BAYESIAN METHODS FOR Hackers

Probabilistic Programming and Bayesian Inference

CAMERON DAVIDSON-PILON

Bayesian Methods for Hackers

The Addison-Wesley Data and Analytics Series

Addison-Wesley

Visit **informit.com/awdataseries** for a complete list of available publications.

The **Addison-Wesley Data and Analytics Series** provides readers with practical knowledge for solving problems and answering questions with data. Titles in this series primarily focus on three areas:

1. **Infrastructure:** how to store, move, and manage data
2. **Algorithms:** how to mine intelligence or make predictions based on data
3. **Visualizations:** how to represent data and insights in a meaningful and compelling way

The series aims to tie all three of these areas together to help the reader build end-to-end systems for fighting spam; making recommendations; building personalization; detecting trends, patterns, or problems; and gaining insight from the data exhaust of systems and user interactions.

Make sure to connect with us!
informit.com/socialconnect

Bayesian Methods for Hackers

Probabilistic Programming and Bayesian Inference

Cameron Davidson-Pilon

Addison-Wesley

New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Davidson-Pilon, Cameron.
Bayesian methods for hackers : probabilistic programming and bayesian inference / Cameron Davidson-Pilon.
pages cm
Includes bibliographical references and index.
ISBN 978-0-13-390283-9 (pbk.: alk. paper)
1. Penetration testing (Computer security)–Mathematics. 2. Bayesian statistical decision theory.
3. Soft computing. I. Title.
QA76.9.A25D376 2015
006.3–dc23

2015017249

ISBN-13: 978-0-13-390283-9
ISBN-10: 0-13-390283-8
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, October 2015

❖

This book is dedicated to many important relationships: my parents, my brothers, and my closest friends. Second to them, it is devoted to the open-source community, whose work we consume every day without knowing.

❖

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

Bayesian methods are one of many in a modern data scientist's toolkit. They can be used to solve problems in prediction, classification, spam detection, ranking, inference, and many other tasks. However, most of the material out there on Bayesian statistics and inference focuses on the mathematical details while giving little attention to the more pragmatic engineering considerations. That's why I'm very pleased to have this book joining the series, bringing a much needed introduction to Bayesian methods targeted at practitioners.

Cameron's knowledge of the topic and his focus on tying things back to tangible examples make this book a great introduction for data scientists or regular programmers looking to learn about Bayesian methods. This book is filled with examples, figures, and working Python code that make it easy to get started solving actual problems. If you're new to data science, Bayesian methods, or new to data science with Python, this book will be an invaluable resource to get you started.

—Paul Dix
Series Editor

This page intentionally left blank

Preface

The Bayesian method is the natural approach to inference, yet it is hidden from readers behind chapters of slow, mathematical analysis. The typical text on Bayesian inference involves two to three chapters on probability theory, then enters into what Bayesian inference is. Unfortunately, due to the mathematical intractability of most Bayesian models, the reader is only shown simple, artificial examples. This can leave the user with a "So what?" feeling about Bayesian inference. In fact, this was my own prior opinion.

After some recent success of Bayesian methods in machine-learning competitions, I decided to investigate the subject again. Even with my mathematical background, it took me three straight days of reading examples and trying to put the pieces together to understand the methods. There was simply not enough literature bridging theory to practice. The problem with my misunderstanding was the disconnect between Bayesian mathematics and probabilistic programming. That being said, I suffered then so the reader would not have to now. This book attempts to bridge the gap.

If Bayesian inference is the destination, then mathematical analysis is a particular path toward it. On the other hand, computing power is cheap enough that we can afford to take an alternate route via probabilistic programming. The latter path is much more useful, as it denies the necessity of mathematical intervention at each step; that is, we remove often intractable mathematical analysis as a prerequisite to Bayesian inference. Simply put, this latter computational path proceeds via small, intermediate jumps from beginning to end, whereas the first path proceeds by enormous leaps, often landing far away from our target. Furthermore, without a strong mathematical background, the analysis required by the first path cannot even take place.

Bayesian Methods for Hackers is designed as an introduction to Bayesian inference from a computational/understanding first, and mathematics second, point of view. Of course, as an introductory book, we can only leave it at that: an introductory book. For the mathematically trained, the curiosity this text generates may be cured by other texts designed with mathematical analysis in mind. For the enthusiast with a less mathematical background, or one who is not interested in the mathematics but simply the practice of Bayesian methods, this text should be sufficient and entertaining.

The choice of PyMC as the probabilistic programming language is twofold. First, as of this writing, there is currently no central resource for examples and explanations in the PyMC universe. The official documentation assumes prior knowledge of Bayesian inference and probabilistic programming. We hope this book encourages users at every level to look at PyMC. Second, with recent core developments and popularity of the scientific stack in Python, PyMC is likely to become a core component soon enough.

PyMC does have dependencies to run, namely NumPy and (optionally) SciPy. To not limit the user, the examples in this book will rely only on PyMC, NumPy, SciPy, and matplotlib.

The progression of the book is as follows. Chapter 1 introduces Bayesian inference and its comparison to other inference techniques. We also see, build, and train our first Bayesian model. Chapter 2 focuses on building models with PyMC, with a strong emphasis on examples. Chapter 3 introduces Markov Chain Monte Carlo, a powerful algorithm behind computational inference, and some techniques on debugging your Bayesian model. In Chapter 4, we detour and again visit the issue of sample sizes in inference and explain why understanding sample size is so important. Chapter 5 introduces the powerful idea of loss functions, where we have not a model but a function that connects inference to real-world problems. We revisit the idea of Bayesian priors in Chapter 6, and give good heuristics to picking good priors. Finally, in Chapter 7, we explore how Bayesian inference can be used in A/B testing.

All the datasets used in this text are available online at `https://github.com/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers`.

Acknowledgments

I would like to acknowledge the many people involved in this book. First and foremost, I'd like to acknowledge the contributors to the online version of *Bayesian Methods for Hackers*. Many of these authors submitted contributions (code, ideas, and text) that helped round out this book. Second, I would like to thank the reviewers of this book, Robert Mauriello and Tobi Bosede, for sacrificing their time to peel through the difficult abstractions I can make and for narrowing the contents down for a much more enjoyable read. Finally, I would like to acknowledge my friends and colleagues, who supported me throughout the process.

This page intentionally left blank

About the Author

Cameron Davidson-Pilon has seen many fields of applied mathematics, from evolutionary dynamics of genes and diseases to stochastic modeling of financial prices. His main contributions to the open-source community include *Bayesian Methods for Hackers* and lifelines. Cameron was raised in Guelph, Ontario, but was educated at the University of Waterloo and Independent University of Moscow. He currently lives in Ottawa, Ontario, working with the online commerce leader Shopify.

This page intentionally left blank

This page intentionally left blank

5

Would You Rather Lose an Arm or a Leg?

5.1 Introduction

Statisticians can be a sour bunch. Instead of considering their winnings, they only measure how much they have lost. In fact, they consider their wins to be *negative losses*. But what's interesting is how they measure their losses.

For example, consider the following:

> A meteorologist is predicting the probability of a hurricane striking his city. He estimates, with 95% confidence, that the probability of it *not* striking is between 99% and 100%. He is very happy with his precision and advises the city that a major evacuation is unnecessary. Unfortunately, the hurricane does strike and the city is flooded.

This stylized example shows the flaw in using a pure accuracy metric to measure outcomes. Using a measure that emphasizes estimation accuracy, while an appealing and objective thing to do, misses the point of why you are even performing the statistical inference in the first place: results of inference. Furthermore, we'd like a method that stresses the importance of payoffs of decisions, not the accuracy of the estimation alone. Read puts this succinctly: "It is better to be roughly right than precisely wrong."[1]

5.2 Loss Functions

We introduce what statisticians and decision theorists call **loss functions**. A loss function is a function of the true parameter, and an estimate of that parameter

$$L(\theta, \hat{\theta}) = f(\theta, \hat{\theta})$$

The important point of loss functions is that they measure how *bad* our current estimate is: The larger the loss, the worse the estimate is according to the loss function. A simple, and very common, example of a loss function is the **squared-error loss**, a type of loss function that increases quadratically with the difference, used in estimators like linear regression, calculation of unbiased statistics, and many areas of machine learning

$$L(\theta, \hat{\theta}) = (\theta - \hat{\theta})^2$$

The squared-error loss function is used in estimators like linear regression, calculation of unbiased statistics, and many areas of machine learning. We can also consider an asymmetric squared-error loss function, something like:

$$L(\theta, \hat{\theta}) = \begin{cases} (\theta - \hat{\theta})^2 & \hat{\theta} < \theta \\ \\ c(\theta - \hat{\theta})^2 & \hat{\theta} \geq \theta, \quad 0 < c < 1 \end{cases}$$

which represents that estimating a value larger than the true estimate is preferable to estimating a value that is smaller. A situation where this might be useful is in estimating Web traffic for the next month, where an overestimated outlook is preferred so as to avoid an underallocation of server resources.

A negative property about the squared-error loss is that it puts a disproportionate emphasis on large outliers. This is because the loss increases quadratically, and not linearly, as the estimate moves away. That is, the penalty of being 3 units away is much less than being 5 units away, but the penalty is not much greater than being 1 unit away, though in both cases the magnitude of difference is the same:

$$\frac{1^2}{3^2} < \frac{3^2}{5^2}, \quad \text{although} \quad 3 - 1 = 5 - 3$$

This loss function implies that large errors are very bad. A more robust loss function that increases linearly with the difference is the **absolute-loss**, a type of loss function that increases linearly with the difference, often used in machine learning and robust statistics.

$$L(\theta, \hat{\theta}) = |\theta - \hat{\theta}|$$

Other popular loss functions include the following.

- $L(\theta, \hat{\theta}) = \mathbf{1}_{\hat{\theta} \neq \theta}$ is the *zero-one loss* often used in machine-learning classification algorithms.
- $L(\theta, \hat{\theta}) = -\hat{\theta} \log(\theta) - (1 - \hat{\theta}) \log(1 - \theta), \quad \hat{\theta} \in 0, 1, \ \theta \in [0, 1]$, called the *log-loss*, is also used in machine learning.

Historically, loss functions have been motivated from (1) mathematical ease and (2) their robustness to application (that is, they are objective measures of loss). The first motivation has really held back the full breadth of loss functions. With computers being agnostic to mathematical convenience, we are free to design our own loss functions, which we take full advantage of later in this chapter.

With respect to the second motivation, the above loss functions are indeed objective in that they are most often a function of the difference between estimate and true parameter,

independent of positivity or negativity, or payoff of choosing that estimate. This last point—its independence of payoff—causes quite pathological results, though. Consider our hurricane example: The statistician equivalently predicted that the probability of the hurricane striking was between 0% and 1%. But if he had ignored being precise and instead focused on outcomes (99% chance of no flood, 1% chance of flood), he might have advised differently.

By shifting our focus from trying to be incredibly precise about parameter estimation to focusing on the outcomes of our parameter estimation, we can customize our estimates to be optimized for our application. This requires us to design new loss functions that reflect our goals and outcomes. Some examples of more interesting loss functions include the following.

- $L(\theta, \hat{\theta}) = \frac{|\theta - \hat{\theta}|}{\theta(1-\theta)}, \;\; \hat{\theta}, \theta \in [0, 1]$ emphasizes an estimate closer to 0 or 1, since if the true value θ is near 0 or 1, the loss will be very large unless $\hat{\theta}$ is similarly close to 0 or 1. This loss function might be used by a political pundit who's job requires him or her to give confident "Yes/No" answers. This loss reflects that if the true parameter is close to 1 (for example, if a political outcome is very likely to occur), he or she would want to strongly agree so as to not look like a skeptic.
- $L(\theta, \hat{\theta}) = 1 - e^{-(\theta - \hat{\theta})^2}$ is bounded between 0 and 1 and reflects that the user is indifferent to sufficiently-far-away estimates. It is similar to the zero-one loss, but not quite as penalizing to estimates that are close to the true parameter.
- Complicated non-linear loss functions can programmed:

```
def loss(true_value, estimate):
    if estimate*true_value > 0:
        return abs(estimate - true_value)
    else:
       return abs(estimate)*(estimate - true_value)**2
```

- Another example in everyday life is the loss function that weather forecasters use. Weather forecasters have an incentive to report accurately on the probability of rain, but also to err on the side of suggesting rain. Why is this? People much prefer to prepare for rain, even when it may not occur, than to be rained on when they are unprepared. For this reason, forecasters tend to artificially bump up the probability of rain and report this inflated estimate, as this provides a better payoff than the uninflated estimate.

5.2.1 Loss Functions in the Real World

So far, we have been acting under the unrealistic assumption that we know the true parameter. Of course, if we know the true parameter, bothering to guess an estimate is pointless. Hence a loss function is really only practical when the true parameter is unknown.

In Bayesian inference, we have a mindset that the unknown parameters are really random variables with prior and posterior distributions. Concerning the posterior distribution, a value drawn from it is a possible realization of what the true parameter could be. Given that realization, we can compute a loss associated with an estimate. As we have a whole distribution of what the unknown parameter could be (the posterior), we should be more interested in computing the *expected loss* given an estimate. This expected loss is a better estimate of the true loss than comparing the given loss from only a single sample from the posterior.

First, it will be useful to explain a **Bayesian point estimate**. The systems and machinery present in the modern world are not built to accept posterior distributions as input. It is also rude to hand someone over a distribution when all they asked for was an estimate. In the course of our day, when faced with uncertainty, we still act by distilling our uncertainty down to a single action. Similarly, we need to distill our posterior distribution down to a single value (or vector, in the multivariate case). If the value is chosen intelligently, we can avoid the flaw of frequentist methodologies that mask the uncertainty and provide a more informative result.The value chosen, if from a Bayesian posterior, is a Bayesian point estimate.

If $P(\theta|X)$ is the posterior distribution of θ after observing data X, then the following function is understandable as the *expected loss of choosing estimate* $\hat{\theta}$ *to estimate* θ:

$$l(\hat{\theta}) = E_\theta \Big[\; L(\theta, \hat{\theta}) \; \Big]$$

This is also known as the *risk* of estimate $\hat{\theta}$. The subscript θ under the expectation symbol is used to denote that θ is the unknown (random) variable in the expectation, something that at first can be difficult to consider.

We spent all of Chapter 4 discussing how to approximate expected values. Given N samples $\theta_i, \; i = 1, ..., N$ from the posterior distribution, and a loss function L, we can approximate the expected loss of using estimate $\hat{\theta}$ by the Law of Large Numbers:

$$\frac{1}{N} \sum_{i=1}^{N} L(\theta_i, \hat{\theta}) \approx E_\theta \Big[\; L(\theta, \hat{\theta}) \; \Big] = l(\hat{\theta})$$

Notice that measuring your loss via an expected value uses more information from the distribution than the MAP estimate—which, if you recall, will only find the maximum value of the distribution and ignore the shape of the distribution. Ignoring information can overexpose yourself to tail risks, like the unlikely hurricane, and leaves your estimate ignorant of how ignorant you really are about the parameter.

Similarly, compare this with frequentist methods, that traditionally only aim to minimize the error, and do not consider the loss associated with the result of that error. Compound this with the fact that frequentist methods are almost guaranteed to never be absolutely accurate. Bayesian point estimates fix this by planning ahead: If your estimate is going to be wrong, you might as well err on the right side of wrong.

5.2.2 Example: Optimizing for the Showcase on *The Price Is Right*

Bless you if you are ever chosen as a contestant on *The Price Is Right*, for here we will show you how to optimize your final price on the Showcase. For those who don't know the rules:

1. Two contestants compete in the Showcase.
2. Each contestant is shown a unique suite of prizes.
3. After the viewing, the contestants are asked to bid on the price for their unique suite of prizes.
4. If a bid price is over the actual price, the bid's owner is disqualified from winning.
5. If a bid price is under the true price by less than \$250, the winner is awarded both prizes.

The difficulty in the game is balancing your uncertainty in the prices, keeping your bid low enough so as to not bid over, and to bid close to the price.

Suppose we have recorded the Showcases from previous *The Price Is Right* episodes and have prior beliefs about what distribution the true price follows. For simplicity, suppose it follows a Normal:

$$\text{True Price} \sim \text{Normal}(\mu_p, \sigma_p)$$

For now, we will assume $\mu_p = 35{,}000$ and $\sigma_p = 7{,}500$.

We need a model of how we should be playing the Showcase. For each prize in the prize suite, we have an idea of what it might cost, but this guess could differ significantly from the true price. (Couple this with increased pressure from being onstage, and you can see why some bids are so wildly off.) Let's suppose your beliefs about the prices of prizes also follow Normal distributions:

$$\text{Prize}_i \sim \text{Normal}(\mu_i, \sigma_i), \quad i = 1, 2$$

This is really why Bayesian analysis is great: We can specify what we think a fair price is through the μ_i parameter, and express uncertainty of our guess in the σ_i parameter. We'll assume two prizes per suite for brevity, but this can be extended to any number. The true price of the prize suite is then given by $\text{Prize}_1 + \text{Prize}_2 + \epsilon$, where ϵ is some error term. We are interested in the updated true price given we have observed both prizes and have belief distributions about them. We can perform this using PyMC.

Let's make some values concrete. Suppose there are two prizes in the observed prize suite:

1. A trip to wonderful Toronto, Canada!
2. A lovely new snowblower!

We have some guesses about the true prices of these objects, but we are also pretty uncertain about them. We can express this uncertainty through the parameters of the Normals:

$$\text{Snowblower} \sim \text{Normal}(3000, 500)$$
$$\text{Toronto} \sim \text{Normal}(12000, 3000)$$

For example, I believe that the true price of the trip to Toronto is 12,000 dollars, and that there is a 68.2% chance the price falls 1 standard deviation away from this; that is, my confidence is that there is a 68.2% chance the trip is in [9000, 15000]. These priors are graphically represented in Figure 5.2.1.

We can create some PyMC code to perform inference on the true price of the suite, as shown in Figure 5.2.2.

```
%matplotlib inline
import scipy.stats as stats
from IPython.core.pylabtools import figsize
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['savefig.dpi'] = 300
plt.rcParams['figure.dpi'] = 300

figsize(12.5, 9)

norm_pdf = stats.norm.pdf

plt.subplot(311)
x = np.linspace(0, 60000, 200)
sp1 = plt.fill_between(x, 0, norm_pdf(x, 35000, 7500),
                color="#348ABD", lw=3, alpha=0.6,
                label="historical total prices")
p1 = plt.Rectangle((0, 0), 1, 1, fc=sp1.get_facecolor()[0])
plt.legend([p1], [sp1.get_label()])

plt.subplot(312)
x = np.linspace(0, 10000, 200)
sp2 = plt.fill_between(x, 0, norm_pdf(x, 3000, 500),
                color="#A60628", lw=3, alpha=0.6,
                 label="snowblower price guess")

p2 = plt.Rectangle((0, 0), 1, 1, fc=sp2.get_facecolor()[0])
plt.legend([p2], [sp2.get_label()])

plt.subplot(313)
x = np.linspace(0, 25000, 200)
sp3 = plt.fill_between(x , 0, norm_pdf( x, 12000, 3000),
```

```
                color="#7A68A6", lw=3, alpha=0.6,
                 label="trip price guess")
plt.autoscale(tight=True)
p3 = plt.Rectangle((0, 0), 1, 1, fc=sp3.get_facecolor()[0])
plt.title("Prior distributions for unknowns: the total price,\
           the snowblower's price, and the trip's price")
plt.legend([p3], [sp3.get_label()]);
plt.xlabel("Price");
plt.ylabel("Density")
```

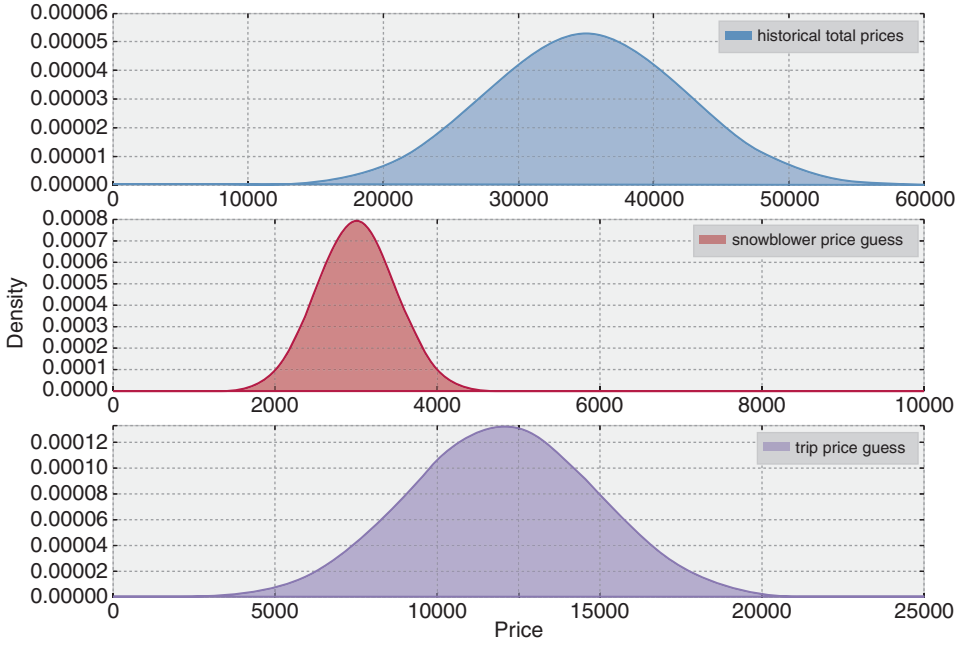

Figure 5.2.1: Prior distributions for unknowns: the total price, the snowblower's price, and the trip's price

```
import pymc as pm

data_mu = [3e3, 12e3]

data_std = [5e2, 3e3]

mu_prior = 35e3
std_prior =  75e2
```

(*Continues*)

(*Continued*)

```
true_price = pm.Normal("true_price", mu_prior, 1.0 / std_prior ** 2)


prize_1 = pm.Normal("first_prize", data_mu[0], 1.0 / data_std[0] ** 2)
prize_2 = pm.Normal("second_prize", data_mu[1], 1.0 / data_std[1] ** 2)
price_estimate = prize_1 + prize_2

@pm.potential
def error(true_price=true_price, price_estimate=price_estimate):
    return pm.normal_like(true_price,  price_estimate, 1 / (3e3) ** 2)


mcmc = pm.MCMC([true_price, prize_1, prize_2, price_estimate, error])
mcmc.sample(50000, 10000)

price_trace = mcmc.trace("true_price")[:]
```

```
[Output]:

[-----------------100%-----------------] 50000 of 50000 complete in
    10.9 sec
```

```
figsize(12.5, 4)

import scipy.stats as stats

# Plot the prior distribution.
x = np.linspace(5000, 40000)
plt.plot(x, stats.norm.pdf(x, 35000, 7500), c="k", lw=2,
         label="prior distribution\n of suite price")

# Plot the posterior distribution, represented by samples from the MCMC.
_hist = plt.hist(price_trace, bins=35, normed=True, histtype="stepfilled")
plt.title("Posterior of the true price estimate")
plt.vlines(mu_prior, 0, 1.1*np.max(_hist[0]), label="prior's mean",
             linestyles="--")
plt.vlines(price_trace.mean(), 0, 1.1*np.max(_hist[0]), \
    label="posterior's mean", linestyles="-.")
plt.legend(loc="upper left");
```

Notice that because of the snowblower prize and trip prize and subsequent guesses (including uncertainty about those guesses), we shifted our mean price estimate down about \$15,000 from the previous mean price.

A frequentist, seeing the two prizes and having the same beliefs about their prices, would bid $\mu_1 + \mu_2 = \$35{,}000$, regardless of any uncertainty. Meanwhile, the *naive Bayesian* would simply pick the mean of the posterior distribution. But we have more information about our eventual outcomes; we should incorporate this into our bid. We will use the loss function to find the *best* bid (*best* according to our loss).

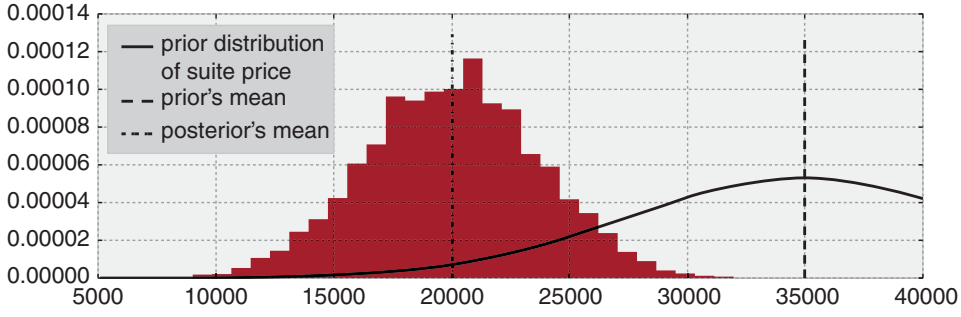

Figure 5.2.2: Posterior of the true price estimate

What might a contestant's loss function look like? I would think it would look something like:

```
def showcase_loss(guess, true_price, risk=80000):
    if true_price < guess:
        return risk
    elif abs(true_price - guess) <= 250:
        return -2 * np.abs(true_price)
    else:
        return np.abs(true_price - guess - 250)
```

where `risk` is a parameter that defines how bad it is if your guess is over the true price. I've arbitrarily picked 80,000. A lower `risk` means that you are more comfortable with the idea of going over. If we do bid under and the difference is less than $250, we receive both prizes (modeled here as receiving twice the original prize). Otherwise, when we bid under the `true_price`, we want to be as close as possible, hence the `else` loss is a increasing function of the distance between the guess and true price.

For every possible bid, we calculate the *expected loss* associated with that bid. We vary the `risk` parameter to see how it affects our loss. The results are shown in Figure 5.2.3.

```
figsize(12.5, 7)
# NumPy-friendly showdown_loss
def showdown_loss(guess, true_price, risk=80000):
        loss = np.zeros_like(true_price)
        ix = true_price < guess
        loss[~ix] = np.abs(guess - true_price[~ix])
        close_mask = [abs(true_price - guess) <= 250]
        loss[close_mask] = -2 * true_price[close_mask]
        loss[ix] = risk
        return loss
```

(*Continues*)

(*Continued*)

```
guesses = np.linspace(5000, 50000, 70)
risks = np.linspace(30000, 150000, 6)
expected_loss = lambda guess, risk: showdown_loss(guess, price_trace,
                                                  risk).mean()

for _p in risks:
    results = [expected_loss (_g, _p) for _g in guesses]
    plt.plot(guesses, results, label="%d"%_p)

plt.title("Expected loss of different guesses, \nvarious risk levels of \
          overestimating")
plt.legend(loc="upper left", title="risk parameter")
plt.xlabel("Price bid")
plt.ylabel("Expected loss")
plt.xlim(5000, 30000);
```

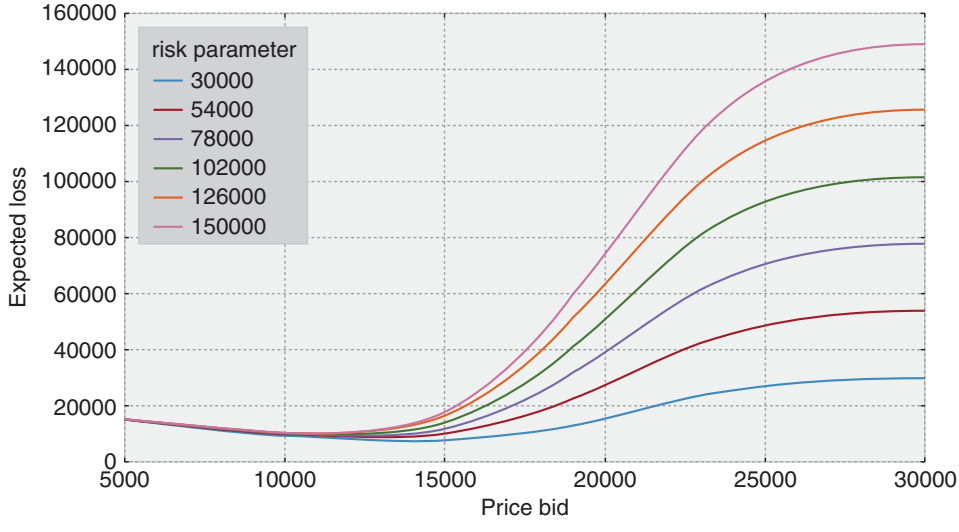

Figure 5.2.3: Expected loss of different guesses, various risk levels of overestimating

Minimizing Our Losses It would be wise to choose the estimate that minimizes our expected loss. This corresponds to the minimum point on each of the curves on the previous figure. More formally, we would like to minimize our expected loss by finding the solution to

$$\arg\min_{\hat{\theta}} \; E_{\theta}\left[\, L(\theta, \hat{\theta}) \,\right]$$

The minimum of the expected loss is called the *Bayes action*. We can solve for the Bayes action using SciPy's optimization routines. The function in `fmin` in the

`scipy.optimize` module uses an intelligent search to find a minimum (not necessarily a *global* minimum) of any univariate or multivariate function. For most purposes, `fmin` will provide you with a good answer.

We'll compute the minimum loss for the Showcase example in Figure 5.2.4.

```
import scipy.optimize as sop

ax = plt.subplot(111)


for _p in risks:
    _color = ax._get_lines.color_cycle.next()
    _min_results = sop.fmin(expected_loss, 15000, args=(_p,),disp=False)
    _results = [expected_loss(_g, _p) for _g in guesses]
    plt.plot(guesses, _results , color=_color)
    plt.scatter(_min_results, 0, s=60,
                color=_color, label="%d"%_p)
    plt.vlines(_min_results, 0, 120000, color=_color, linestyles="--")
    print "minimum at risk %d: %.2f"%(_p, _min_results)

plt.title("Expected loss and Bayes actions of different guesses, \n \
          various risk levels of overestimating")
plt.legend(loc="upper left", scatterpoints=1,
           title="Bayes action at risk:")
plt.xlabel("Price guess")
plt.ylabel("Expected loss")
plt.xlim(7000, 30000)
plt.ylim(-1000, 80000);
```

```
[Output]:

minimum at risk 30000: 14189.08
minimum at risk 54000: 13236.61
minimum at risk 78000: 12771.73
minimum at risk 102000: 11540.84
minimum at risk 126000: 11534.79
minimum at risk 150000: 11265.78
```

```
[Output]:

(-1000, 80000)
```

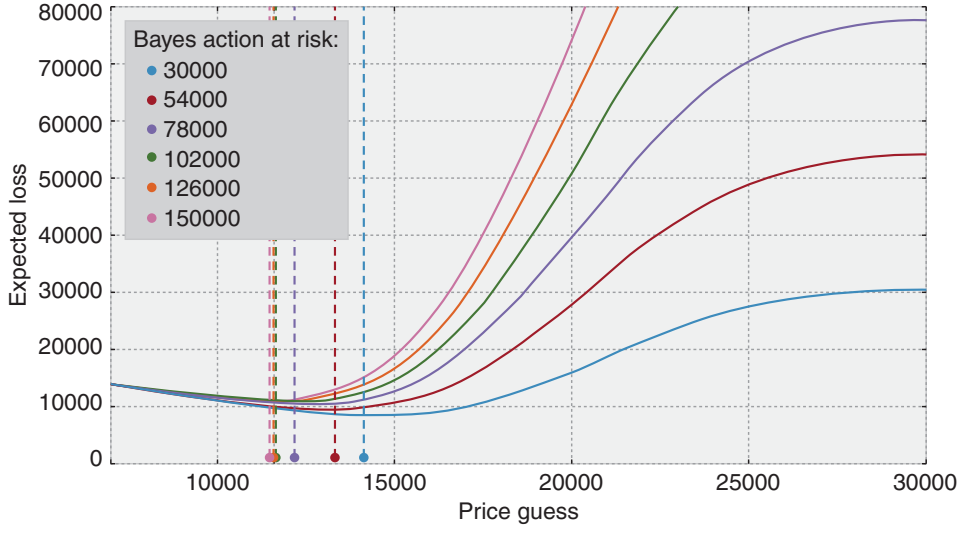

Figure 5.2.4: Expected loss and Bayes actions of different guesses, various risk levels of overestimating

As we decrease the risk threshold (care about overbidding less), we increase our bid, willing to edge closer to the true price. It is interesting how far away our optimized loss is from the posterior mean, which was about 20,000.

Suffice it to say, in higher dimensions, being able to eyeball the minimum expected loss is impossible. That is why we require use of SciPy's `fmin` function.

Shortcuts For some loss functions, the Bayes action is known in closed form. We list some of them here.

- If using the mean-squared loss, the Bayes action is the mean of the posterior distribution; that is, the value

$$E_\theta\,[\theta]$$

  minimizes $E_\theta[\ (\theta - \hat{\theta})^2\ ]$. Computationally, this requires us to calculate the average of the posterior samples (see Chapter 4 on the Law of Large Numbers).
- Whereas the median of the posterior distribution minimizes the expected absolute loss, the sample median of the posterior samples is an appropriate and very accurate approximation to the true median.
- In fact, it is possible to show that the MAP estimate is the solution to using a loss function that shrinks to the zero-one loss.

Maybe it is clear now why the first-introduced loss functions are used most often in the mathematics of Bayesian inference: No complicated optimizations are necessary. Luckily, we have machines to do the complications for us.

5.3 Machine Learning via Bayesian Methods

Whereas frequentist methods strive to achieve the best precision about all possible parameters, machine learning cares to achieve the best *prediction* among all possible parameters. Often, your prediction measure and what frequentist methods are optimizing for are very different.

For example, least-squares linear regression is the simplest active machine-learning algorithm. I say *active*, as it engages in some learning, whereas predicting the sample mean is technically simpler, but is learning very little (if anything). The loss that determines the coefficients of the regressors is a squared-error loss. On the other hand, if your prediction loss function (or score function, which is the negative loss) is not a squared-error, your least-squares line will not be optimal for the prediction loss function. This can lead to prediction results that are suboptimal.

Finding Bayes actions is equivalent to finding parameters that optimize not *parameter accuracy* but an *arbitrary performance measure*; however, we wish to define "performance" (loss functions, AUC, ROC, precision/recall, etc.).

The next two examples demonstrate these ideas. The first example is a linear model where we can choose to predict using the least-squares loss or a novel, outcome-sensitive loss. The second example is adapted from a Kaggle data science project. The loss function associated with our predictions is incredibly complicated.

5.3.1 Example: Financial Prediction

Suppose the future return of a stock price is very small, say 0.01 (or 1%). We have a model that predicts the stock's future price, and our profit and loss is directly tied to our acting on the prediction. How should we measure the loss associated with the model's predictions, and subsequent future predictions? A squared-error loss is agnostic to the signage and would penalize a prediction of -0.01 equally as badly as a prediction of 0.03:

$$(0.01 - (-0.01))^2 = (0.01 - 0.03)^2 = 0.004$$

If you had made a bet based on your model's prediction, you would have earned money with a prediction of 0.03, and lost money with a prediction of -0.01, yet our loss did not capture this. We need a better loss that takes into account the *sign* of the prediction and true value. We design a new loss that is better for financial applications, shown in Figure 5.3.1.

```
figsize(12.5, 4)
def stock_loss(true_return, yhat, alpha=100.):
    if true_return*yhat < 0:
        # opposite signs, not good
        return alpha*yhat**2 - np.sign(true_return)*yhat \
                        + abs(true_return)
    else:
        return abs(true_return - yhat)
```

(*Continues*)

(*Continued*)

```
true_value = .05
pred = np.linspace(-.04, .12, 75)

plt.plot(pred, [stock_loss(true_value, _p) for _p in pred], \
    label = "loss associated with\n prediction if true value = 0.05", lw=3)
plt.vlines(0, 0, .25, linestyles="--")

plt.xlabel("Prediction")
plt.ylabel("Loss" )
plt.xlim(-0.04, .12)
plt.ylim(0, 0.25)

true_value = -.02
plt.plot(pred, [stock_loss(true_value, _p) for _p in pred], alpha=0.6, \
    label="loss associated with\n prediction if true value = -0.02", lw=3)
plt.legend()
plt.title("Stock returns loss if true value = 0.05, -0.02" );
```

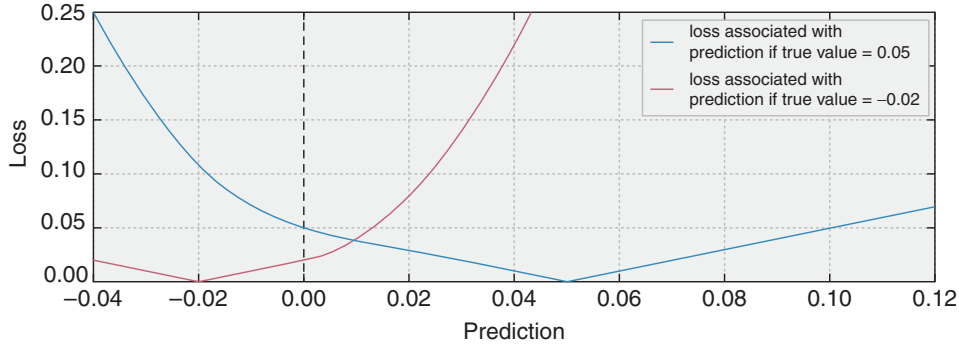

Figure 5.3.1: Stock returns loss if true value = 0.05, −0.02

Note the change in the shape of the loss as the prediction crosses 0. This loss reflects that the user really does *not* want to guess the wrong sign, and especially doesn't want to be wrong *and* with a large magnitude.

Why would the user care about the magnitude? Why is the loss not 0 for predicting the correct sign? Surely, if the return is 0.01 and we bet millions, we will still be (very) happy.

Financial institutions treat *downside risk* (as in predicting a lot on the wrong side) and *upside risk* (as in predicting a lot on the right side) similarly. Both are seen as risky behavior and are discouraged. Therefore, we have an increasing loss as we move further away from the true price, with less extreme loss in the direction of the correct sign.

We will perform a regression on a trading signal that we believe predicts future returns well. Our dataset is artificial, as most financial data is not even close to linear. In Figure 5.3.2, we plot the data along with the least-squares line.

```
# code to create artificial data
N = 100
X = 0.025 * np.random.randn(N)
Y = 0.5 * X + 0.01 * np.random.randn(N)

ls_coef_ = np.cov(X, Y)[0,1]/np.var(X)
ls_intercept = Y.mean() - ls_coef_*X.mean()

plt.scatter(X, Y, c="k")
plt.xlabel("Trading signal")
plt.ylabel("Returns")
plt.title("Empirical returns versus trading signal")
plt.plot(X, ls_coef_ * X + ls_intercept, label="least-squares line")
plt.xlim(X.min(), X.max())
plt.ylim(Y.min(), Y.max())
plt.legend(loc="upper left");
```

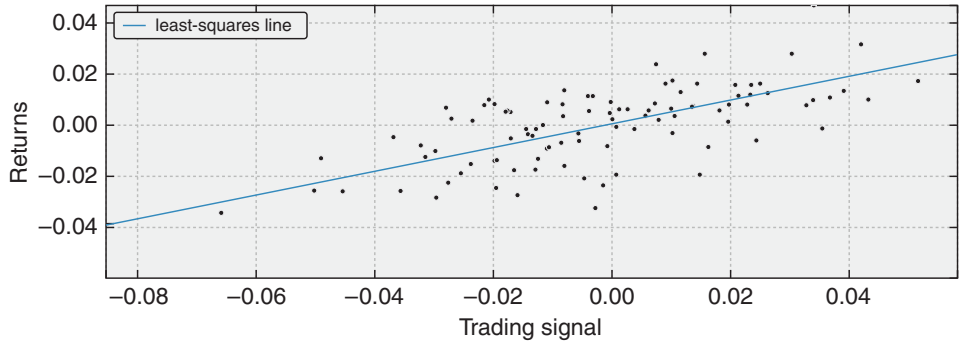

Figure 5.3.2: Empirical returns versus trading signal

We perform a simple Bayesian linear regression on this dataset. We look for a model like

$$R = \alpha + \beta x + \epsilon$$

where α, β are our unknown parameters and $\epsilon \sim \text{Normal}(0, 1/\tau)$. The most common priors on β and α are Normal priors. We will also assign a prior on τ, so that $\sigma = 1/\sqrt{\tau}$ is uniform over 0 to 100 (equivalently, then, $\tau = 1/\text{Uniform}(0, 100)^2$).

```
import pymc as pm
from pymc.Matplot import plot as mcplot

std = pm.Uniform("std", 0, 100, trace=False)

@pm.deterministic
def prec(U=std):
    return 1.0 / U **2

beta = pm.Normal("beta", 0, 0.0001)
alpha = pm.Normal("alpha", 0, 0.0001)

@pm.deterministic
def mean(X=X, alpha=alpha, beta=beta):
    return alpha + beta * X

obs = pm.Normal("obs", mean, prec, value=Y, observed=True)
mcmc = pm.MCMC([obs, beta, alpha, std, prec])

mcmc.sample(100000, 80000);
```

```
[Output]:

[-----------------100%-----------------] 100000 of 100000 complete in
    23.2 sec
```

For a specific trading signal, call it x, the distribution of possible returns has the form

$$R_i(x) = \alpha_i + \beta_i x + \epsilon$$

where $\epsilon \sim \text{Normal}(0, 1/\tau_i)$ and i indexes our posterior samples. We wish to find the solution to

$$\arg\min_r \quad E_{R(x)}\,[\, L(R(x), r)\,]$$

according to the loss given. This r is our Bayes action for trading signal x. In Figure 5.3.3, we plot the Bayes action over different trading signals. What do you notice?

```
figsize(12.5, 6)
from scipy.optimize import fmin

def stock_loss(price, pred, coef=500):
    sol = np.zeros_like(price)
    ix = price*pred < 0
    sol[ix] = coef * pred **2 - np.sign(price[ix]) * pred + abs(price[ix])
    sol[~ix] = abs(price[~ix] - pred)
    return sol

tau_samples = mcmc.trace("prec")[:]
alpha_samples = mcmc.trace("alpha")[:]
beta_samples = mcmc.trace("beta")[:]
```

```
N = tau_samples.shape[0]

noise = 1. / np.sqrt(tau_samples) * np.random.randn(N)

possible_outcomes = lambda signal: alpha_samples + beta_samples * signal \
                                    +u noise


opt_predictions = np.zeros(50)
trading_signals = np.linspace(X.min(), X.max(), 50)
for i, _signal in enumerate(trading_signals):
       _possible_outcomes = possible_outcomes(_signal)
       tomin = lambda pred: stock_loss(_possible_outcomes, pred).mean()
       opt_predictions[i] = fmin(tomin, 0, disp=False)


plt.xlabel("Trading signal")
plt.ylabel("Prediction")
plt.title("Least-squares prediction versus Bayes action prediction" )
plt.plot(X, ls_coef_ * X + ls_intercept,
          label="least-squares prediction")
plt.xlim(X.min(), X.max())
plt.plot(trading_signals, opt_predictions,
          label="Bayes action prediction")
plt.legend(loc="upper left");
```

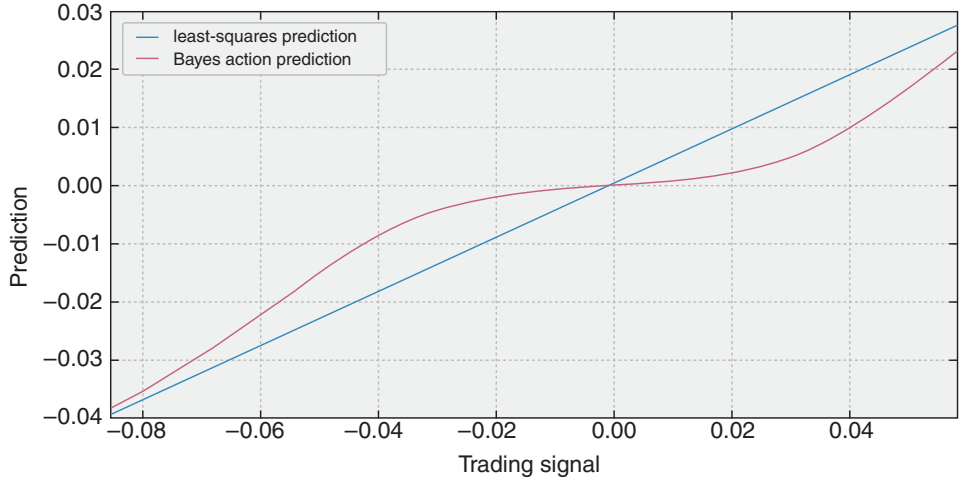

Figure 5.3.3: Least-squares prediction versus Bayes action prediction

What is interesting about Figure 5.3.3 is that when the signal is near 0, and many of the possible returns are possibly both positive and negative, our best (with respect to our loss)

move is to predict close to 0; that is, take on no position. Only when we are very confident do we enter into a position. I call this style of model a **sparse prediction**, where we feel uncomfortable with our uncertainty so choose not to act. (Compare this with the least-squares prediction, which will rarely, if ever, predict 0.)

A good sanity check that our model is still reasonable is that as the signal becomes more and more extreme, and we feel more and more confident about the positiveness/negativeness of returns, our position converges with that of the least-squares line.

The sparse-prediction model is not trying to fit the data the best according to a squared-error loss definition of fit. That honor would go to the least-squares model. The sparse-prediction model is trying to find the best prediction *with respect to our* `stock_loss`*-defined loss*. We can turn this reasoning around: The least-squares model is not trying to predict the best (according to a `stock-loss` definition of "predict"). That honor would go the sparse-prediction model. The least-squares model is trying to find the best fit of the data *with respect to the squared-error loss*.

5.3.2 Example: Kaggle Contest on Observing Dark Worlds

A personal motivation for learning Bayesian methods was trying to piece together the winning solution to Kaggle's Observing Dark Worlds contest. From the contest's website:[2]

> There is more to the Universe than meets the eye. Out in the cosmos exists a form of matter that outnumbers the stuff we can see by almost 7 to 1, and we don't know what it is. What we do know is that it does not emit or absorb light, so we call it ***Dark Matter***.
>
> Such a vast amount of aggregated matter does not go unnoticed. In fact we observe that this stuff aggregates and forms massive structures called ***Dark Matter Halos***.
>
> Although dark, it warps and bends spacetime such that any light from a background galaxy which passes close to the *Dark Matter* will have its path altered and changed. This bending causes the galaxy to appear as an ellipse in the sky.

The contest required predictions about where dark matter was likely to be. The winner, Tim Salimans, used Bayesian inference to find the best locations for the halos (interestingly, the second-place winner also used Bayesian inference). With Tim's permission, we provide his solution[3] here.

1. Construct a prior distribution for the halo positions $p(x)$, i.e. formulate our expectations about the halo positions before looking at the data.
2. Construct a probabilistic model for the data (observed ellipticities of the galaxies) given the positions of the dark matter halos: $p(e|x)$.
3. Use Bayes' rule to get the posterior distribution of the halo positions, i.e. use to [*sic*] the data to guess where the dark matter halos might be.
4. Minimize the expected loss with respect to the posterior distribution over the predictions for the halo positions: $\hat{x} = \arg\min_{\text{prediction}} E_{p(x|e)}[L(\text{prediction}, x)]$, i.e. tune our predictions to be as good as possible for the given error metric.

The loss function in this problem is very complicated. For the very determined, the loss function is contained in the file DarkWorldsMetric.py. Though I suggest not reading it all, suffice it to say the loss function is about 160 lines of code—not something that can be written down in a single mathematical line. The loss function attempts to measure the accuracy of prediction, in a Euclidean distance sense, such that no shift bias is present. More details can be found on the contest's homepage.

We will attempt to implement Tim's winning solution using PyMC and our knowledge of loss functions.

5.3.3 The Data

The dataset is actually 300 separate files, each representing a sky. In each file, or sky, are between 300 and 720 galaxies. Each galaxy has an x and y position associated with it, ranging from 0 to 4,200, and measures of ellipticity: e_1 and e_2. Information about what these measures mean can be found at `https://www.kaggle.com/c/DarkWorlds/details/an-introduction-to-ellipticity`, but we only care about that for visualization purposes. Thus, a typical sky might look like Figure 5.3.4.

```
from draw_sky2 import draw_sky

n_sky = 3 # choose a file/sky to examine
data = np.genfromtxt("data/Train_Skies/Train_Skies/\
Training_Sky%d.csv"%(n_sky),
                     dtype=None,
                     skip_header=1,
                     delimiter=",",
                     usecols=[1,2,3,4])
print "Data on galaxies in sky %d."%n_sky
print "position_x, position_y, e_1, e_2 "
print data[:3]

fig = draw_sky(data)
plt.title("Galaxy positions and ellipticities of sky %d."%n_sky)
plt.xlabel("$x$ position")
plt.ylabel("$y$ position");
```

```
[Output]:

Data on galaxies in sky 3.
position_x, position_y, e_1, e_2
[[ 1.62690000e+02 1.60006000e+03 1.14664000e-01 -1.90326000e-01]
[  2.27228000e+03 5.40040000e+02 6.23555000e-01  2.14979000e-01]
[  3.55364000e+03 2.69771000e+03 2.83527000e-01 -3.01870000e-01]]
```

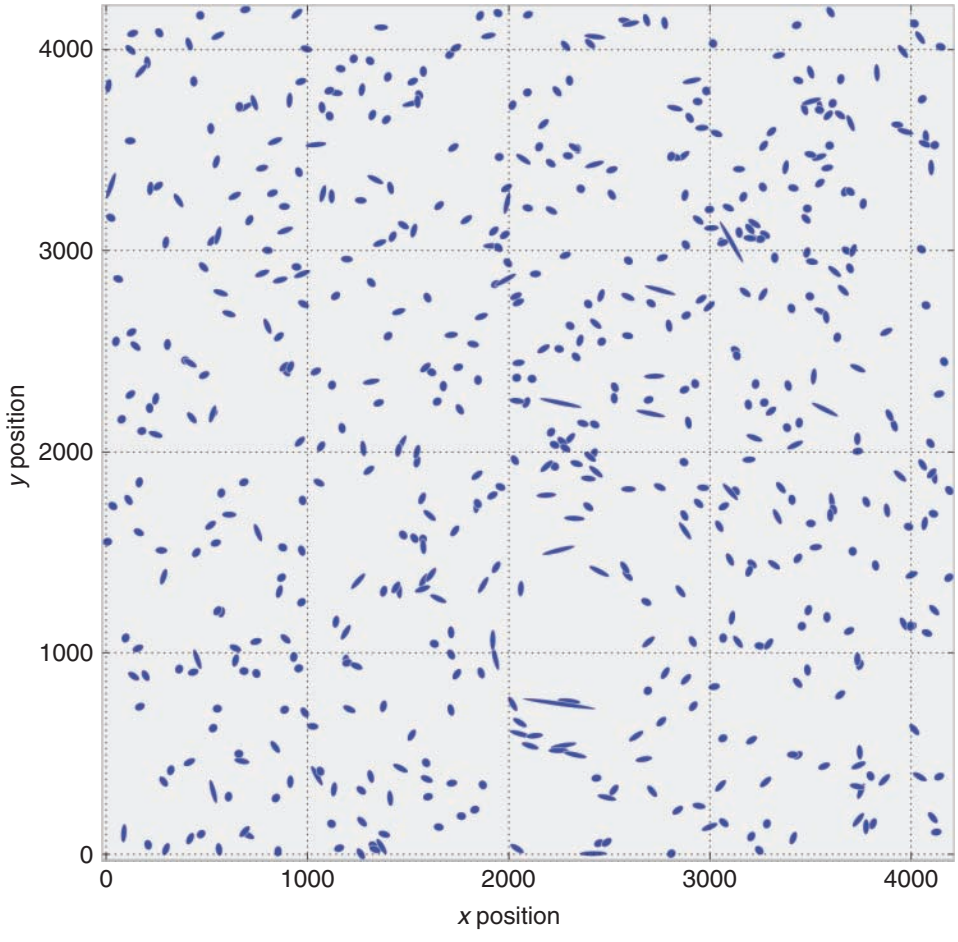

Figure 5.3.4: Galaxy positions and ellipticities of sky 3

5.3.4 Priors

Each sky has one, two, or three dark matter halos in it. Tim's solution details that his prior distribution of halo positions was uniform; that is,

$$x_i \sim \text{Uniform}(0, 4200)$$
$$y_i \sim \text{Uniform}(0, 4200), \quad i = 1, 2, 3$$

Tim and other competitors noted that most skies had one large halo, and other halos, if present, were much smaller. Larger halos, having more mass, will influence the surrounding galaxies more. He decided that the large halos would have a mass distributed

as a *log*-uniform random variable between 40 and 180; that is,

$$m_{\text{large}} = \log \text{Uniform}(40, 180)$$

and in PyMC,

```
exp_mass_large = pm.Uniform("exp_mass_large", 40, 180)
@pm.deterministic
def mass_large(u = exp_mass_large):
    return np.log(u)
```

(This is what we mean when we say "*log*-uniform.") For smaller galaxies, Tim set the mass to be the logarithm of 20. Why did Tim not create a prior for the smaller mass, or treat it as a unknown? I believe this decision was made to speed up convergence of the algorithm. This is not too restrictive, as by construction, the smaller halos have less influence on the galaxies.

Tim logically assumed that the ellipticity of each galaxy is dependent on the position of the halos, the distance between the galaxy and halo, and the mass of the halos. Thus, the vector of ellipticity of each galaxy, $\mathbf{e}_i$, are *children* variables of the vector of halo positions $(\mathbf{x}, \mathbf{y})$, distance (which we will formalize), and halo masses.

Tim conceived a relationship to connect positions and ellipticity by reading literature and forum posts. He supposed the following was a reasonable relationship:

$$e_i|(\mathbf{x}, \mathbf{y}) \sim \text{Normal}\Big(\sum_{j=\text{halo positions}} d_{i,j} m_j f(r_{i,j}), \sigma^2\Big)$$

where $d_{i,j}$ is the *tangential direction* (the direction in which halo j bends the light of galaxy i), m_j is the mass of halo j, and $f(r_{i,j})$ is a *decreasing function* of the Euclidean distance between halo j and galaxy i.

Tim's function f was defined:

$$f(r_{i,j}) = \frac{1}{\min(r_{i,j}, 240)}$$

for large halos, and for small halos

$$f(r_{i,j}) = \frac{1}{\min(r_{i,j}, 70)}$$

This fully bridges our observations and unknown. This model is incredibly simple, and Tim mentions that this simplicity was purposely designed; it prevents the model from overfitting.

5.3.5 Training and PyMC Implementation

For each sky, we run our Bayesian model to find the posteriors for the halo positions—we ignore the (known) halo position. This is slightly different from perhaps more traditional approaches to Kaggle competitions, where this model uses no data from other skies or from the known halo location. That does not mean other data are not necessary; in fact, the model was created by comparing different skies.

```
def euclidean_distance(x, y):
    return np.sqrt(((x - y) **2).sum(axis=1))

def f_distance(gxy_pos, halo_pos, c):
    # foo_position should be a 2D numpy array.
    return np.maximum(euclidean_distance(gxy_pos, halo_pos), c)[:,None]

def tangential_distance(glxy_position, halo_position):
    # foo_position should be a 2D numpy array.
    delta = glxy_position - halo_position
    t = (2*np.arctan(delta[:,1]/delta[:,0]))[:,None]
    return np.concatenate([-np.cos(t), -np.sin(t)], axis=1)

import pymc as pm

# Set the size of the halo's mass.
mass_large = pm.Uniform("mass_large", 40, 180, trace=False)

# Set the initial prior position of the halos; it's a 2D Uniform
# distribution.
halo_position = pm.Uniform("halo_position", 0, 4200, size=(1,2))


@pm.deterministic
def mean(mass=mass_large, h_pos=halo_position, glx_pos=data[:,:2]):
    return mass/f_distance(glx_pos, h_pos, 240)*\
            tangential_distance(glx_pos, h_pos)

ellpty = pm.Normal("ellipticity", mean, 1./0.05, observed=True,
                   value=data[:,2:] )
mcmc = pm.MCMC([ellpty, mean, halo_position, mass_large])
map_ = pm.MAP([ellpty, mean, halo_position, mass_large])
map_.fit()
mcmc.sample(200000, 140000, 3)
```

```
[Output]:

[****************100%******************] 200000 of 200000 complete
```

In Figure 5.3.5, we plot a heatmap of the posterior distribution (this is just a scatter plot of the posterior, but we can visualize it as a heatmap). As you can see in the figure, the red spot denotes our posterior distribution over where the halo is.

```
t = mcmc.trace("halo_position")[:].reshape( 20000,2)

fig = draw_sky(data)
plt.title("Galaxy positions and ellipticities of sky %d."%n_sky)
plt.xlabel("$x$ position")
plt.ylabel("$y$ position")
scatter(t[:,0], t[:,1], alpha=0.015, c="r")
plt.xlim(0, 4200)
plt.ylim(0, 4200);
```

The most probable position reveals itself like a lethal wound.

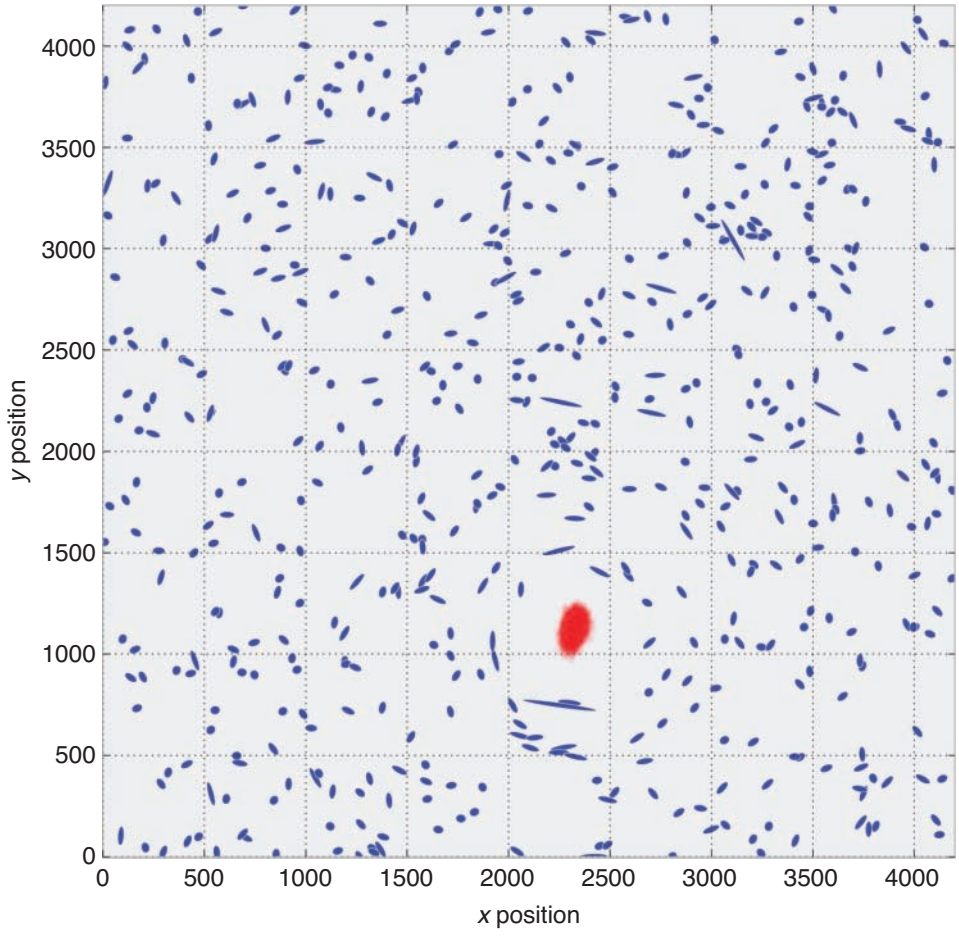

Figure 5.3.5: Galaxy positions and ellipticities of sky 3

Associated with each sky is another data point, located in `Training_halos.csv`, that holds the locations of up to three dark matter halos contained in the sky. For example, the night sky we trained on has halo locations

```
halo_data = np.genfromtxt("data/Training_halos.csv",
                          delimiter=",",
                          usecols=[1,2,3,4,5,6,7,8,9],
                          skip_header=1)
print halo_data[n_sky]
```

```
[Output]:

[ 3.00000000e+00 2.78145000e+03 1.40691000e+03 3.08163000e+03
  1.15611000e+03 2.28474000e+03 3.19597000e+03 1.80916000e+03
  8.45180000e+02]
```

The third and fourth column represent the true x and y position of the halo. It appears that the Bayesian method has located the halo within a tight vicinity, as denoted by the black dot in Figure 5.3.6.

```
fig = draw_sky(data)
plt.title("Galaxy positions and ellipticities of sky %d."%n_sky)
plt.xlabel("$x$ position")
plt.ylabel("$y$ position" )
plt.scatter(t[:,0], t[:,1], alpha=0.015, c="r")
plt.scatter(halo_data[n_sky-1][3], halo_data[n_sky-1][4],
            label="true halo position",
            c="k", s=70)
plt.legend(scatterpoints=1, loc="lower left")
plt.xlim(0, 4200)
plt.ylim(0, 4200);

print "True halo location:", halo_data[n_sky][3], halo_data[n_sky][4]
```

```
[Output]:

True halo location: 1408.61 1685.86
```

Perfect. Our next step is to use the loss function to optimize our location. A naive strategy would be to simply choose the mean:

```
mean_posterior = t.mean(axis=0).reshape(1,2)
print mean_posterior
```

```
[Output]:

[[ 2324.07677813 1122.47097816]]
```

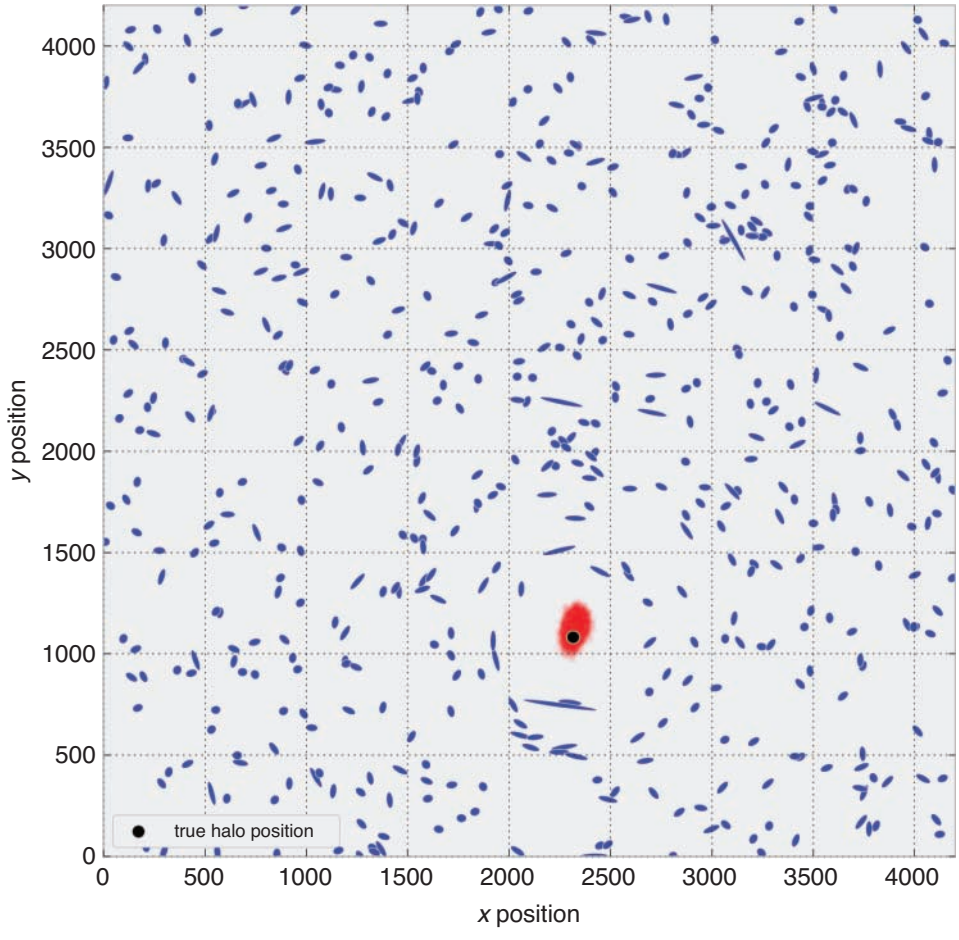

Figure 5.3.6: Galaxy positions and ellipticities of sky 3

```
from DarkWorldsMetric import main_score

_halo_data = halo_data[n_sky-1]

nhalo_all =  _halo_data[0].reshape(1,1)
x_true_all = _halo_data[3].reshape(1,1)
y_true_all = _halo_data[4].reshape(1,1)
x_ref_all = _halo_data[1].reshape(1,1)
y_ref_all = _halo_data[2].reshape(1,1)
sky_prediction = mean_posterior

print "Using the mean:"
```

(*Continues*)

(*Continued*)

```
main_score(nhalo_all, x_true_all, y_true_all, \
            x_ref_all, y_ref_all, sky_prediction)

# What's a bad score?
print
random_guess = np.random.randint(0, 4200, size=(1,2))
print "Using a random location:", random_guess
main_score(nhalo_all, x_true_all, y_true_all, \
            x_ref_all, y_ref_all, random_guess)
print
```

```
[Output]:

Using the mean:
Your average distance in pixels away from the true halo is
    31.1499201664
Your average angular vector is 1.0
Your score for the training data is 1.03114992017

Using a random location: [[2755 53]]
Your average distance in pixels away from the true halo is
    1773.42717812
Your average angular vector is 1.0
Your score for the training data is 2.77342717812
```

This is a good guess; it is not very far from the true location, but it ignores the loss function that was provided to us. We also need to extend our code to allow for up to two additional, *smaller* halos. Let's create a function for automatizing our PyMC.

```
from pymc.Matplot import plot as mcplot

def halo_posteriors(n_halos_in_sky, galaxy_data,
                    samples = 5e5, burn_in = 34e4, thin = 4):

    # Set the size of the halo's mass.

    mass_large = pm.Uniform("mass_large", 40, 180)

    mass_small_1 = 20
    mass_small_2 = 20

    masses = np.array([mass_large,mass_small_1, mass_small_2],
                      dtype=object)

    # Set the initial prior positions of the halos; it's a 2D Uniform
    # distribution.
    halo_positions = pm.Uniform("halo_positions", 0, 4200,
                    size=(n_halos_in_sky,2))
```

```
    fdist_constants = np.array([240, 70, 70])

    @pm.deterministic
    def mean(mass=masses, h_pos=halo_positions, glx_pos=data[:,:2],
             n_halos_in_sky = n_halos_in_sky):

        _sum = 0
        for i in range(n_halos_in_sky):
            _sum += mass[i] / f_distance( glx_pos,h_pos[i, :],
                 fdist_constants[i])*\
                     tangential_distance( glx_pos, h_pos[i, :])


        return _sum


    ellpty = pm.Normal("ellipticity", mean, 1. / 0.05, observed=True,
                       value = data[:,2:])

    map_ = pm.MAP([ellpty, mean, halo_positions, mass_large])
    map_.fit(method="fmin_powell")

    mcmc = pm.MCMC([ellpty, mean, halo_positions, mass_large])
    mcmc.sample(samples, burn_in, thin)
    return mcmc.trace("halo_positions")[:]

n_sky =215
data = np.genfromtxt("data/Train_Skies/Train_Skies/\
Training_Sky%d.csv"%(n_sky),
                     dtype=None,
                     skip_header=1,
                     delimiter=",",
                     usecols=[1,2,3,4])

# There are 3 halos in this file.
samples = 10.5e5
traces = halo_posteriors(3, data, samples=samples,
                                    burn_in=9.5e5,
                                    thin=10)
```

```
[Output]:

[****************100%******************] 1050000 of 1050000 complete
```

```
fig = draw_sky(data)
plt.title("Galaxy positions, ellipticities, and halos of sky %d."%n_sky)
plt.xlabel("$x$ position")
plt.ylabel("$y$ position")
```

(*Continues*)

(*Continued*)

```
colors = ["#467821", "#A60628", "#7A68A6"]

for i in range(traces.shape[1]):
    plt.scatter(traces[:, i, 0], traces[:, i, 1], c=colors[i],
                alpha=0.02)


for i in range(traces.shape[1]):
    plt.scatter(halo_data[n_sky-1][3 + 2 * i],
         halo_data[n_sky-1][4 + 2 * i],
            label="true halo position", c="k", s=90)

plt.xlim(0, 4200)
plt.ylim(0, 4200);
```

```
[Output]:

(0, 4200)
```

As you can see in Figure 5.3.7, this looks pretty good, though it took a long time for the system to (sort of) converge. Our optimization step would look something like this.

```
_halo_data = halo_data[n_sky-1]
print traces.shape

mean_posterior = traces.mean(axis=0).reshape(1,4)
print mean_posterior


nhalo_all = _halo_data[0].reshape(1,1)
x_true_all = _halo_data[3].reshape(1,1)
y_true_all = _halo_data[4].reshape(1,1)
x_ref_all = _halo_data[1].reshape(1,1)
y_ref_all = _halo_data[2].reshape(1,1)
sky_prediction = mean_posterior


print "Using the mean:"
main_score([1], x_true_all, y_true_all, \
            x_ref_all, y_ref_all, sky_prediction)

# What's a bad score?
print
random_guess = np.random.randint(0, 4200, size=(1,2))
print "Using a random location:", random_guess
main_score([1], x_true_all, y_true_all, \
            x_ref_all, y_ref_all, random_guess)
print
```

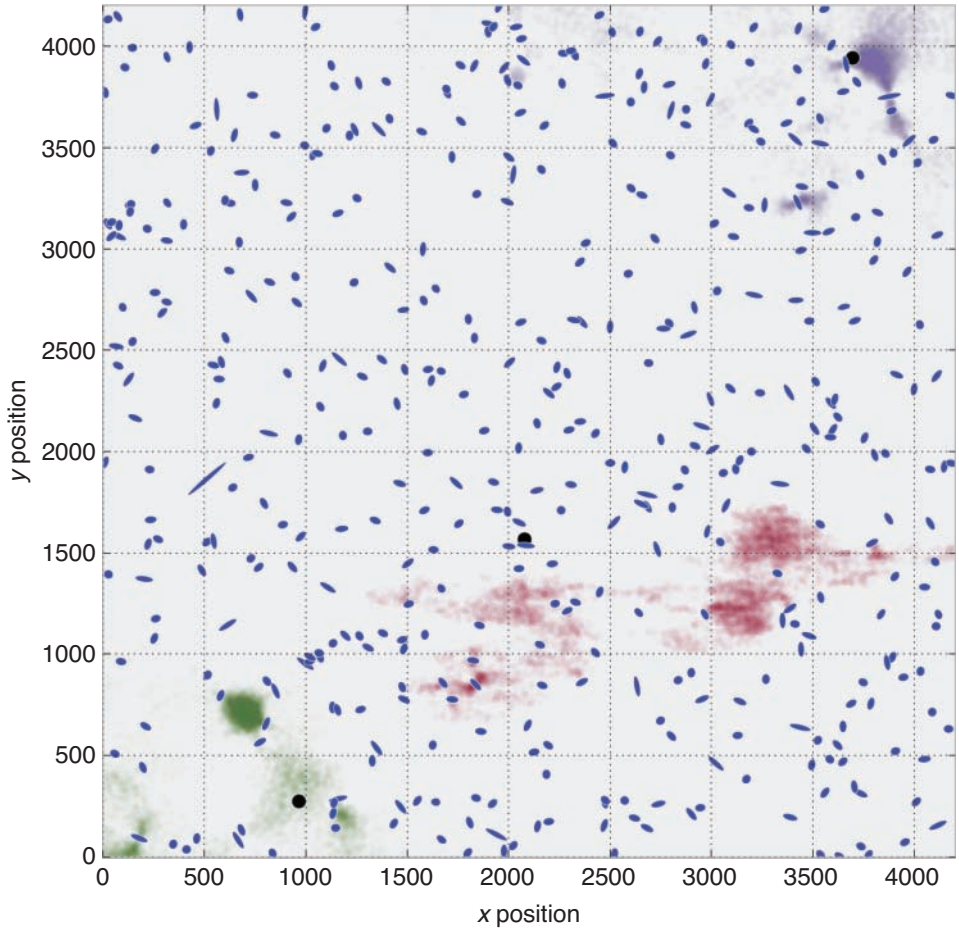

Figure 5.3.7: Galaxy positions, ellipticities, and halos of sky 215

```
[Output]:

(10000L, 2L, 2L)
[[ 48.55499317 1675.79569424 1876.46951857 3265.85341193]]
Using the mean:
Your average distance in pixels away from the true halo is
    37.3993004245
Your average angular vector is 1.0
Your score for the training data is 1.03739930042
```

(*Continues*)

(*Continued*)

```
Using a random location: [[2930 4138]]
Your average distance in pixels away from the true halo is
    3756.54446887
Your average angular vector is 1.0
Your score for the training data is 4.75654446887
```

5.4 Conclusion

Loss functions are one of the most interesting parts of statistics. They directly connect inference and the domain the problem is in. One thing *not* mentioned is that the loss function is another degree of freedom in your overall model. This is a good thing, as we saw in this chapter; loss functions can be used very effectively, but can be a bad thing, too. An extreme case is that a practitioner can change his or her loss function if the results do not fit the desired result. For this reason, it's best to set the loss function as soon as possible in the analysis, and have its derivation open and logical.

5.5 References

1. Read, Carveth. *Logic: Deductive and Inductive*. London: Simkin, Marshall, 1920, p. vi.
2. "Observing Dark Worlds," Kaggle, accessed November 30, 2014, `https://www.kaggle.com/c/DarkWorlds`.
3. Salimans, Tim. "Observing Dark Worlds," Tim Salimans on Data Analysis, accessed May 19, 2015, `http://timsalimans.com/observing-dark-worlds/`.

This page intentionally left blank

Index

Symbols and Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z