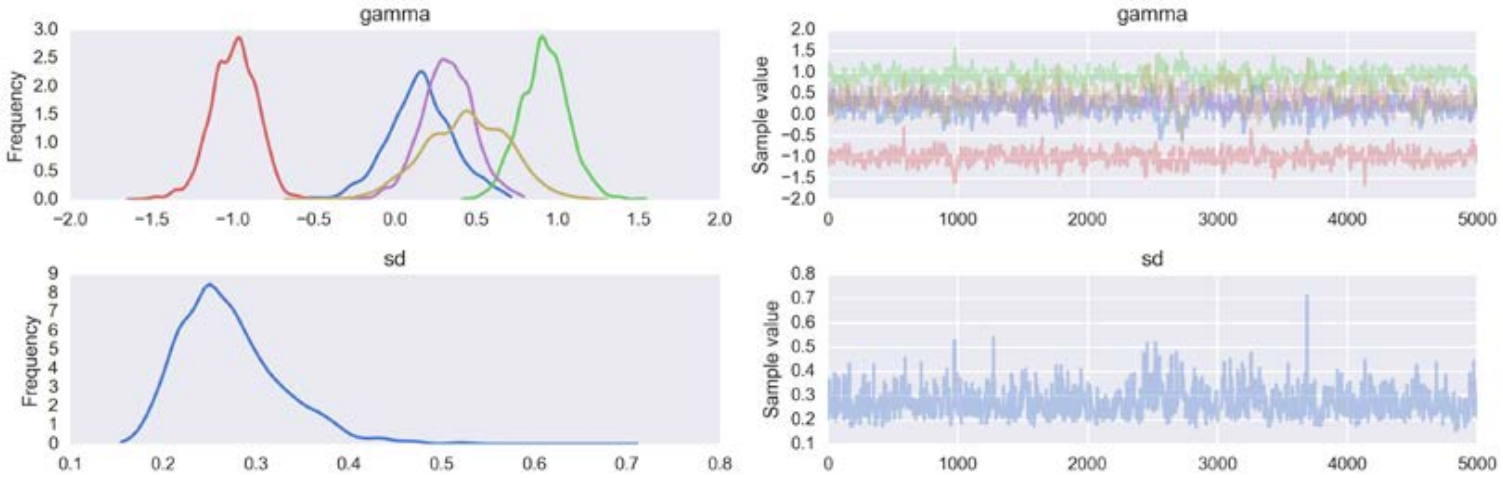
All that is left is determining the $\gamma$ coefficients. We will need as many coefficients as knots. Remember that the coefficients indicate if the curve we are estimating should increase or decrease at the knots.

Sometimes it makes sense to write the model as in a linear regression, something like the following:

$$\mu = \alpha + \beta x + \sum_i^N \gamma_i K_i\left(x, x'\right)$$

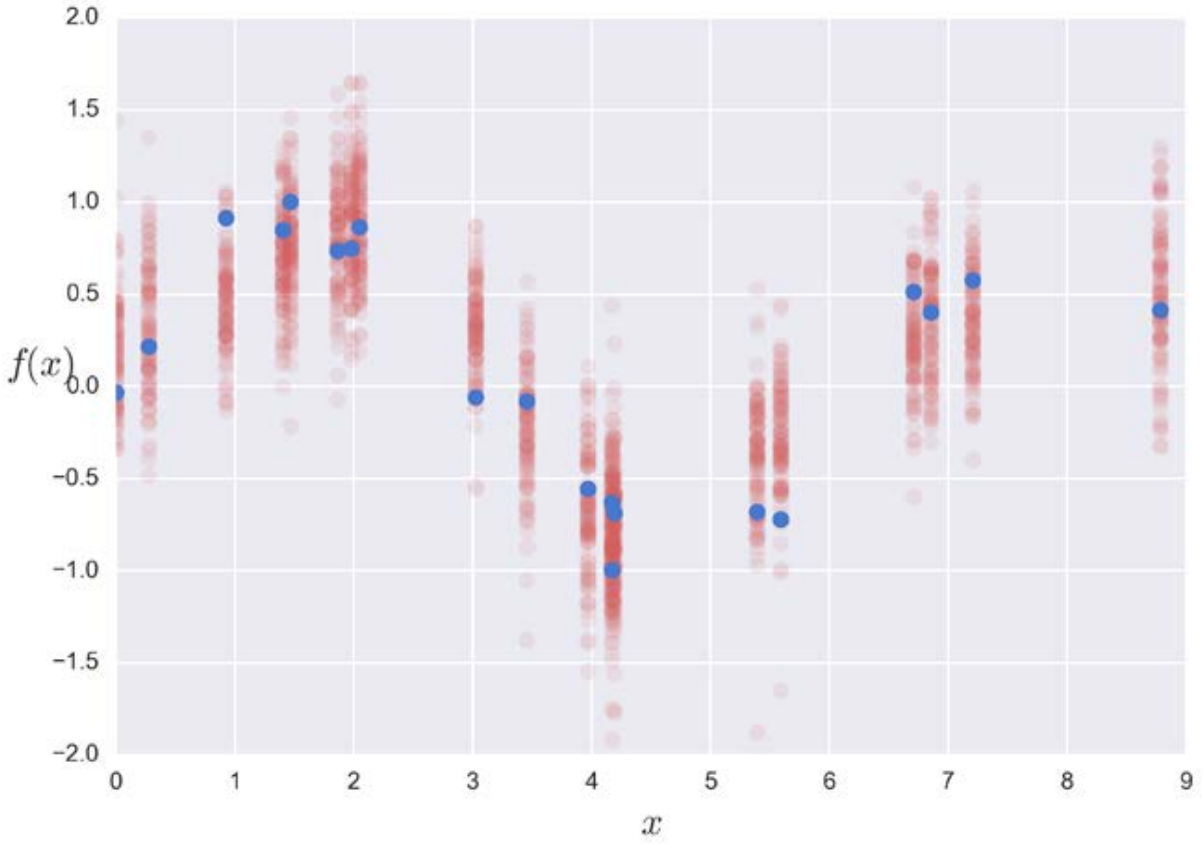
But now we are going to omit specifying an intercept $\alpha$ and the slope $\beta$ and we are just going to use the last term. As priors we are using a Cauchy distribution, later we are going to discuss some important points about choosing priors when working with kernel methods, but now let's and run some computations:

```
with pm.Model() as kernel_model:
    gamma = pm.Cauchy('gamma', alpha=0, beta=1, shape=n_knots)
    sd = pm.Uniform('sd',0,  10)
    mu =  pm.math.dot(gamma, gauss_kernel(x, n_knots))
    yl = pm.Normal('yl', mu=mu, sd=sd, observed=y)
    kernel_trace = pm.sample(10000, step=pm.Metropolis())
chain = kernel_trace[5000:]
pm.traceplot(chain);
```



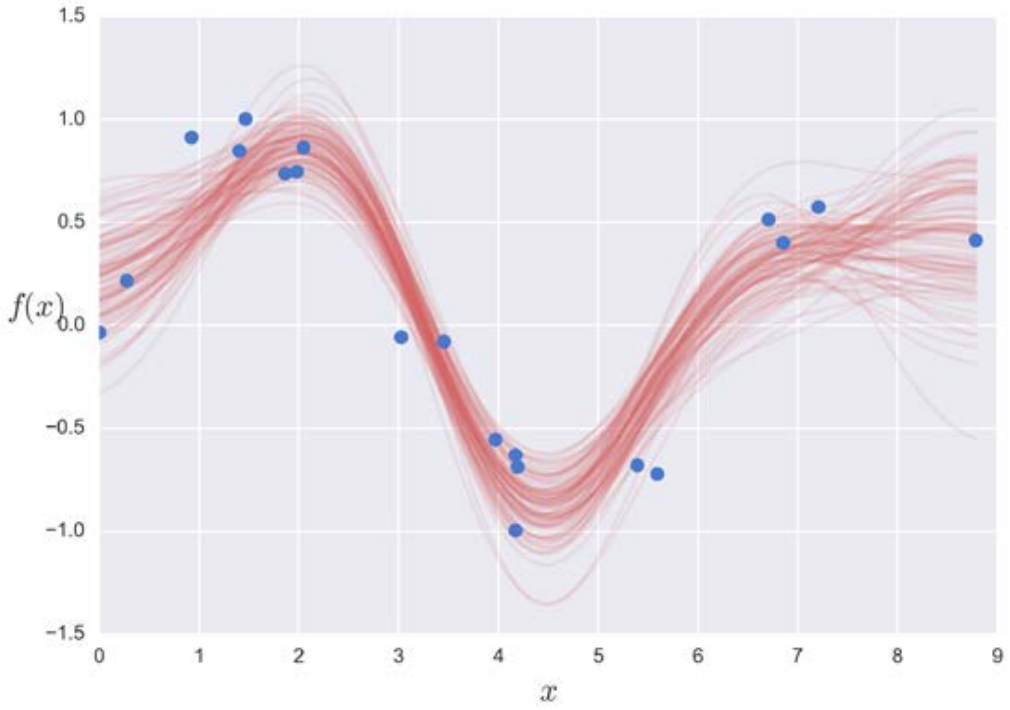
As you may have noticed, the models runs as smooth as a sin function. We are now going to perform a posterior check to see what the model has learned:

```
ppc = pm.sample_ppc(chain, model=kernel_model, samples=100)

plt.plot(x, ppc['yl'].T, 'ro', alpha=0.1)

plt.plot(x, y, 'bo')
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$f(x)$', fontsize=16, rotation=0)
```




The model seems to track the data points very well, let's now check how the model performs for points other than the ones we have observed:

```
new_x = np.linspace(np.floor(x.min()), np.ceil(x.max()), 100)
k = gauss_kernel(new_x, n_knots)
gamma_pred = chain['gamma']
for i in range(100):
    idx = np.random.randint(0, len(gamma_pred))
    y_pred = np.math.dot(gamma_pred[idx], k)
    plt.plot(new_x, y_pred, 'r-', alpha=0.1)
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$f(x)$', fontsize=16, rotation=0)
plt.plot(x, y, 'bo');
```




We use blue dots for the data and red lines for the fitted curves. Now you may want to explore the effect of changing the bandwidth and changing the number of knots. See *Exercise 1* for this, and to also explore the effect of fitting another type of function see *Exercise 2*.

# Overfitting and priors

An obvious concern when working with **kernelized** models is how to choose the number of knots and the location. One alternative is using the datapoints as the knots, that is, to place a Gaussian on top of each data point. You may remember that we have already mentioned that this is how KDE plots are done.

Another option is to let the number and location of knots be determined during the modeling. This may require some special computational methods that do not generalize well, or at least are not easy to work with. Another option is to use variable selection; the idea is to introduce a model index variable. The size of the vector should be the same as the number of $\gamma$ coefficient, the elements of this vector can take only two values, either `zero` or `one`. In such a way, we can turn the coefficient off and on in our model. One problem with this approach is that it only works for low dimensional problems since the number of possible combinations of indexes grows as $2^H$, where $H$ is the number of coefficients. One alternative is to use regularizing priors. We want priors concentrated around zero in order to push the $\gamma$ coefficients towards zero and with long tails to avoid pushing too much. One such prior is the Cauchy distribution another option is the Laplace. Remember that we have already discussed this in *Chapter 6*, *Model Comparison* in the context of Regularization priors, Ridge and Lasso regression.

# Gaussian processes

We just saw a brief introduction on how to use kernels to build statistical models to describe arbitrary functions. Maybe the kernelized regression sounds a little bit like ad hoc trickery and the idea of having to somehow specify the number and distribution of a set of knots is a little problematic. Now we are going to see an alternative way to use kernels by doing inference directly in the function space. This alternative is mathematically and computationally more appealing and is based on using **Gaussian processes**.

Before introducing Gaussian processes let's think about what a function is? We may think of a function as mapping from a set of inputs to a set of outputs. One way to learn this mapping is by restricting it to a line, as we did in *Chapter 4*, *Understanding and Predicting Data with Linear Regression Models*, and then to use the Bayesian machinery to infer the plausible values of the parameters controlling that line. But suppose we do not want to restrict our model to a line, we want to infer any possible function. As usual in Bayesian statistics, when we do not know a quantity we put a prior over it. So if we do not know which function could be a good model for our data, we need to find a prior over functions. Interestingly, such a prior is a **Multivariate Gaussian**, well in fact it is something similar, but play along with me for a moment. We can use a Multivariate Gaussian to describe a function in a very broad (but useful) way. We are going to say that for every $x_i$ value there is a $y_i$ value that is Gaussian distributed with an unknown mean and unknown standard deviation. In this way, if our **x** vector has length *n* we will have a n-multivariate Gaussian distribution.

For a real valued function these sets of inputs $\boldsymbol{x}$ and outputs $\boldsymbol{y}$ are indeed infinite; the reason is that between two points there is an infinite number of other points. So, at least in principle we should need an infinite-multivariate Gaussian. And that mathematical object is known as **Gaussian Process (GP)** and is parameterized with a **mean function** and a **covariance function**:

$$f(\boldsymbol{x}) \sim GP\left(\mu(\boldsymbol{x}), K(\boldsymbol{x}, \boldsymbol{x}')\right)$$

A formal definition says that for a GP, every point in a continuous space has associated a normally distributed variable, and the GP is the joint distribution of those infinitely many random variables. The mean function is an infinite vector of mean values. A covariance function is an infinite covariance matrix and as we will see it is a way to effectively model how a change in $\boldsymbol{x}$ is related to a change in $\boldsymbol{y}$.

To summarize, in previous chapters, we learned how to estimate $p(\boldsymbol{y}|\boldsymbol{x})$, for example, in linear regression we assume $y = f(\boldsymbol{x}) + \in$, where $f$ is a linear model and we proceed to estimate parameters of that linear model, that is, we end up estimating $p(\theta|\boldsymbol{x})$. Using a GP we can instead estimate $p(f|\boldsymbol{x})$. Later we will see that we still need to estimate parameters, but conceptually it is a very good idea to think we are working directly with functions.

# Building the covariance matrix

In practice, and although it is not really necessary, the mean function of the GP is usually set to zero, and hence the entire behavior of the GP is controlled by the covariance function. So let's focus on how to build the covariance function.

## Sampling from a GP prior

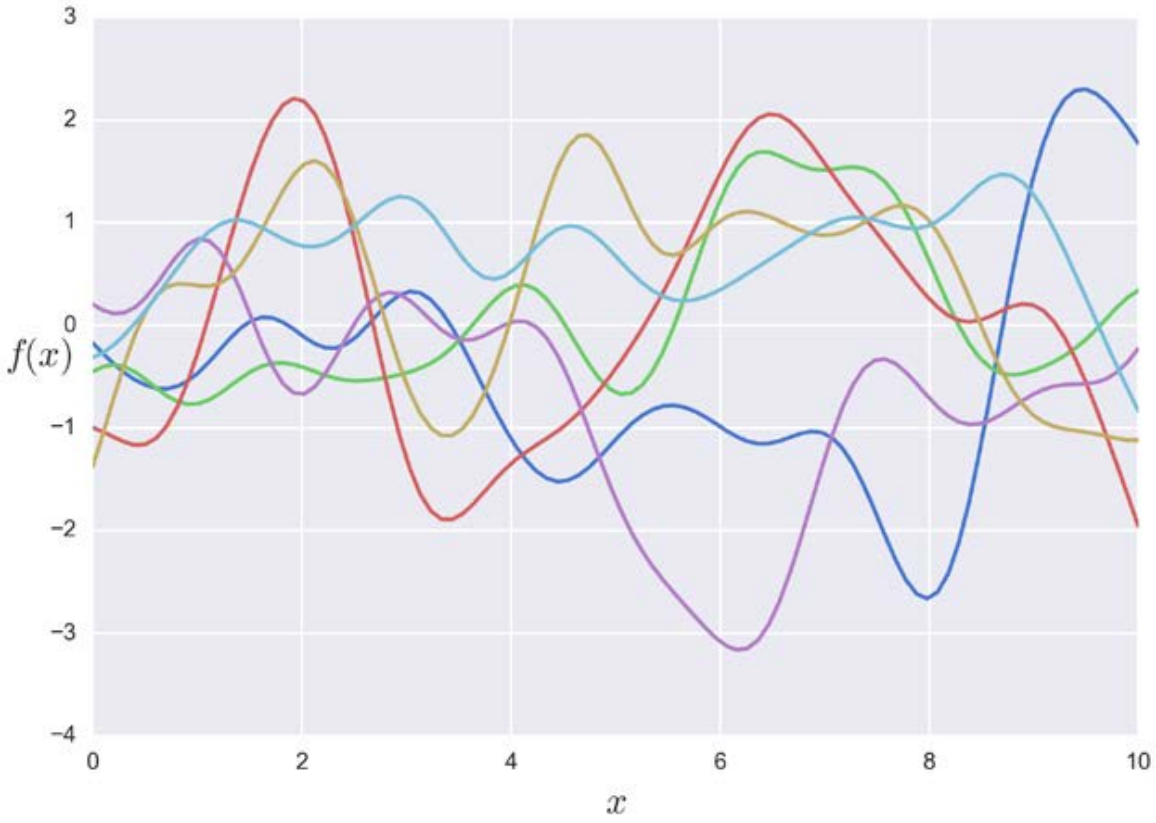
The GP concept is like a mental scaffold, in practice we do not really directly use this infinite object; instead we *collapse* the infinite GP prior to a finite multivariate Gaussian. Mathematically, this is done by marginalizing over the infinitely unobserved dimensions that are left out of our model. If we do so, we will end up with a multivariate Gaussian distribution. This follows from the definition of Gaussian processes as a collection of random variables of which any finite subset have a joint Gaussian distribution. Thus in practice, we only evaluate the GP at the points where we have data, so we end up working with a Multivariate Gaussian distribution with as many dimensions as data points! Hence, for a zero mean function, we will obtain:

$$f\left(\boldsymbol{x}\right) \sim MvNormal\left(\mu = \left[0 \ldots 0\right], K\left(\boldsymbol{x}, \boldsymbol{x}'\right)\right)$$

Now that we have tamed the infinitely headed creature, let's continue with the definition of the covariance matrix, notice that we have written the covariance matrix as $K(\boldsymbol{x}, \boldsymbol{x}')$. This is intentionally the same notation used for the kernelized regression example since we are in fact using a kernel to build the covariance function. The covariance function is going to describe how we expected $\boldsymbol{y}$ to vary when $\boldsymbol{x}$ varies. As we already saw for the kernelized regression, using a Gaussian kernel is a reasonable assumption and it is equivalent to saying that a small variation of $x_i$ will give (on average) a small variation of $y_i$ and a large variation of $x_i$ will give (on average) a large variation of $y_i$.

To gain intuition into what a GP prior is, we are going to sample from it. In the following figure, we have plotted six arbitrary functions (or realizations) from the GP prior. Notice that while we have plotted the realizations as continuous functions, in fact we only have values of $f(\boldsymbol{x}_*)$ at a given set of $\boldsymbol{x}_*$ test points. Of course, given our smooth assumption and the impossibility of actually computing an infinite set of values, it makes total sense to *connect the dots* of the individual realizations into a continuous function. We just need to provide enough test points to accurately reflect the shape of the function. I hope you find this *natural*; after all, this is the same old trick we always use in practice to compute and plot functions using computers.

```
np.random.seed(1)
test_points = np.linspace(0, 10, 100)
cov = np.exp(-squared_distance(test_points, test_points))
  plt.plot(test_points, stats.multivariate_normal.rvs(cov=cov,
size=6).T)

plt.xlabel('$x$', fontsize=16)
plt.ylabel('$f(x)$', fontsize=16, rotation=0)
```




As you can see in the preceding figure our GP prior, with a Gaussian kernel, implies a wide variety of smooth functions centered around 0. See *Exercise 3*.

# Using a parameterized kernel

In order to learn about our unknown function from our data, we define the covariance matrix in terms of a parameterized kernel. We will call the parameters of the kernel, **hyper-parameters**. The reason is two-fold:

- They are parameters for the GP prior
- The name helps to emphasize that we are working with a non-parametric method

By learning the hyper-parameters of the GP prior we hope to approximate the unknown function.

As we have already mentioned, there are many options for kernels, a pretty common one is the Gaussian kernel. Earlier in this chapter we saw a version of it that was parameterized using one parameter, the bandwidth. Now, we are going to introduce a version with two more parameters. We can write this kernel as follows:

$$K_{ij} = \begin{cases} \eta \exp\left(-\rho D\right) & if\ i \neq j \\ \eta + \sigma & if\ i = j \end{cases}$$

Where $D$ is the SED, that is, the quantity $\|x - x'\|^2$. $\eta$ is a parameter controlling the vertical scale, that is, it allows the covariance matrix to model larger or smaller values of $f(x)$. The next parameter is $\rho$ and it is just the bandwidth. As we have already seen, $\rho$ is in charge of controlling the smoothness of the function. Finally, we have $\sigma$, which captures the noise in the data.

Let's talk a little bit about $\sigma$ and why we are using a different expression whether $i$ and $j$ are equal or not.

In some settings, like when performing interpolation, we want a model that for each observed $x_i$ point returns the observed value of $f(x_i)$ without any uncertainty. In other settings, like in all the examples through this book, we instead want to get an estimation of the uncertainty for $f(x_i)$, and thus we want a value that is somewhat close to our output observed values, but not exactly equal to them. As we have already written several times, we have the following:

$$y = f(\boldsymbol{x}) + \in$$

Where the error term is modeled as $\in \sim N(0, \sigma)$

Thus the covariance matrix should be constructed in order to properly take into account noisy data. In such a case, we will have the following:

$$\text{cov}\left[ y_i, y_j \right] = k\left( x_i, x_j \right) + \sigma\delta_{ij}$$

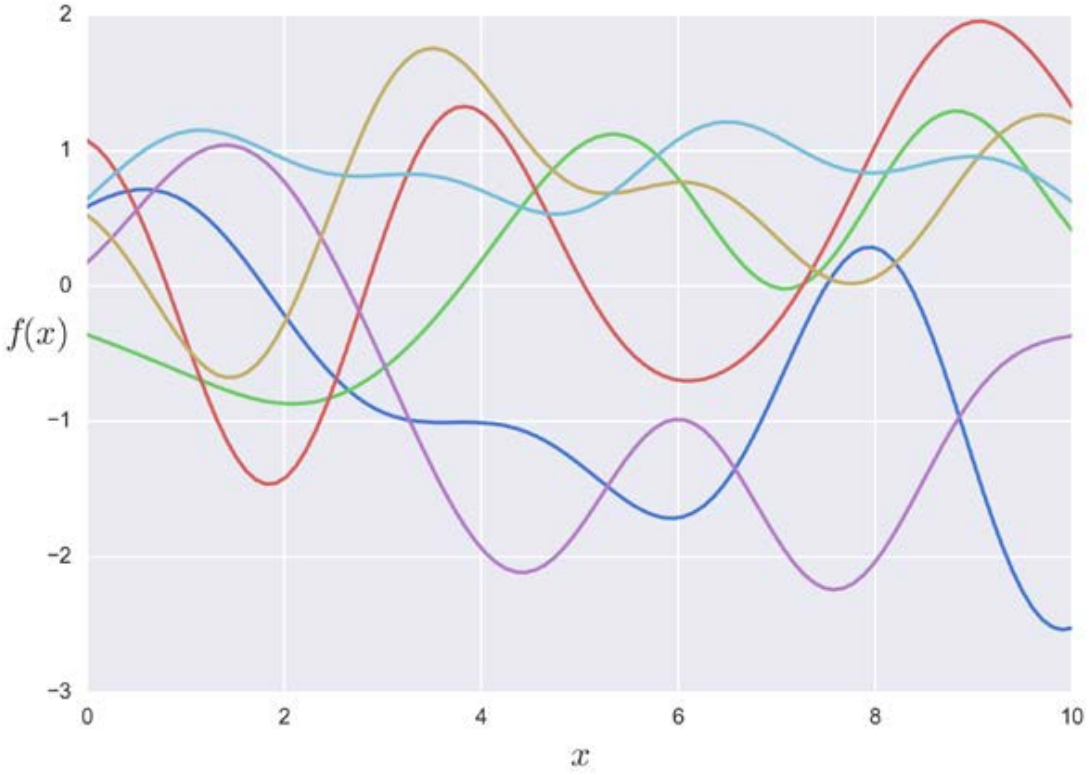
Where:

$$\delta_{ij} = \begin{cases} 0 \; if \; i \neq j \\ 1 \; if \; i = j \end{cases}$$

$\delta_{ij}$ is known as the **Kronecker delta**. That is, we model the noise in the data by allowing the diagonal of the covariance matrix to not be exactly *1*. Instead, we estimate the diagonal values from the data. This allows us to add a *jitter* term to the model, we need to add this term because the assumption conveyed by the kernel is that, the closer two inputs are the closer the outputs will be, at the extreme of two points being equal, their outputs should be equal. Adding a jitter term allows us to capture the uncertainty we have around the observed data points.

To get a better feeling for the meaning of each hyper-parameter, let's do a plot using the expanded version of the Gaussian kernel. Feel free to experiment with different values for the hyper-parameters:

```
np.random.seed(1)
eta = 1.5
rho = 0.2
sigma = 0.007

D = squared_distance(test_points, test_points)

cov = eta * np.exp(-rho * D)
diag = eta + sigma

np.fill_diagonal(cov, diag)

for i in range(6):
    plt.plot(test_points, stats.multivariate_normal.rvs(cov=cov))
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$f(x)$', fontsize=16, rotation=0)
```




# Making predictions from a GP

The next step is to be able to make predictions from the GP. One of the advantages of GP is that it is analytically tractable. If we combine a GP prior with a Gaussian likelihood we get a GP posterior. That is, if we apply the rules for conditioning Gaussians at a set of test points we obtain the following expression for the posterior predictive density:

$$p\left(f\left(\boldsymbol{X}_*\right)|\,\boldsymbol{X}_*,\boldsymbol{X},\boldsymbol{y}\right)\sim N\left(\boldsymbol{\mu}_*,\boldsymbol{\Sigma}_*\right)$$

$$\boldsymbol{\mu}=\boldsymbol{K}_*^T\boldsymbol{K}^{-1}y$$

$$\boldsymbol{\Sigma}=\boldsymbol{K}_{**}-\boldsymbol{K}_*^T\boldsymbol{K}^{-1}\boldsymbol{K}_*$$

We are computing the values of the unknown function at a (yet) unseen point $f(\boldsymbol{x}_*)$, given our data $\boldsymbol{X}$ and $\boldsymbol{y}$, and a set of test points $\boldsymbol{x}_*$. Notice we are using the symbol $_*$ to indicate the computations that are over the test points, where:

$$\boldsymbol{K} = K(\boldsymbol{X}, \boldsymbol{X})$$
$$\boldsymbol{K}_{**} = K(\boldsymbol{X}_*, \boldsymbol{X}_*)$$
$$\boldsymbol{K}_{*} = K(\boldsymbol{X}, \boldsymbol{X}_*)$$

At first, this expression can be a little bit intimidating; in a moment we will see how to code this expression and things will become, hopefully, clearer. But now let's try to get some intuition out of this expression.

$K_{**}$ is the covariance of $\boldsymbol{x}_*$ to itself, so it is just the variance of $\boldsymbol{x}_*$. That is, the variance of the test points or equivalently the variance of the prior. Notice that $\Sigma$ is equal to $K_{**}$ minus a quantity. This expression tells us that using data we are able to reduce the prior variance by this exact quantity . Besides the exact value of this quantity we can see the following, that for a $\boldsymbol{x}_*$ test point far away from a data point $x_i$, the kernel returns a value close to zero, and thus this quantity will be close to zero; as a consequence we will not be able to reduce the prior variance too much. In other words, evaluating the function far away from the data does not help us to reduce the uncertainty. Thus, the inference is based on local features of the data around the test points.

If you want to know more about the mathematical properties of Multivariate Gaussians and how the previous expression are derived, please refer to the *Keep reading* section at the end of this chapter. There, I have added some references to more advanced material that you will certainly find interesting. For our current discussion, we take the previous expression for the posterior predictive as given. The important take-home message is that we can use it to get samples from the GP posterior. Notice that having this analytical expression to describe the unknown function (once we have learned the correct parameters) is very valuable. One problem is that the computation involves inverting a matrix and inverting matrices are $O(n^3)$. If you are not familiar with this expression, we can loosely say that this type of operation is a slow one, thus in practice we cannot use GPs for more than a few thousand data points. Anyway, for those cases approximations exist to speed up computations, but we are not going to explore them here. Also, in practice, directly inverting matrices can lead to numerical problems and instabilities and thus alternatives are preferred, such as using the **Cholesky decomposition** to compute the mean and covariance posterior functions. A Cholesky decomposition is like the square root for scalars, which we are all familiar with, but for matrices.

Before moving on to the examples using both the Cholesky decomposition and directly inverting the matrices, we have to make an observation. If inverting matrices is problematic, why do we define GP in terms of covariance matrices and not in terms of the inverse of the covariance matrices? Well, when using a covariance matrix we are saying that the computation of a subset of it is independent of the computation of the rest and is also independent of the computation of yet unseen points. But for the inverse covariance matrix the computation of a subset of points will depend on whether we observed those other points or not. Only the use of the covariance matrix (and not of its inverse)will lead us to the definition of Gaussian processes as a consistent collection of random variables.

To summarize, in order to use a GP in a fully Bayesian setting to approximate a function, we need to:

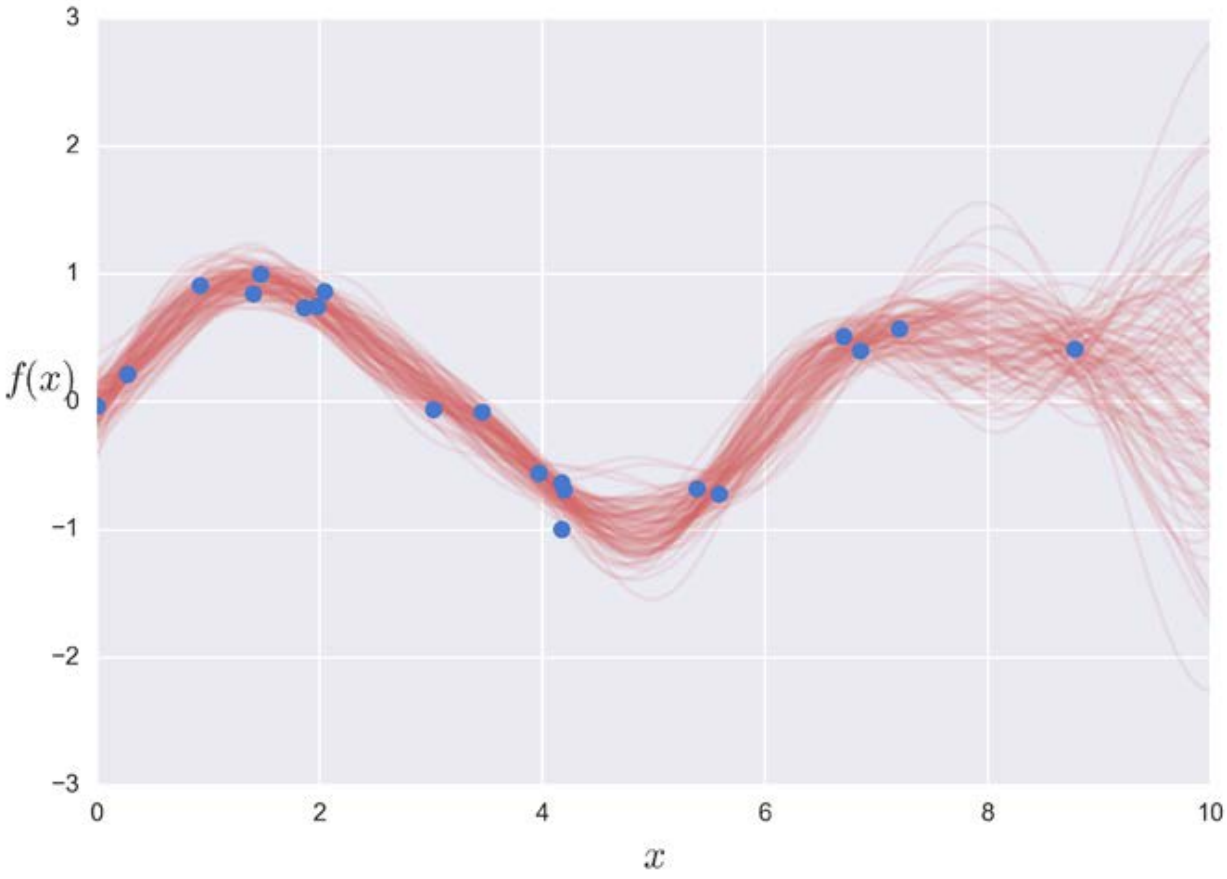
- Choose a kernel to build the covariance matrix of a multivariate distribution
- Use Bayesian statistics to infer the values of the parameters in the kernel
- Compute (analytically) the mean and standard deviation at each test point

Notice, that in fact we never compute an actual GP; we just use the mathematical concepts as a guide to be sure we are doing something reasonable. But in practice, all the computations are done using Multivariate Gaussians.

After this lengthy theoretical dissertation comes the moment everyone was expecting, when we put all these ideas into code. First we are going to assume we know the values of the kernel's parameters and we are going to put in code the analytical expression for the posterior. The values of the hyper-parameters and the values of the `test_points` are the same as the ones defined earlier. And the data is the same as we used for the kernelized regression example:

```
np.random.seed(1)

K_oo = eta * np.exp(-rho * D)

D_x = squared_distance(x, x)
K = eta * np.exp(-rho * D_x)
diag_x = eta + sigma
np.fill_diagonal(K, diag_x)

D_off_diag = squared_distance(x, test_points)
K_o = eta * np.exp(-rho * D_off_diag)

mu_post = np.math.dot(np.math.dot(K_o, np.linalg.inv(K)), y)
```

```
SIGMA_post = K_oo - np.math.dot(np.math.dot(K_o, np.linalg.inv(K)),
K_o.T)

for i in range(100):
    fx = stats.multivariate_normal.rvs(mean=mu_post, cov=SIGMA_post)
    plt.plot(test_points, fx, 'r-', alpha=0.1)

plt.plot(x, y, 'o')

plt.xlabel('$x$', fontsize=16)
plt.ylabel('$f(x)$', fontsize=16, rotation=0)
```

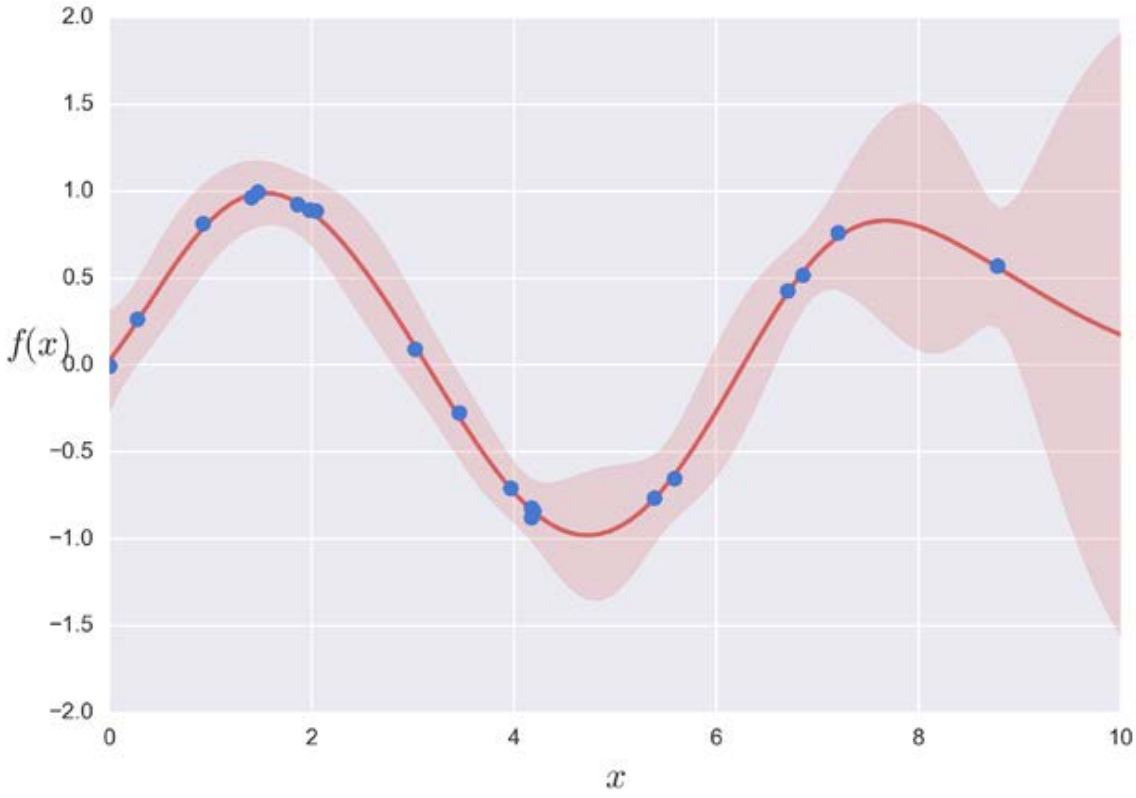



We have modeled the uncertainty by superimposing several realizations from the GP posterior using red lines. Notice how the uncertainty is smaller closer to the data points, and larger between the last two points, and even larger at the right extreme where there are no more points after ~*9*.

Now we are going to re-implement the preceding computations, but this time using the Cholesky decomposition. The following code was adapted from one created by *Nando de Freitas*, to teach his Machine Learning course (`goo.gl/byM3SE`). In the code, `N` is the number of data points, and `n` is the number of test points:

```
np.random.seed(1)
eta = 1
rho = 0.5
sigma = 0.03

f = lambda x: np.sin(x).flatten()

def kernel(a, b):
    """ GP squared exponential kernel """
    sqdist = np.sum(a**2,1).reshape(-1,1) + np.sum(b**2,1) - 2*np.
dot(a, b.T)
    return eta * np.exp(- rho * sqdist)

N = 20
n = 100

X = np.random.uniform(0, 10, size=(N,1))
y = f(X) + sigma * np.random.randn(N)

K = kernel(X, X)
L = np.linalg.cholesky(K + sigma * np.eye(N))

test_points = np.linspace(0, 10, n).reshape(-1,1)

Lk = np.linalg.solve(L, kernel(X, test_points))
mu = np.dot(Lk.T, np.linalg.solve(L, y))

K_ = kernel(Xtest, Xtest)
sd_pred = (np.diag(K_) - np.sum(Lk**2, axis=0))**0.5

plt.fill_between(test_points.flat, mu-2*s, mu+2*s, color="r",
alpha=0.2)
```

```
plt.plot(test_points, mu, 'r', lw=2)
plt.plot(x, y, 'o')
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$f(x)$', fontsize=16, rotation=0)
```




Now we have plotted the data as blue dots, the mean function as a red line, and the uncertainty as a semi-transparent red band. The uncertainty is represented as two standard deviations from the mean.

# Implementing a GP using PyMC3

To summarize, we have a *GP* prior:

$$f(x) \sim GP\left(\mu = [0 \ldots 0], k(x, x')\right)$$

A Gaussian likelihood:

$$p\left(y \mid x, f(x)\right) \sim N\left(\boldsymbol{f}, \sigma^2 I\right)$$

And finally, a *GP* posterior:

$$p\left(f(\boldsymbol{x}) \mid \boldsymbol{x}, \boldsymbol{y}\right) \sim GP\left(\mu_{post}, \Sigma_{post}\right)$$

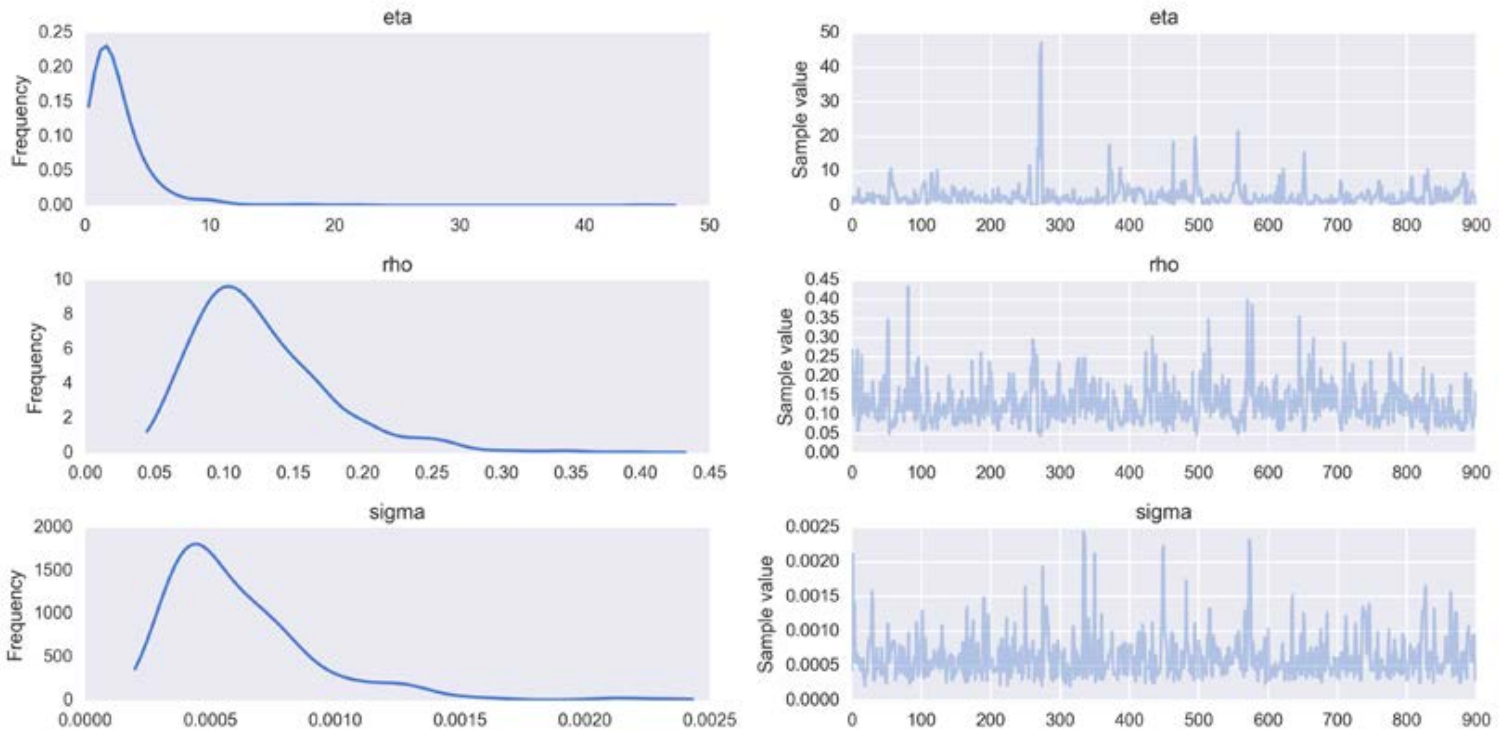
Remember that, in practice, we use multivariate Gaussians, since a GP is a multivariate Gaussian when evaluated at a finite set of points.

We are going to use the Bayesian machinery to learn the hyper-parameters of the covariance matrix. At this point, you will see that, while this is really simple using PyMC3, there is a little overhead in the coding right now. You will see that at this point we have to *manually* invert matrices (or compute Cholesky decomposition). Chances are high that in the near future PyMC3 will come with a specialized GP module to make building GP models easier. Maybe as you are reading this such GP module is already available!

The following model was adapted from the Stan repository by *Chris Fonnesbeck* (BDFL of PyMC3):

```
with pm.Model() as GP:
    mu = np.zeros(N)
    eta = pm.HalfCauchy('eta', 5)
    rho = pm.HalfCauchy('rho', 5)
    sigma = pm.HalfCauchy('sigma', 5)

    D = squared_distance(x, x)

    K = tt.fill_diagonal(eta * pm.math.exp(-rho * D), eta + sigma)

    obs = pm.MvNormal('obs', mu, tt.nlinalg.matrix_inverse(K),
observed=y)

    test_points = np.linspace(0, 10, 100)
    D_pred = squared_distance(test_points, test_points)
    D_off_diag = squared_distance(x, test_points)

    K_oo = eta * pm.math.exp(-rho * D_pred)
    K_o = eta * pm.math.exp(-rho * D_off_diag)

    mu_post = pm.Deterministic('mu_post', pm.math.dot(pm.math.dot(K_o,
tt.nlinalg.matrix_inverse(K)), y))
    SIGMA_post = pm.Deterministic('SIGMA_post', K_oo - pm.math.dot(pm.
math.dot(K_o, tt.nlinalg.matrix_inverse(K)), K_o.T))

    start = pm.find_MAP()
    trace = pm.sample(1000, start=start)
```

```
varnames = ['eta', 'rho', 'sigma']
chain = trace[100:]
pm.traceplot(chain, varnames)
```




If you pay attention, you will notice that the mean of the estimated parameters, `eta`, `rho`, and `sigma`, were the ones we were using in the previous examples. And that explains why we were getting such a good fit; the hyper-parameters for those examples were not taken out of my head!

```
pm.df_summary(chain, varnames).round(4)
```

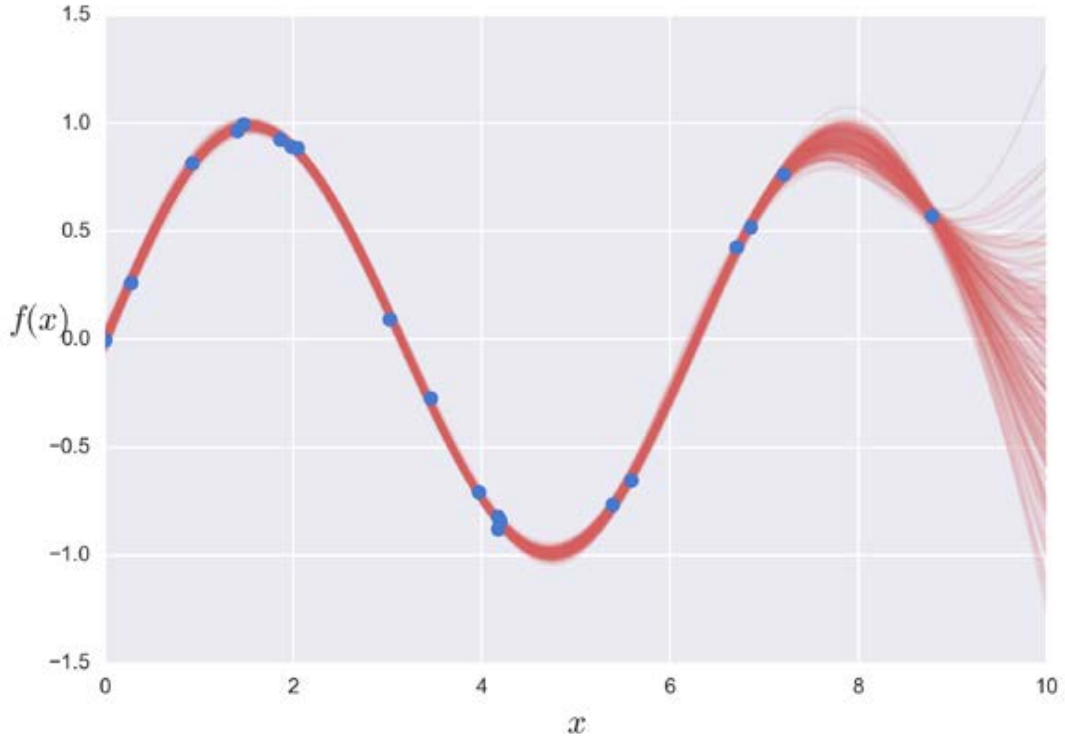
| | **mean** | **sd** | **mc_ error** | **hpd_2.5** | **hpd_97.5** |
|---|---|---|---|---|---|
| eta | 2.5798 | 2.5296 | 0.1587 | 0.1757 | 6.3445 |
| rho | 0.1288 | 0.0485 | 0.0027 | 0.0589 | 0.2290 |
| sigma | 0.0006 | 0.0003 | 0.0000 | 0.0002 | 0.0012 |

# Posterior predictive checks

Now we are going to plot the data together with realizations taken from the GP posterior with the estimated hyper-parameters. Notice we are including the uncertainty in the hyper-parameters and not just their mean values:

```
y_pred = [np.random.multivariate_normal(m, S) for m,S in
zip(chain['mu_post'][::5], chain['SIGMA_post'][::5])]


for yp in y_pred:
```

```
        plt.plot(test_points, yp, 'r-', alpha=0.1)

    plt.plot(x, y, 'bo')
    plt.xlabel('$x$', fontsize=16)
    plt.ylabel('$f(x)$', fontsize=16, rotation=0)
```




# Periodic kernel

In the previous figure you may have noticed that we were able to closely match the underlying `sin` function, but the model shows great uncertainty between ~*9* and ~*10*, where there is no more data to constrain it. One problem of our model is that our data was generated by a periodic function but our kernel makes no assumption of periodicity. When we know/suspect our data could be periodic we should instead use a periodic kernel. One example of such a periodic kernel is:

$$Kp\left(\boldsymbol{x},\boldsymbol{x}'\right)=\exp\left(-\frac{\sin^2\left(\frac{\boldsymbol{x}-\boldsymbol{x}'}{2}\right)}{w}\right)$$

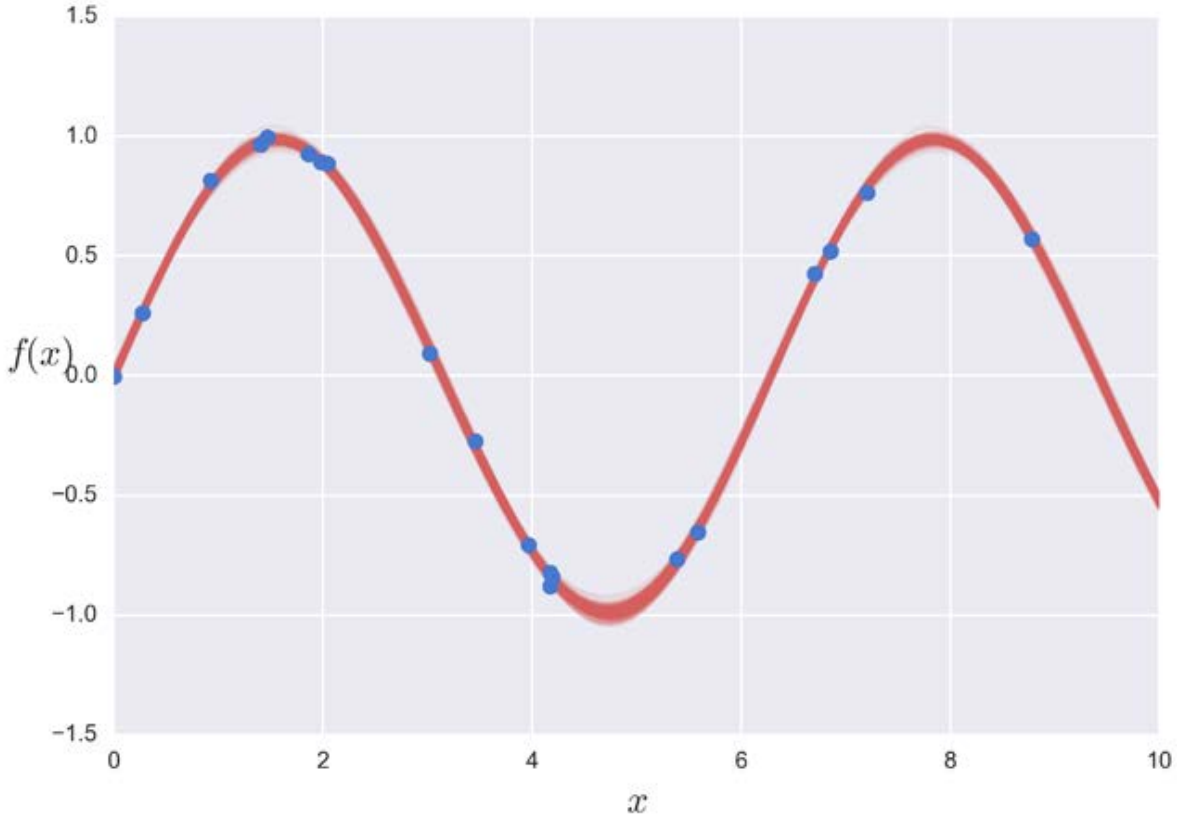
Notice that the main difference with respect to the Gaussian kernel is the inclusion of the `sin` function.

We can use the exact same code as before. The only difference is that now we have to define a periodic function instead of the `squared_distance`, that is:

```
periodic = lambda x, y: np.array([[np.sin((x[i] - y[j])/2)**2 for i in
range(len(x))] for j in range(len(y))])
```

and in the model we need to replace `squared_distance` with the `periodic` function.

After running the model you should get something like:



# Summary

We began this chapter by learning about non-parametric statistics in a Bayesian setting and how we can represent statistical problems through the use of kernel functions, as an example, we used a kernelized version of linear regression to model non-linear responses. Then we moved on to an alternative way of building and conceptualizing kernel methods using Gaussian processes.

A Gaussian process is a generalization of the multivariate Gaussian distribution to infinitively many dimensions and is fully specified by a mean function and a covariance function. Since we can conceptually think of functions as infinitively long vectors, we can use Gaussian processes as priors for functions. In practice, we work 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 hyper-parameters, we ended up learning about arbitrary complex and unknown functions.

In this chapter we have just seen a short introduction to GPs. There are many topics related to this type of model that remain to learn, such as building a semi-parametric model by using for example, a linear model as a mean function. Or combining two or more kernels to better describe our unknown function or how a GP can be used also for classification tasks, or how GP are related to many other models in statistics and machine learning. Nevertheless, I hope this introduction to GP as well as all the other topics we have covered in the book have served to motivate you to keep using, reading and learning about Bayesian statistics.

# Keep reading

- *Gaussian Processes for Machine Learning* by *Carl Edward Rasmussen* and *Christopher K. I. Williams*
- Chapter 4 and 15 *Machine Learning a Probabilistic Perspective* by *Kevin Murhpy*
- Chapter 11 *Statistical Rethinking*, *Richard McElreath*.
- Chapter 22 *Bayesian Data Analysis, Third Edition Gelman et al*

# Exercises

1. In the kernelized regression example, try changing the number of knots and the bandwidth (one at a time). What is the effect of those changes? Try also using a single knot; what do you observe?
2. Experiment with fitting other functions using kernelized regression. For example $y = np.sin(x) + x**0.7$ or $y = x$. Using these functions changes the number of data points and parameters like in Exercise 1
3. In the example where we sample from the GP prior increase the number or realizations, by replacing:

   ```
   plt.plot(test_points, stats.multivariate_normal.rvs(cov=cov,
   size=6).T)
   ```

   with

   ```
   plt.plot(test_points, stats.multivariate_normal.rvs(cov=cov,
   size=1000).T, alpha=0.05, color='b')
   ```

   How does the GP prior look? Do you see that $f(x)$ is distributed as a Gaussian centered at $0$ and standard deviation $1$?

4. For the GP posterior using the Gaussian kernel, try defining `test_points` outside the interval $[0, 10]$. What happened outside the `data interval`? What does this tell us about extrapolating results (especially for non-linear functions)?
5. Repeat Exercise 4 this time using the Periodic kernel. What are your conclusions now?

# Index

# E



# F



# G



# H



# I



# K

## L



## M



## N

# O



# P



# Q



# R

## S



## T



## V



## W



## Z