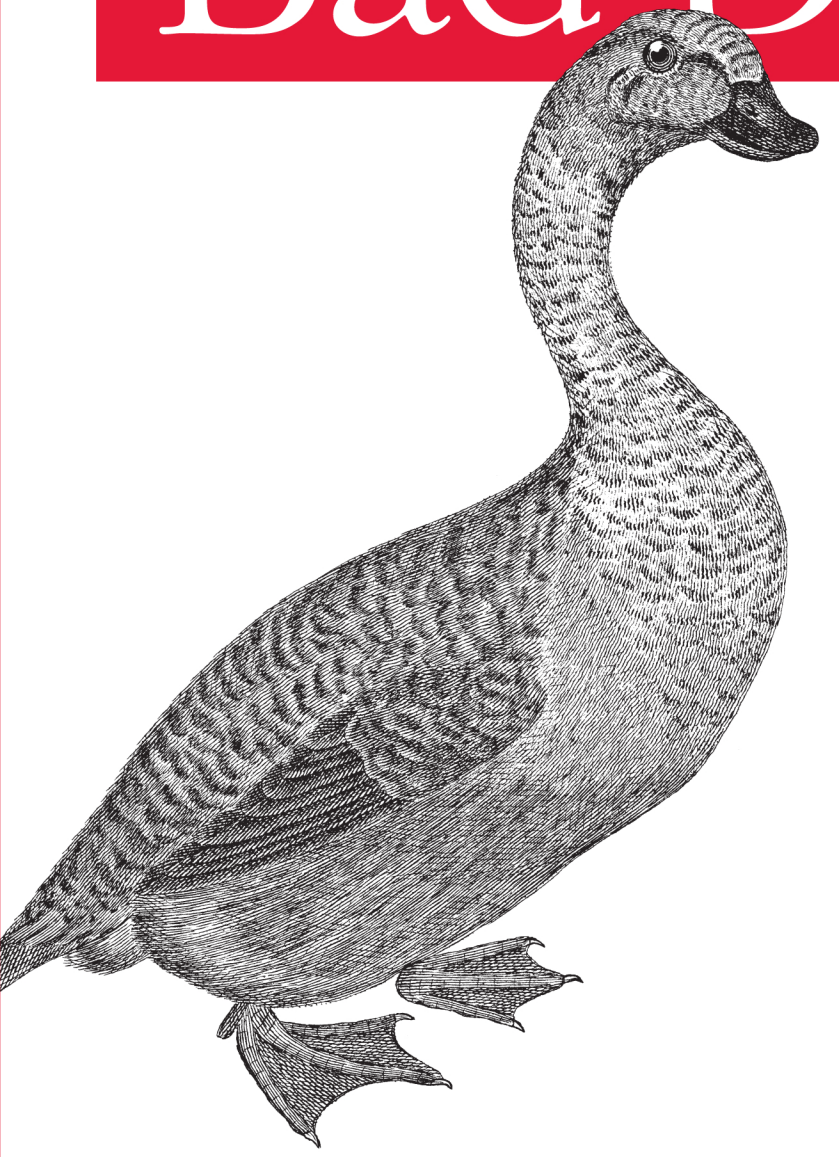
Mapping the World of Data Problems

Bad Data Handbook



O'REILLY®

Q. Ethan McCallum

Bad Data Handbook

Q. Ethan McCallum

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Bad Data Handbook

by Q. Ethan McCallum

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Meghan Blanchette
Production Editor: Melanie Yarbrough
Copyeditor: Gillian McGarvey

Proofreader: Melanie Yarbrough
Indexer: Angela Howard
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

November 2012: First Edition

Revision History for the First Edition:

2012-11-05 First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449321888* for release details.

ISBN: 978-1-449-32188-8

[LSI]

Table of Contents

About the Authors

(Guilty parties are listed in order of appearance.)

Kevin Fink is an experienced biztech executive with a passion for turning data into business value. He has helped take two companies public (as CTO of N2H2 in 1999 and SVP Engineering at Demand Media in 2011), in addition to helping grow others (including as CTO of WhitePages.com for four years). On the side, he and his wife run Traumhof, a dressage training and boarding stable on their property east of Seattle. In his copious free time, he enjoys hiking, riding his tandem bicycle with his son, and geocaching.

Paul Murrell is a senior lecturer in the Department of Statistics at the University of Auckland, New Zealand. His research area is Statistical Computing and Graphics and he is a member of the core development team for the R project. He is the author of two books, *R Graphics* and *Introduction to Data Technologies,* and is a Fellow of the American Statistical Association.

Josh Levy is a data scientist in Austin, Texas. He works on content recommendation and text mining systems. He earned his doctorate at the University of North Carolina where he researched statistical shape models for medical image segmentation. His favorite foosball shot is banked from the backfield.

Adam Laiacano has a BS in Electrical Engineering from Northeastern University and spent several years designing signal detection systems for atomic clocks before joining a prominent NYC-based startup.

Jacob Perkins is the CTO of Weotta, a NLTK contributer, and the author of *Python Text Processing with NLTK Cookbook.* He also created the NLTK demo and API site text-processing.com, and periodically blogs at streamhacker.com. In a previous life, he invented the refrigerator.

Spencer Burns is a data scientist/engineer living in San Francisco. He has spent the past 15 years extracting information from messy data in fields ranging from intelligence to quantitative finance to social media.

Richard Cotton is a data scientist with a background in chemical health and safety, and has worked extensively on tools to give non-technical users access to statistical models. He is the author of the R packages "assertive" for checking the state of your variables and "sig" to make sure your functions have a sensible API. He runs The Damned Liars statistics consultancy.

Philipp K. Janert was born and raised in Germany. He obtained a Ph.D. in Theoretical Physics from the University of Washington in 1997 and has been working in the tech industry since, including four years at Amazon.com, where he initiated and led several projects to improve Amazon's order fulfillment process. He is the author of two books on data analysis, including the best-selling *Data Analysis with Open Source Tools* (O'Reilly, 2010), and his writings have appeared on Perl.com, IBM developerWorks, IEEE Software, and in the Linux Magazine. He also has contributed to CPAN and other open-source projects. He lives in the Pacific Northwest.

Jonathan Schwabish is an economist at the Congressional Budget Office. He has conducted research on inequality, immigration, retirement security, data measurement, food stamps, and other aspects of public policy in the United States. His work has been published in the *Journal of Human Resources,* the *National Tax Journal,* and elsewhere. He is also a data visualization creator and has made designs on a variety of topics that range from food stamps to health care to education. His visualization work has been featured on the visualizaing.org and visual.ly websites. He has also spoken at numerous government agencies and policy institutions about data visualization strategies and best practices. He earned his Ph.D. in economics from Syracuse University and his undergraduate degree in economics from the University of Wisconsin at Madison.

Brett Goldstein is the Commissioner of the Department of Innovation and Technology for the City of Chicago. He has been in that role since June of 2012. Brett was previously the city's Chief Data Officer. In this role, he lead the city's approach to using data to help improve the way the government works for its residents. Before coming to City Hall as Chief Data Officer, he founded and commanded the Chicago Police Department's Predictive Analytics Group, which aims to predict when and where crime will happen. Prior to entering the public sector, he was an early employee with OpenTable and helped build the company for seven years. He earned his BA from Connecticut College, his MS in criminal justice at Suffolk University, and his MS in computer science at University of Chicago. Brett is pursuing his PhD in Criminology, Law, and Justice at the University of Illinois-Chicago. He resides in Chicago with his wife and three children.

Bobby Norton is the co-founder of Tested Minds, a startup focused on products for social learning and rapid feedback. He has built software for over 10 years at firms such as Lockheed Martin, NASA, GE Global Research, ThoughtWorks, DRW Trading Group, and Aurelius. His data science tools of choice include Java, Clojure, Ruby, Bash, and R. Bobby holds a MS in Computer Science from FSU.

Steve Francia is the Chief Evangelist at 10gen where he is responsible for the MongoDB user experience. Prior to 10gen he held executive engineering roles at OpenSky, Portero, Takkle and Supernerd. He is a popular speaker on a broad set of topics including cloud computing, big data, e-commerce, development and databases. He is a published author, syndicated blogger (spf13.com) and frequently contributes to industry publications. Steve's work has been featured by the New York Times, Guardian UK, Mashable, ReadWriteWeb, and more. Steve is a long time contributor to open source. He enjoys coding in Vim and maintains a popular Vim distribution. Steve lives with his wife and four children in Connecticut.

Tim McNamara is a New Zealander with a laptop and a desire to do good. He is an active participant in both local and global open data communities, jumping between organising local meetups to assisting with the global CrisisCommons movement. His skills as a programmer began while assisting with the development Sahana Disaster Management System, were refined helping Sugar Labs, the software which runs the One Laptop Per Child XO. Tim has recently moved into the escience field, where he works to support the research community's uptake of technology.

Marck Vaisman is a data scientist and claims he's been one before the term was en vogue. He is also a consultant, entrepreneur, master munger, and hacker. Marck is the principal data scientist at DataXtract, LLC where he helps clients ranging from startups to Fortune 500 firms with all kinds of data science projects. His professional experience spans the management consulting, telecommunications, Internet, and technology industries. He is the co-founder of Data Community DC, an organization focused on building the Washington DC area data community and promoting data and statistical sciences by running Meetup events (including Data Science DC and R Users DC) and other initiatives. He has an MBA from Vanderbilt University and a BS in Mechanical Engineering from Boston University. When he's not doing something data related, you can find him geeking out with his family and friends, swimming laps, scouting new and interesting restaurants, or enjoying good beer.

Pete Warden is an ex-Apple software engineer, wrote the *Big Data Glossary* and the *Data Source Handbook* for O'Reilly, created the open-source projects Data Science Toolkit and OpenHeatMap, and broke the story about Apple's iPhone location tracking file. He's the CTO and founder of Jetpac, a data-driven social photo iPad app, with over a billion pictures analyzed from 3 million people so far.

Jud Valeski is co-founder and CEO of Gnip, the leading provider of social media data for enterprise applications. From client-side consumer facing products to large scale

backend infrastructure projects, he has enjoyed working with technology for over twenty years. He's been a part of engineering, product, and M&A teams at IBM, Netscape, onebox.com, AOL, and me.dium. He has played a central role in the release of a wide range of products used by tens of millions of people worldwide.

Reid Draper is a functional programmer interested in distributed systems, programming languages, and coffee. He's currently working for Basho on their distributed database: Riak.

Ken Gleason's technology career experience spans more than twenty years, including real-time trading system software architecture and development and retail financial services application design. He has spent the last ten years in the data-driven field of electronic trading, where he has managed product development and high-frequency trading strategies. Ken holds an MBA from the University of Chicago Booth School of Business and a BS from Northwestern University.

Q. Ethan McCallum works as a professional-services consultant. His technical interests range from data analysis, to software, to infrastructure. His professional focus is helping businesses improve their standing—in terms of reduced risk, increased profit, and smarter decisions—through practical applications of technology. His written work has appeared online and in print, including *Parallel R: Data Analysis in the Distributed World* (O'Reilly, 2011).

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

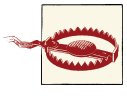
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Bad Data Handbook* by Q. Ethan McCallum (O'Reilly). Copyright 2013 Q. McCallum, 978-1-449-32188-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/bad_data_handbook*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

It's odd, really. Publishers usually stash a book's acknowledgements into a small corner, outside the periphery of the "real" text. That makes it easy for readers to trivialize all that it took to bring the book into being. Unless you've written a book yourself, or have had a hand in publishing one, it may surprise you to know just what is involved in turning an idea into a neat package of pages (or screens of text).

To be blunt, a book is a Big Deal. To publish one means to assemble and coordinate a number of people and actions over a stretch of time measured in months or even years. My hope here is to shed some light on, and express my gratitude to, the people who made this book possible.

Mike Loukides: This all started as a casual conversation with Mike. Our meandering chat developed into a brainstorming session, which led to an idea, which eventually turned into this book. (Let's also give a nod to serendipity. Had I spoken with Mike on a different day, at a different time, I wonder whether we would have decided on a completely different book?)

Meghan Blanchette: As the book's editor, Meghan kept everything organized and on track. She was a tireless source of ideas and feedback. That's doubly impressive when you consider that *Bad Data Handbook* was just one of several titles under her watch. I look forward to working with her on the next project, whatever that may be and whenever that may happen.

Contributors, and those who helped me find them: I shared writing duties with 18 other people, which accounts for the rich variety of topics and stories here. I thank all

of the contributors for their time, effort, flexibility, and especially their grace in handling my feedback. I also thank everyone who helped put me in contact with prospective contributors, without whom this book would have been quite a bit shorter, and more limited in coverage.

The entire O'Reilly team: It's a pleasure to write with the O'Reilly team behind me. The whole experience is seamless: things *just work,* and that means I get to focus on the writing. Thank you all!

CHAPTER 1

Setting the Pace: What Is Bad Data?

We all *say* we like data, but we don't.

We like *getting insight* out of data. That's not quite the same as liking the data itself.

In fact, I dare say that I don't quite care for data. It sounds like I'm not alone.

It's tough to nail down a precise definition of "Bad Data." Some people consider it a purely hands-on, technical phenomenon: missing values, malformed records, and cranky file formats. Sure, that's part of the picture, but Bad Data is so much more. It includes data that eats up your time, causes you to stay late at the office, drives you to tear out your hair in frustration. It's data that you can't access, data that you had and then lost, data that's not the same today as it was yesterday…

In short, Bad Data is *data that gets in the way*. There are so many ways to get there, from cranky storage, to poor representation, to misguided policy. If you stick with this data science bit long enough, you'll certainly encounter your fair share.

To that end, we decided to compile *Bad Data Handbook,* a rogues gallery of data troublemakers. We found 19 people from all reaches of the data arena to talk about how data issues have bitten them, and how they've healed.

In particular:

Guidance for Grubby, Hands-on Work

You can't assume that a new dataset is clean and ready for analysis. Kevin Fink's *Is It Just Me, or Does This Data Smell Funny?* (Chapter 2) offers several techniques to take the data for a test drive.

There's plenty of data trapped in spreadsheets, a format as prolific as it is inconvenient for analysis efforts. In *Data Intended for Human Consumption, Not Machine Consumption* (Chapter 3), Paul Murrell shows off moves to help you extract that data into something more usable.

If you're working with text data, sooner or later a character encoding bug will bite you. *Bad Data Lurking in Plain Text* (Chapter 4), by Josh Levy, explains what sort of problems await and how to handle them.

To wrap up, Adam Laiacano's *(Re)Organizing the Web's Data* (Chapter 5) walks you through everything that can go wrong in a web-scraping effort.

Data That Does the Unexpected

Sure, people lie in online reviews. Jacob Perkins found out that people lie in some very strange ways. Take a look at *Detecting Liars and the Confused in Contradictory Online Reviews* (Chapter 6) to learn how Jacob's natural-language programming (NLP) work uncovered this new breed of lie.

Of all the things that can go wrong with data, we can at least rely on unique identifiers, right? In *When Data and Reality Don't Match* (Chapter 9), Spencer Burns turns to his experience in financial markets to explain why that's not always the case.

Approach

The industry is still trying to assign a precise meaning to the term "data scientist," but we all agree that writing software is part of the package. Richard Cotton's *Blood, Sweat, and Urine* (Chapter 8) offers sage advice from a software developer's perspective.

Philipp K. Janert questions whether there is such a thing as truly bad data, in *Will the Bad Data Please Stand Up?* (Chapter 7).

Your data may have problems, and you wouldn't even know it. As Jonathan A. Schwabish explains in *Subtle Sources of Bias and Error* (Chapter 10), how you collect that data determines what will hurt you.

In *Don't Let the Perfect Be the Enemy of the Good: Is Bad Data Really Bad?* (Chapter 11), Brett J. Goldstein's career retrospective explains how dirty data will give your classical statistics training a harsh reality check.

Data Storage and Infrastructure

How you store your data weighs heavily in how you can analyze it. Bobby Norton explains how to spot a graph data structure that's trapped in a relational database in *Crouching Table, Hidden Network* (Chapter 13).

Cloud computing's scalability and flexibility make it an attractive choice for the demands of large-scale data analysis, but it's not without its faults. In *Myths of Cloud Computing* (Chapter 14), Steve Francia dissects some of those assumptions so you don't have to find out the hard way.

We debate using relational databases over NoSQL products, Mongo over Couch, or one Hadoop-based storage over another. Tim McNamara's *When Databases Attack: A Guide for When to Stick to Files* (Chapter 12) offers another, simpler option for storage.

The Business Side of Data

Sometimes you don't have enough work to hire a full-time data scientist, or maybe you need a particular skill you don't have in-house. In *How to Feed and Care for Your Machine-Learning Experts* (Chapter 16), Pete Warden explains how to outsource a machine-learning effort.

Corporate bureaucracy policy can build roadblocks that inhibit you from even analyzing the data at all. Marck Vaisman uses *The Dark Side of Data Science* (Chapter 15) to document several worst practices that you should avoid.

Data Policy

Sure, you know the methods you used, but do you truly understand how those final figures came to be? Reid Draper's *Data Traceability* (Chapter 17) is food for thought for your data processing pipelines.

Data is particularly bad when it's in the wrong place: it's supposed to be inside but it's gotten outside, or it still exists when it's supposed to have been removed. In *Social Media: Erasable Ink?* (Chapter 18), Jud Valeski looks to the future of social media, and thinks through a much-needed recall feature.

To close out the book, I pair up with longtime cohort Ken Gleason on *Data Quality Analysis Demystified: Knowing When Your Data Is Good Enough* (Chapter 19). In this complement to Kevin Fink's article, we explain how to assess your data's quality, and how to build a structure around a data quality effort.

CHAPTER 2

Is It Just Me, or Does This Data Smell Funny?

Kevin Fink

You are given a dataset of unknown provenance. How do you know if the data is any good?

It is not uncommon to be handed a dataset without a lot of information as to where it came from, how it was collected, what the fields mean, and so on. In fact, it's probably more common to receive data in this way than not. In many cases, the data has gone through many hands and multiple transformations since it was gathered, and nobody really knows what it all means anymore. In this chapter, I'll walk you through a step-by-step approach to understanding, validating, and ultimately turning a dataset into usable information. In particular, I'll talk about specific ways to look at the data, and show some examples of what I learned from doing so.

As a bit of background, I have been dealing with quite a variety of data for the past 25 years or so. I've written code to process accelerometer and hydrophone signals for analysis of dams and other large structures (as an undergraduate student in Engineering at Harvey Mudd College), analyzed recordings of calls from various species of bats (as a graduate student in Electrical Engineering at the University of Washington), built systems to visualize imaging sonar data (as a Graduate Research Assistant at the Applied Physics Lab), used large amounts of crawled web content to build content filtering systems (as the co-founder and CTO of N2H2, Inc.), designed intranet search systems for portal software (at DataChannel), and combined multiple sets of directory assistance data into a searchable website (as CTO at WhitePages.com). For the past five years or so, I've spent most of my time at Demand Media using a wide variety of data sources to build optimization systems for advertising and content recommendation systems, with various side excursions into large-scale data-driven search engine optimization (SEO) and search engine marketing (SEM).

Most of my examples will be related to work I've done in Ad Optimization, Content Recommendation, SEO, and SEM. These areas, as with most, have their own terminology, so a few term definitions may be helpful.

Table 2-1. Term Definitions

Term	Definition
PPC	Pay Per Click—Internet advertising model used to drive traffic to websites with a payment model based on clicks on advertisements. In the data world, it is used more specifically as Price Per Click, which is the amount paid per click.
RPM	Revenue Per 1,000 Impressions (usually ad impressions).
CTR	Click Through Rate—Ratio of Clicks to Impressions. Used as a measure of the success of an advertising campaign or content recommendation.
XML	Extensible Markup Language—Text-based markup language designed to be both human and machine-readable.
JSON	JavaScript Object Notation—Lightweight text-based open standard designed for human-readable data interchange. Natively supported by JavaScript, so often used by JavaScript widgets on websites to communicate with back-end servers.
CSV	Comma Separated Value—Text file containing one record per row, with fields separated by commas.

Understand the Data Structure

When receiving a dataset, the first hurdle is often basic accessibility. However, I'm going to skip over most of these issues and assume that you can read the physical medium, uncompress or otherwise extract the files, and get it into a readable format of some sort. Once that is done, the next important task is to understand the structure of the data. There are many different data structures commonly used to transfer data, and many more that are (thankfully) used less frequently. I'm going to focus on the most common (and easiest to handle) formats: columnar, XML, JSON, and Excel.

The single most common format that I see is some version of columnar (i.e., the data is arranged in rows and columns). The columns may be separated by tabs, commas, or other characters, and/or they may be of a fixed length. The rows are almost always separated by newline and/or carriage return characters. Or for smaller datasets the data may be in a proprietary format, such as those that various versions of Excel have used, but are easily converted to a simpler textual format using the appropriate software. I often receive Excel spreadsheets, and almost always promptly export them to a tab-delimited text file.

Comma-separated value (CSV) files are the most common. In these files, each record has its own line, and each field is separated by a comma. Some or all of the values (particularly commas within a field) may also be surrounded by quotes or other characters to protect them. Most commonly, double quotes are put around strings containing commas when the comma is used as the delimiter. Sometimes all strings are protected; other times only those that include the delimiter are protected. Excel can automatically load CSV files, and most languages have libraries for handling them as well.

In the example code below, I will be making occasional use of some basic UNIX commands: particularly *echo* and *cat*. This is simply to provide clarity around sample data. Lines that are meant to be typed or at least understood in the context of a UNIX shell start with the dollar-sign ($) character. For example, because tabs and spaces look a lot alike on the page, I will sometimes write something along the lines of

```
$ echo -e 'Field 1\tField 2\nRow 2\n'
```

to create sample data containing two rows, the first of which has two fields separated by a tab character. I also illustrate most pipelines verbosely, by starting them with

```
$ cat filename |
```

even though in actual practice, you may very well just specify the filename as a parameter to the first command. That is,

```
$ cat filename | sed -e 's/cat/dog/'
```

is functionally identical to the shorter (and slightly more efficient)

```
$ sed -e 's/cat/dog/' filename
```

Here is a Perl one-liner that extracts the third and first columns from a CSV file:

```
$ echo -e 'Column 1,"Column 2, protected","Column 3"'
Column 1,"Column 2, protected","Column 3"

$ echo -e 'Column 1,"Column 2, protected","Column 3"' | \
        perl -MText::CSV -ne '
        $csv = Text::CSV->new();
        $csv->parse($_); print join("\t",($csv->fields())[2,0]);'
Column 3        Column 1
```

Here is a more readable version of the Perl script:

```
use Text::CSV;

while(<>) {
 my $csv = Text::CSV->new();
 $csv->parse($_);
 my @fields = $csv->fields();
 print join("\t",@fields[2,0]),"\n";
}
```

Most data does not include tab characters, so it is a fairly safe and therefore popular, delimiter. Tab-delimited files typically completely disallow tab characters in the data itself, so don't use quotes or escape sequences, making them easier to work with than CSV files. They can be easily handled by typical UNIX command line utilities such as perl, awk, cut, join, comm, and the like, and many simple visualization tools such as Excel can semi-automatically import tab-separated-value files, putting each field into a separate column for easy manual fiddling.

Here are some simple examples of printing out the first and third columns of a tab-delimited string. The *cut* command will only print out data in the order it appears, but other tools can rearrange it. Here are examples of *cut* printing the first and third columns, and *awk* and *perl* printing the third and first columns, in that order:

```
$ echo -e 'Column 1\tColumn 2\tColumn 3\n'
Column 1     Column 2     Column 3
```

cut:

```
$ echo -e 'Column 1\tColumn 2\tColumn 3\n' | \
        cut -f1,3
Column 1        Column 3
```

awk:

```
$ echo -e 'Column 1\tColumn 2\tColumn 3\n' | \
        awk -F"\t" -v OFS="\t" '{ print $3,$1 }'
Column 3        Column 1
```

perl:

```
$ echo -e 'Column 1\tColumn 2\tColumn 3\n' | \
        perl -a -F"\t" -n -e '$OFS="\t"; print @F[2,0],"\n"'
Column 3        Column 1
```

In some arenas, XML is a common data format. Although they haven't really caught on widely, some databases (e.g., BaseX) store XML internally, and many can export data in XML. As with CSV, most languages have libraries that will parse it into native data structures for analysis and transformation.

Here is a Perl one-liner that extracts fields from an XML string:

```
$ echo -e '<config>\n\t<key name="key1" value="value 1">
\n\t<description>Description 1</description>
\n\t</key>\n</config>'
<config>
     <key name="key1" value="value 1">
     <description>Description 1</description>
     </key>
</config>

$ echo '<config><key name="key1" value="value 1">
        <description>Description 1</description>
        </key></config>' | \
        perl -MXML::Simple -e 'my $ref = XMLin(<>);
        print $ref->{"key"}->{"description"}'
Description 1
```

Here is a more readable version of the Perl script:

```
use XML::Simple;

my $ref = XMLin(join('',<>));
print $ref->{"key"}->{"description"}';
```

Although primarily used in web APIs to transfer information between servers and JavaScript clients, JSON is also sometimes used to transfer bulk data. There are a number of databases that either use JSON internally (e.g., CouchDB) or use a serialized form of it (e.g., MongoDB), and thus a data dump from these systems is often in JSON.

Here is a Perl one-liner that extracts a node from a JSON document:

```
$ echo '{"config": {"key1":"value 1","description":"Description 1"}}'
{"config": {"key1":"value 1","description":"Description 1"}}

$ echo '{"config": {"key1":"value 1","description":"Description 1"}}' | \
        perl -MJSON::XS -e 'my $json = decode_json(<>);
        print $json->{"config"}->{"description"}'
Description 1
```

Here is a more readable version of the Perl script:

```
use JSON::XS;

my $json = decode_json(join('',<>));
print $json->{"config"}->{"description"}';
```

Field Validation

Once you have the data in a format where you can view and manipulate it, the next step is to figure out what the data means. In some (regrettably rare) cases, all of the information about the data is provided. Usually, though, it takes some sleuthing. Depending on the format of the data, there may be a header row that can provide some clues, or each data element may have a key. If you're lucky, they will be reasonably verbose and in a language you understand, or at least that someone you know can read. I've asked my Russian QA guy for help more than once. This is yet another advantage of diversity in the workplace!

One common error is misinterpreting the units or meaning of a field. Currency fields may be expressed in dollars, cents, or even micros (e.g., Google's AdSense API). Revenue fields may be gross or net. Distances may be in miles, kilometers, feet, and so on. Looking at both the definitions and actual values in the fields will help avoid misinterpretations that can lead to incorrect conclusions.

You should also look at some of the values to make sure they make sense in the context of the fields. For example, a PageView field should probably contain integers, not decimals or strings. Currency fields (prices, costs, PPC, RPM) should probably be decimals with two to four digits after the decimal. A User Agent field should contain strings that look like common user agents. IP addresses should be integers or dotted quads.

A common issue in datasets is missing or empty values. Sometimes these are fine, while other times they invalidate the record. These values can be expressed in many ways. I've seen them show up as nothing at all (e.g., consecutive tab characters in a tab-delimited file), an empty string (contained either with single or double quotes), the explicit string NULL or undefined or N/A or NaN, and the number 0, among others. No matter how they appear in your dataset, knowing what to expect and checking to make sure the data matches that expectation will reduce problems as you start to use the data.

Value Validation

I often extend these anecdotal checks to true validation of the fields. Most of these types of validations are best done with regular expressions. For historical reasons (i.e., I've been using it for 20-some years), I usually write my validation scripts in Perl, but there are many good choices available. Virtually every language has a regular expression implementation.

For enumerable fields, do all of the values fall into the proper set? For example, a "month" field should only contain months (integers between 0 and 12, string values of *Jan*, *Feb*, … or *January*, *February*, …).

```
my $valid_month = map { $_ => 1 } (0..12,
qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
January February March April May June July August
September October November December));
print "Invalid!" unless($valid_month{$month_to_check});
```

For numeric fields, are all of the values numbers? Here is a check to see if the third column consists entirely of digits.

```
$ cat sample.txt
one     two     3
one     two     three
1       2       3
1       2       three

$ cat sample.txt | \
        perl -ape 'warn if $F[2] !~ /^\d+$/'
one     two     3
Warning: something's wrong at -e line 1, <> line 2.
one     two     three
1       2       3
Warning: something's wrong at -e line 1, <> line 4.
1       2       three
```

For fixed-format fields, do all of the values match a regular expression? For example, IP addresses are often shown as dotted quads (e.g., 127.0.0.1), which can be matched with something like ^\d+\.\d+\.\d+\.\d+$ (or more rigorous variants).

```
$ cat sample.txt
fink.com     127.0.0.1
bogus.com     1.2.3

$ cat sample.txt | \
        perl -ape 'warn "Invalid IP!" if $F[1] !~ /^\d+\.\d+\.\d+\.\d+$/'
fink.com     127.0.0.1
Invalid IP! at -e line 1, <> line 2.
bogus.com     1.2.3
```

Physical Interpretation of Simple Statistics

For numeric fields, I like to do some simple statistical checks. Does the minimum value make sense in the context of the field? The minimum value of a counter (number of clicks, number of pageviews, and so on) should be 0 or greater, as should many other types of fields (e.g., PPC, CTR, CPM). Similarly, does the maximum value make sense? Very few fields can logically accommodate values in the billions, and in many cases much smaller numbers than that don't make sense.

Depending on the exact definition, a ratio like CTR should not exceed 1. Of course, no matter the definition, it often will (this book is about bad data, after all…), but it generally shouldn't be much greater than 1. Certainly if you see values in the hundreds or thousands, there is likely a problem.

Financial values should also have a reasonable upper bound. At least for the types of data I've dealt with, PPC or CPC values in the hundreds of dollars might make sense, but certainly not values in the thousands or more. Your acceptable ranges will probably be different, but whatever they are, check the data to make sure it looks plausible.

You can also look at the average value of a field (or similar statistic like the mode or median) to see if it makes sense. For example, if the sale price of a widget is somewhere around $10, but the average in your "Widget Price" field is $999, then something is not right. This can also help in checking units. Perhaps 999 is a reasonable value if that field is expressed in cents instead of dollars.

The nice thing about these checks is that they can be easily automated, which is very handy for datasets that are periodically updated. Spending a couple of hours checking a new dataset is not too onerous, and can be very valuable for gaining an intuitive feel for the data, but doing it again isn't nearly as much fun. And if you have an hourly feed, you might as well work as a Jungle Cruise tour guide at Disneyland ("I had so much fun, I'm going to go again! And again! And again…").

Visualization

Another technique that I find very helpful is to create a histogram of the values in a data field. This is especially helpful for extremely large datasets, where the simple statistics discussed above barely touch the surface of the data. A histogram is a count of the number of times each unique value appears in a dataset, so it can be generated on nonnumeric values where the statistical approach isn't applicable.

For example, consider a dataset containing referral keywords, which are phrases searched for using Google, Bing, or some other search engine that led to pageviews on a site. A large website can receive millions of referrals from searches for hundreds of thousands of unique keywords per day, and over a reasonable span of time can see billions of unique keywords. We can't use statistical concepts like minimum, maximum, or average to summarize the data because the key field is nonnumeric: keywords are arbitrary strings of characters.

We can use a histogram to summarize this very large nonnumeric dataset. A first order histogram counts the number of referrals per keyword. However, if we have billions of keywords in our dataset, our histogram will be enormous and not terribly useful. We can perform another level of aggregation, using the number of referrals per keyword as the value, resulting in a much smaller and more useful summary. This histogram will show the number of keywords having each number of referrals. Because small differences in the number of referrals isn't very meaningful, we can further summarize by placing the values into bins (e.g., 1-10 referrals, 11-20 referrals, 21-29 referrals, and so on). The specific bins will depend on the data, of course.

For many simple datasets, a quick pipeline of commands can give you a useful histogram. For example, let's say you have a simple text file (*sample.txt*) containing some enumerated field (e.g., URLs, keywords, names, months). To create a quick histogram of the data, simply run:

```
$ cat sample.txt | sort | uniq -c
```

So, what's going on here? The *cat* command reads a file and sends the contents of it to *STDOUT*. The pipe symbol (|) catches this data and sends it on to the next command in the pipeline (making the pipe character an excellent choice!), in this case the *sort* command, which does exactly what you'd expect: it sorts the data. For our purposes we actually don't care whether or not the data is sorted, but we do need identical rows to be adjacent to each other, as the next command, *uniq*, relies on that. This (aptly named, although what happened to the "ue" at the end I don't know) command will output each unique row only once, and when given the *-c* option, will prepend it with the number of rows it saw. So overall, this pipeline will give us the number of times each row appears in the file: that is, a histogram!

Here is an example.

Example 2-1. Generating a Sample Histogram of Months

```
$ cat sample.txt
January
January
February
October
January
March
September
September
February

$ cat sample.txt | sort
February
February
January
January
January
March
October
September
September

$ cat sample.txt | sort | uniq -c
      2 February
      3 January
      1 March
      1 October
      2 September
```

For slightly more complicated datasets, such as a tab-delimited file, simply add a filter to extract the desired column. There are several (okay, many) options for extracting a column, and the "best" choice depends on the specifics of the data and the filter criteria. The simplest is probably the *cut* command, especially for tab-delimited data. You simply specify which column (or columns) you want as a command line parameter. For example, if we are given a file containing names in the first column and ages in the second column and asked how many people are of each age, we can use the following code:

```
$ cat sample2.txt
Joe     14
Marci   17
Jim     16
Bob     17
Liz     15
Sam     14

$ cat sample2.txt | cut -f2 | sort | uniq -c
```

```
      2 14
      1 15
      1 16
      2 17
```

The first column contains the counts, the second the ages. So two kids are 14-year-olds, one is 15-year-old, one is 16-year-old, and two are 17-year-olds.

The *awk* language is another popular choice for selecting columns (and can do much, much more), albeit with a slightly more complicated syntax:

```
$ cat sample2.txt | \
    awk '{print $2}' | sort | uniq -c
      2 14
      1 15
      1 16
      2 17
```

As with virtually all text-handling tasks, Perl can also be used (and as with anything in Perl, there are many ways to do it). Here are a few examples:

```
$ cat sample2.txt | \
    perl -ane 'print $F[1],"\n"' | sort | uniq -c
      2 14
      1 15
      1 16
      2 17

$ cat sample2.txt | \
    perl -ne 'chomp; @F = split(/\t/,$_); print $F[1],"\n"' | sort | uniq -c
      2 14
      1 15
      1 16
      2 17
```

For real datasets (i.e., ones consisting of lots of data points), a histogram provides a reasonable approximation of the distribution function and can be assessed as such. For example, you typically expect a fairly smooth function. It may be flat, or Gaussian (looks like a bell curve), or even decay exponentially (long-tail), but a discontinuity in the graph should be viewed with suspicion: it may indicate some kind of problem with the data.

Keyword PPC Example

One example of a histogram study that I found useful was for a dataset consisting of estimated PPC values for two sets of about 7.5 million keywords. The data had been collected by third parties and I was given very little information about the methodology they used to collect it. The data files were comma-delimited text files of keywords and corresponding PPC values.

Example 2-2. PPC Data File

```
waco tourism, $0.99
calibre cpa, $1.99,,,,,
c# courses,$2.99 ,,,,,
cad computer aided dispatch, $1.49 ,,,,,
cadre et album photo, $1.39 ,,,,,
cabana beach apartments san marcos, $1.09,,,
"chemistry books, a level", $0.99
cake decorating classes in san antonio, $1.59 ,,,,,
k & company, $0.50
p&o mini cruises, $0.99
c# data grid,$1.79 ,,,,,
advanced medical imaging denver, $9.99 ,,,,,
canadian commercial lending, $4.99 ,,,,,
cabin vacation packages, $1.89 ,,,,,
c5 envelope printing, $1.69 ,,,,,
mesothelioma applied research, $32.79 ,,,,,
ca antivirus support, $1.29 ,,,,,
"trap, toilet", $0.99
c fold towels, $1.19 ,,,,,
cabin rentals wa, $0.99
```

Because this dataset was in CSV (including some embedded commas in quoted fields), the quick tricks described above don't work perfectly. A quick first approximation can be done by removing those entries with embedded commas, then using a pipeline similar to the above. We'll do that by skipping the rows that contain the double-quote character. First, though, let's check to see how many records we'll skip.

```
$ cat data*.txt | grep -c '"'
5505
$ cat data*.txt | wc -l
7533789
```

We only discarded 0.07% of the records, which isn't going to affect anything, so our pipeline is:

```
$ cat ppc_data_sample.txt | grep -v '"' | cut -d, -f2 | sort | uniq -c | sort -k2
      1  $0.50
      3  $0.99
      1  $1.09
      1  $1.19
      1  $1.29
      1  $1.39
      1  $1.49
      1  $1.59
      1  $1.69
      1  $1.79
      1  $1.89
      1  $1.99
```

```
1  $2.99
1  $32.79
1  $4.99
1  $9.99
```

This may look a little complicated, so let's walk through it step-by-step. First, we create a data stream by using the *cat* command and a shell glob that matches all of the data files. Next, we use the *grep* command with the *-v* option to remove those rows that contain the double-quote character, which the CSV format uses to encapsulate the delimiter character (the comma, in our case) when it appears in a field. Then we use the *cut* command to extract the second field (where fields are defined by the comma character). We then sort the resulting rows so that duplicates will be in adjacent rows. Next we use the *uniq* command with the *-c* option to count the number of occurrences of each row. Finally, we sort the resulting output by the second column (the PPC value).

In reality, this results in a pretty messy outcome, because the format of the PPC values varies (some have white space between the comma and dollar sign, some don't, among other variations). If we want cleaner output, as well as a generally more flexible solution, we can write a quick Perl script to clean and aggregate the data:

```
use strict;
use warnings;

use Text::CSV_XS;

my $csv = Text::CSV_XS->new({binary=>1});
my %count;

while(<>) {
  chomp;
  s/\cM//;
  $csv->parse($_) || next;
  my $ppc = ($csv->fields())[1];
  $ppc =~ s/^[ \$]+//;
  $count{$ppc}++;
}

foreach my $ppc (sort {$a <=> $b} keys %count) {
  print "$ppc\t$count{$ppc}\n";
}
```

For our sample dataset shown above, this results in a similar output, but with the two discarded $0.99 records included and the values as actual values rather than strings of varying format:

```
0.50    1
0.99    5
1.09    1
1.19    1
1.29    1
1.39    1
```

```
1.49    1
1.59    1
1.69    1
1.79    1
1.89    1
1.99    1
2.99    1
4.99    1
9.99    1
32.79   1
```

For the real dataset, the output looks like:

```
0.05    1071347
0.06    2993
0.07    3359
0.08    3876
0.09    4803
0.10    443838
0.11    28565
0.12    32335
0.13    36113
0.14    42957
0.15    50026
...
23.97   1
24.64   1
24.82   1
25.11   1
25.93   1
26.07   1
26.51   1
27.52   1
32.79   1
```

As an aside, if your data is already in a SQL database, generating a histogram is very easy. For example, assume we have a table called MyTable containing the data described above, with two columns: Term and PPC. We simply aggregate by PPC:

```
SELECT PPC, COUNT(1) AS Terms
FROM MyTable
GROUP BY PPC
ORDER BY PPC ASC
```

No matter how we generate this data, the interesting features can most easily be visualized by graphing it, as shown in Figure 2-1.

There are a lot of keywords with relatively small PPC values, and then an exponential decay (note that the vertical axis is on a logarithmic scale) as PPC values increase. However, there is a big gap in the middle of the graph! There are almost no PPC values between $15.00 and $15.88, and then more than expected (based on the shape of the curve) from $15.89 to about $18.00, leading me to hypothesize that the methodology

used to generate this data shifted everything between $15.00 and $15.88 up by $0.89 or so. After talking to the data source, we found out two things. First, this was indeed due to the algorithm they used to test PPC values. Second, they had no idea that their algorithm had this unfortunate characteristic! By doing this analysis we knew to avoid ascribing relative values to any keywords with PPC values between $15.89 and $18.00, and they knew to fix their algorithm.

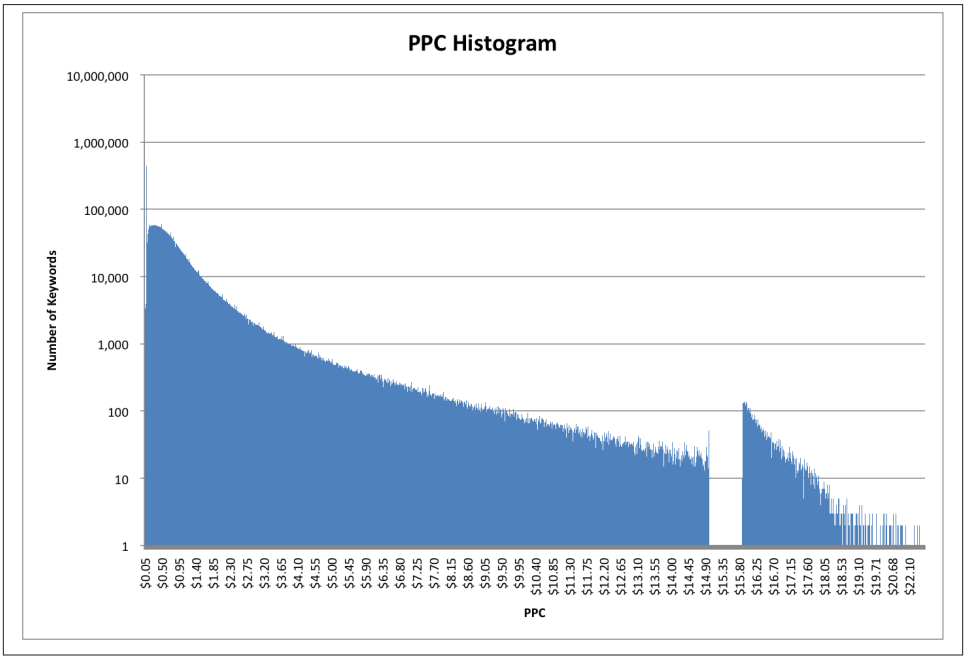


Figure 2-1. PPC Histogram Overview

Another interesting feature of this dataset is that the minimum value is $0.05. This could be caused by the marketplace being measured as having a minimum bid, or the algorithm estimating the bids starting at $0.05, or the data being post-filtered to remove bids below $0.05, or perhaps other explanations. In this case, it turned out to be the first option: the marketplace where the data was collected had a minimum bid of five cents. In fact, if we zoom in on the low-PPC end of the histogram (Figure 2-2), we can see another interesting feature. Although there are over a million keywords with a PPC value of $0.05, there are virtually none (less than 3,000 to be precise) with a PPC value of $0.06, and similarly up to $0.09. Then there are quite a few (almost 500,000) at $0.10, and again fewer (less than 30,000) at $0.11 and up. So apparently the marketplace has two different minimum bids, depending on some unknown factor.

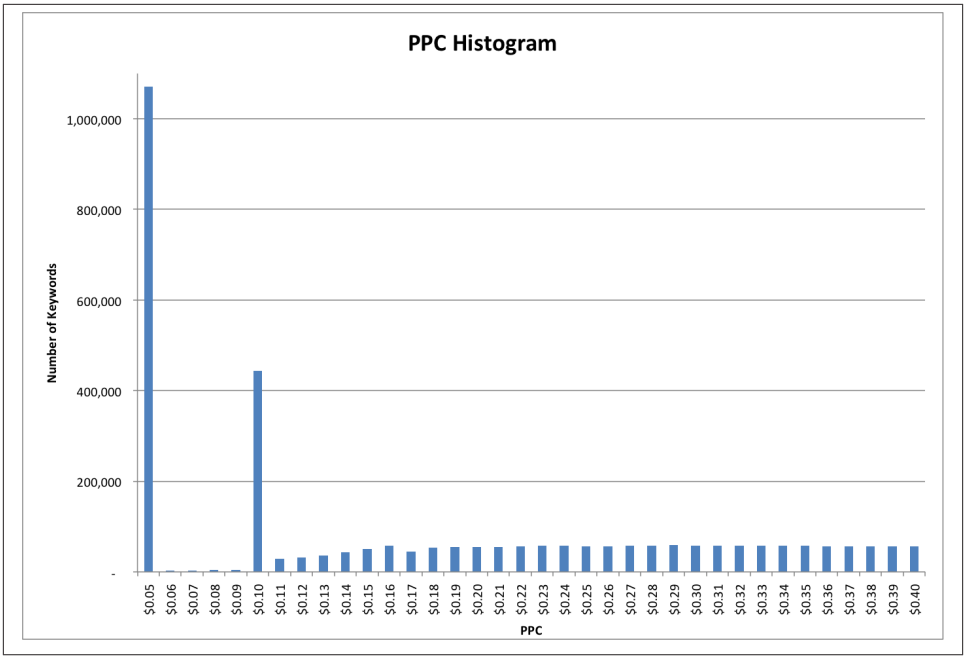


Figure 2-2. PPC Histogram Low Values

Search Referral Example

Another example of the usefulness of a histogram came from looking at search referral data. When users click on links to a website on a Google search results page, Google (sometimes) passes along the "rank" of the listing (1 for the first result on the page, 2 for the second, and so on) along with the query keyword. This information is very valuable to websites because it tells them how their content ranks in the Google results for various keywords. However, it can be pretty noisy data. Google is constantly testing their algorithms and user behavior by changing the order of results on a page. The order of results is also affected by characteristics of the specific user, such as their country, past search and click behavior, or even their friends' recommendations. As a result, this rank data will typically show many different ranks for a single keyword/URL combination, making interpretation difficult. Some people also contend that Google purposefully obfuscates this data, calling into question any usefulness.

In order to see if this rank data had value, I looked at the referral data from a large website with a significant amount of referral traffic (millions of referrals per day) from Google. Rather than the usual raw source of standard web server log files, I had the

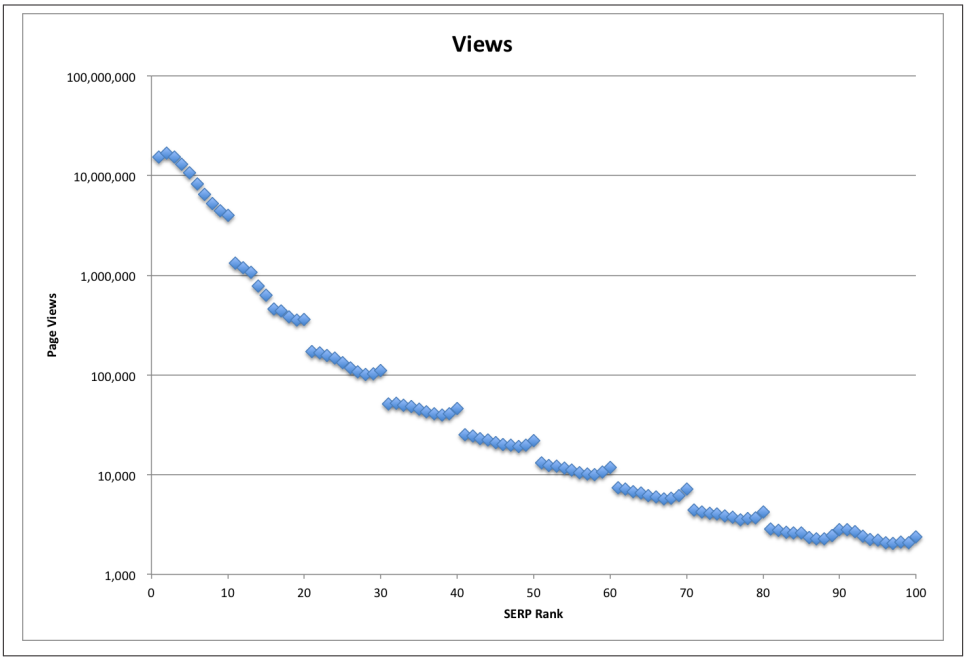


Figure 2-3. Search Referral Views by Rank

luxury of data already stored in a data warehouse, with the relevant fields already extracted out of the URL of the referring page. This gave me fields of date, URL, referring keyword, and rank for each pageview. I created a histogram showing the number of pageviews for each Rank (Figure 2-3):

Looking at the histogram, we can clearly see this data isn't random or severely obfuscated; there is a very clear pattern that corresponds to expected user behavior. For example, there is a big discontinuity between the number of views from Rank 10 vs the views from Rank 11, between 20 and 21, and so on. This corresponds to the Google's default of 10 results per page.

Within a page (other than the first—more on that later), we can also see that more users click on the first position on the page than the second, more on the second than the third, and so forth. Interestingly, more people click on the last couple of results than those "lost" in the middle of the page. This behavior has been well-documented by various other mechanisms, so seeing this fine-grained detail in the histogram lends a lot of credence to the validity of this dataset.

So why is this latter pattern different for the first page than the others? Remember that this data isn't showing CTR (click-through rate), it's showing total pageviews. This particular site doesn't have all that many pages that rank on the top of the first page for

high-volume terms, but it does have a fair number that rank second and third, so even though the CTR on the first position is the highest (as shown on the other pages), that doesn't show up for the first page. As the rank increases across the third, fourth, and subsequent pages, the amount of traffic flattens out, so the pageview numbers start to look more like the CTR.

Recommendation Analysis

Up to now, I've talked about histograms based on counts of rows sharing a common value in a column. As we've seen, this is useful in a variety of contexts, but for some use cases this method provides too much detail, making it difficult to see useful patterns. For example, let's look at the problem of analyzing recommendation patterns. This could be movie recommendations for a user, product recommendations for another product, or many other possibilities, but for this example I'll use article recommendations. Imagine a content-rich website containing millions of articles on a wide variety of topics. In order to help a reader navigate from the current article to another that they might find interesting or useful, the site provides a short list of recommendations based on manual curation by an editor, semantic similarity, and/or past traffic patterns.

We'll start with a dataset consisting of recommendation pairs: one recommendation per row, with the first column containing the URL of the source article and the second the URL of the destination article.

Example 2-3. Sample Recommendation File

```
http://example.com/fry_an_egg.html       http://example.com/boil_an_egg.html
http://example.com/fry_an_egg.html       http://example.com/fry_bacon.html
http://example.com/boil_an_egg.html      http://example.com/fry_an_egg.html
http://example.com/boil_an_egg.html      http://example.com/make_devilled_eggs.html
http://example.com/boil_an_egg.html      http://example.com/color_easter_eggs.html
http://example.com/color_easter_eggs.html        http://example.com/boil_an_egg.html
...
```

So readers learning how to fry an egg would be shown articles on boiling eggs and frying bacon, and readers learning how to boil an egg would be shown articles on frying eggs, making devilled eggs, and coloring Easter eggs.

For a large site, this could be a large-ish file. One site I work with has about 3.3 million articles, with up to 30 recommendations per article, resulting in close to 100 million recommendations. Because these are automatically regenerated nightly, it is important yet challenging to ensure that the system is producing reasonable recommendations. Manually checking a statistically significant sample would take too long, so we rely on statistical checks. For example, how are the recommendations distributed? Are there some articles that are recommended thousands of times, while others are never recommended at all?

We can generate a histogram showing how many times each article is recommended as described above:

Example 2-4. Generate a Recommendation Destination Histogram

```
$ cat recommendation_file.txt | cut -f2 | sort | uniq -c

2 http://example.com/boil_an_egg.html
1 http://example.com/fry_bacon.html
1 http://example.com/fry_an_egg.html
1 http://example.com/make_devilled_eggs.html
1 http://example.com/color_easter_eggs.html
```

"How to Boil an Egg" was recommended twice, while the other four articles were recommended once each. That's fine and dandy, but with 3.3M articles, we're going to have 3.3M rows in our histogram, which isn't very useful. Even worse, the keys are URLs, so there really isn't any way to combine them into bins like we would normally do with numeric keys. To provide a more useful view of the data, we can aggregate once more, creating a histogram of our histogram:

Example 2-5. Generate a Recommendation Destination Count Histogram

```
$ cat recommendation_file.txt \
| cut -f2 \
| sort \
| uniq -c \
| sort -n \
| awk '{print $1}' \
| uniq -c

4 1
1 2
```

Four articles were recommended once, and one article was recommended twice.

Using this same technique on a 33 million recommendation dataset (top 10 recommendations for each of 3.3 million articles), we get a graph like that in Figure 2-4, or if we convert it to a cumulative distribution, we get Figure 2-5.

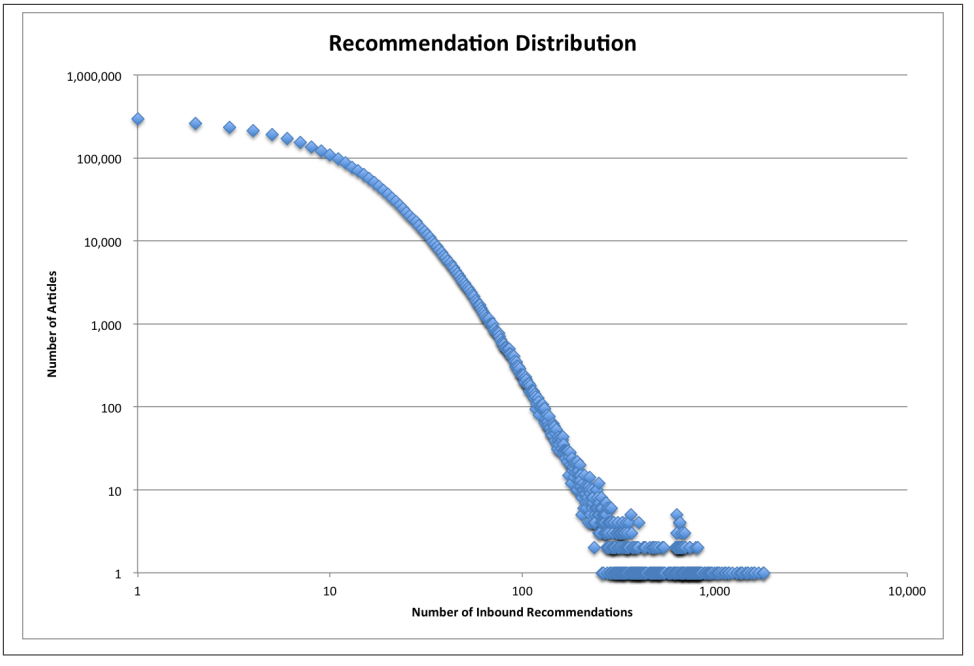


Figure 2-4. Recommendation Distribution

Note that the distribution graph uses a log-log scale while the cumulative distribution graph has a linear vertical scale. Overall, the dataset looks reasonable; we have a nice smooth curve with no big discontinuities. Most articles receive only a few recommendations, with about 300,000 receiving a single recommendation. The number of articles falls rapidly as the number of recommendations increases, with about half of the articles having seven or fewer recommendations. The most popular article receives about 1,800 recommendations.

We can run this analysis on recommendation sets generated by different algorithms in order to help us understand the behavior of the algorithms. For example, if a new algorithm produces a wider distribution, we know that more articles are receiving a disproportionate share of recommendations. Or if the distribution is completely different —say a bell curve around the average number of recommendations (10, in our example) —then the recommendations are being distributed very evenly.

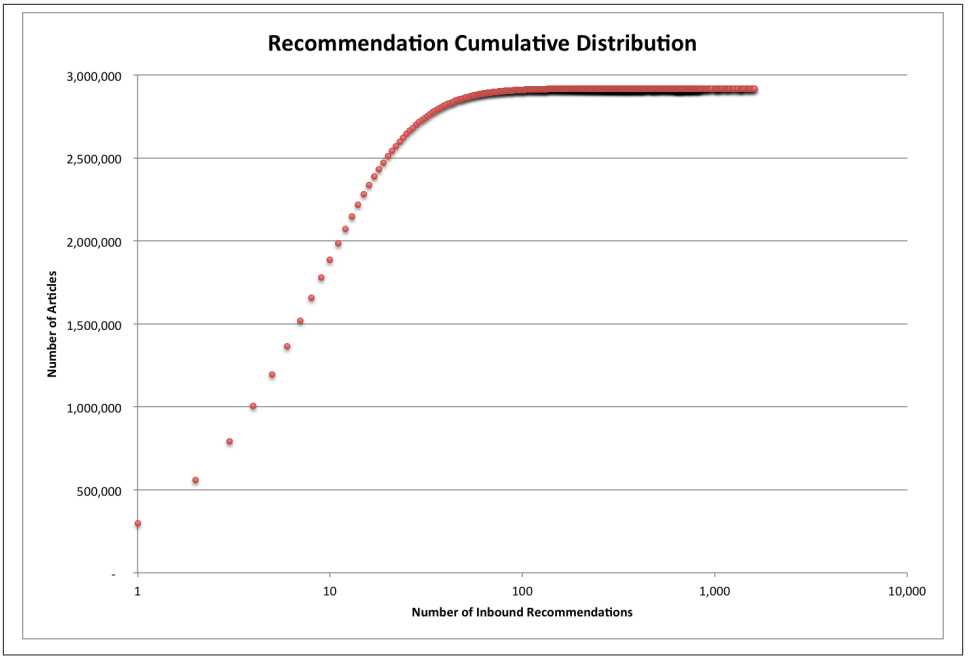


Figure 2-5. Recommendation Cumulative Distribution

Time Series Data

Although a histogram generally doesn't make sense for time-series data, similar visualization techniques can be helpful, particularly when the data can be aggregated across other dimensions. For example, web server logs can be aggregated up to pageviews per minute, hour, or day, then viewed as a time series. This is particularly useful for finding missing data, which can badly skew the conclusions of certain analyses if not handled properly.

Most website owners like to carefully watch their traffic volumes in order to detect problems on their site, with their marketing campaigns, with their search engine optimization, and so on. For many sites, this is greatly complicated by changes in traffic caused by factors completely independent of the site itself. Some of these, like seasonality, are somewhat predictable, which can reduce the number of panic drills over a traffic drop that turns out to have nothing to do with the site. Seasonality can be tricky, as it is in part a function of the type of site: a tax advice site will see very different seasonal traffic patterns than one on gardening, for example. In addition, common sense doesn't always provide a good guide and almost never provides a quantitative guide. Does a 30%

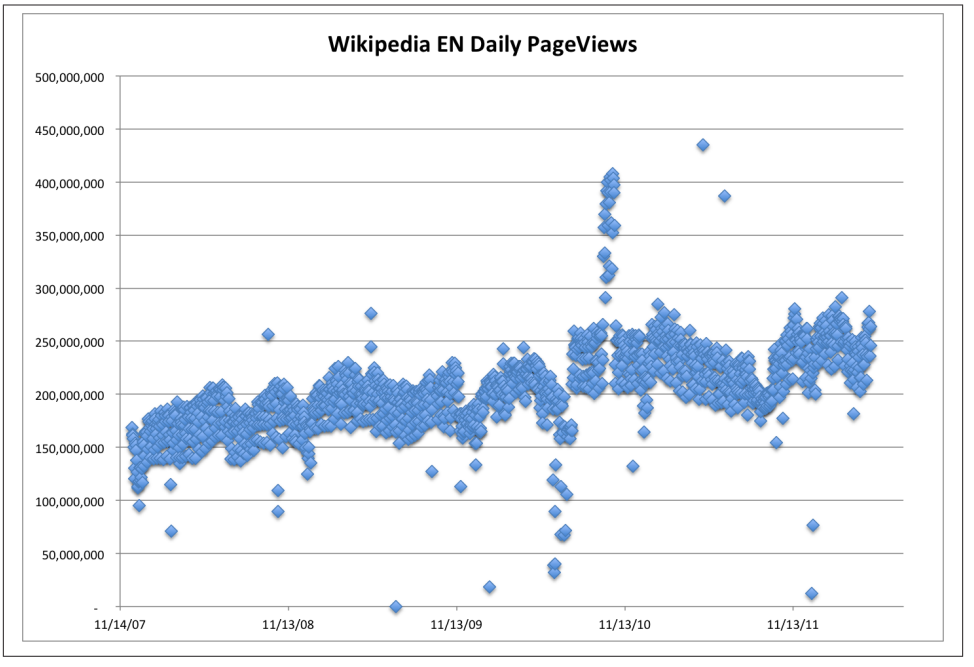


Figure 2-6. Wikipedia Daily Chart

drop in traffic on Superbowl Sunday make sense? When schools start to let out in the spring, how much will traffic drop? Sites with large amounts of historical traffic data can build models (explicitly or implicitly via "tribal knowledge"), but what about a new site or a translation of an existing site?

In order to address this, I tried using the publicly available Wikipedia logs to build a long-term, large-scale seasonality model for traffic in various languages. Wikipedia serves close to 500M pages per day, a bit more than half of them in English. An automated process provides aggregations of these by language (site), hour, and page. As with any automated process, it fails sometimes, which results in lower counts for a specific hour or day. This can cause havoc in a seasonality model, implying a drop in traffic that isn't real.

My first pass through the Wikipedia data yielded some nonsensical values. 1.06E69 pageviews on June 7th of 2011?!? That was easy to discard. After trimming out a few more large outliers, I had the time-series graph shown in Figure 2-6.

You can see that there are still some outliers, both low and high. In addition, the overall shape is very fuzzy. Zooming in (Figure 2-7), we can see that some of this fuzziness is due to variation by day of week. As with most informational sites, weekdays see a lot more traffic than weekends.

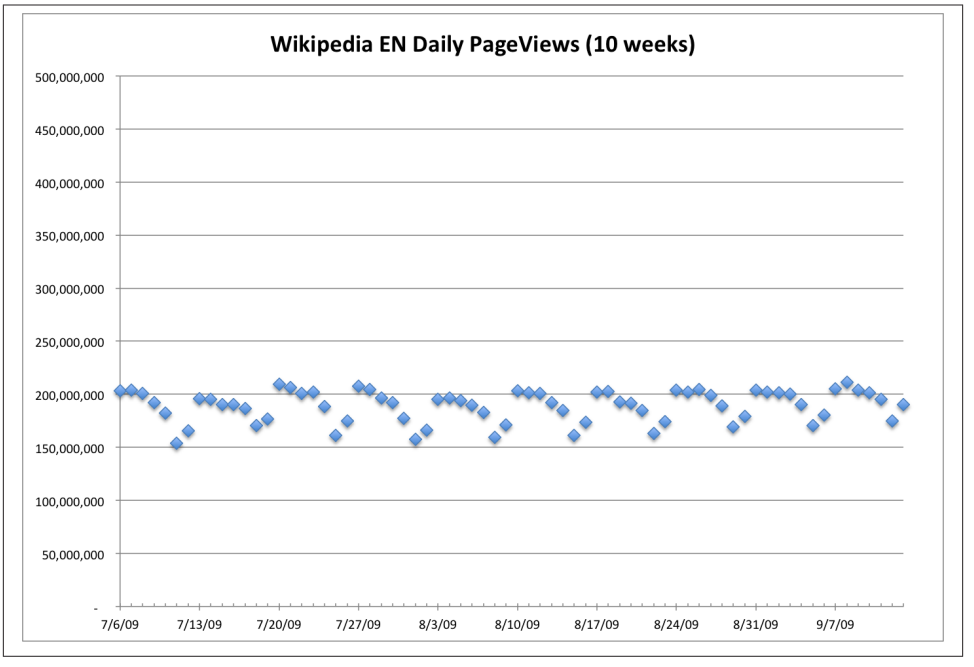


Figure 2-7. Wikipedia Daily Chart Zoom

Because we're looking for longer-term trends, I chose to look at the median of each week in order to both remove some of the outliers and get rid of the day-of-week variation (Figure 2-8).

Now we start to see a clearer picture. Overall traffic has been trending up over the past several years, as expected. There are still some pretty big dips and spikes, as well as a lot of variation over the course of each year. Because the data came in hourly files, one way we can detect bad data is by counting how many hours of data we have for each day. Obviously, there should be 24 hours per day (with some potential shifting around of Daylight Savings Time boundaries, depending on how Wikipedia logs).

Adding the number of hours to the graph results in Figure 2-9, which shows quite a few days with less than 24 hours, and a couple with more (up to 30!). May 11th, 2009, had 30 hours, and May 12th had 27. Closer inspection shows likely data duplication. For example, there are two files for hour 01 of 2009-05-11, each of which contains about the same number of pageviews as hour 00 and hour 02. It is likely that something in the system that generated these files from the raw web server logs duplicated some of the data.

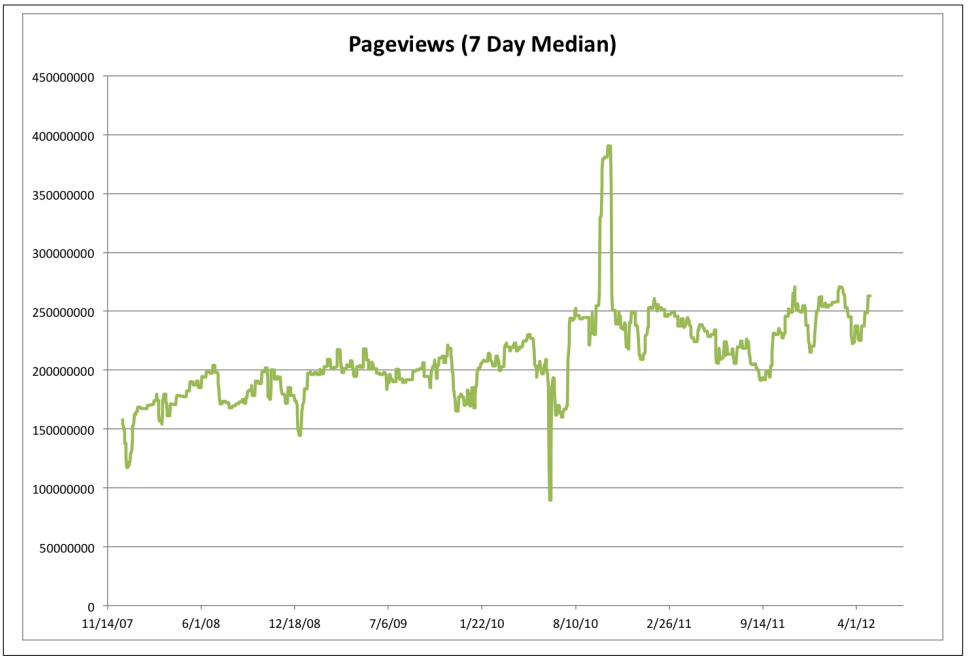


Figure 2-8. Wikipedia Seven Day Median

Example 2-6. Wikipedia EN Page Counts for May 11th, 2009

```
$ grep -P '^en\t' pagecounts-20090511-*.gz.summary
pagecounts-20090511-000000.gz.summary:en        20090511        9419692
pagecounts-20090511-010000.gz.summary:en        20090511        9454193
pagecounts-20090511-010001.gz.summary:en        20090511        8297669 <== duplicate
pagecounts-20090511-020000.gz.summary:en        20090511        9915606
pagecounts-20090511-030000.gz.summary:en        20090511        9855711
pagecounts-20090511-040000.gz.summary:en        20090511        9038523
pagecounts-20090511-050000.gz.summary:en        20090511        8200638
pagecounts-20090511-060000.gz.summary:en        20090511        7270928
pagecounts-20090511-060001.gz.summary:en        20090511        7271485 <== duplicate
pagecounts-20090511-070000.gz.summary:en        20090511        6750575
pagecounts-20090511-070001.gz.summary:en        20090511        6752474 <== duplicate
pagecounts-20090511-080000.gz.summary:en        20090511        6392987
pagecounts-20090511-090000.gz.summary:en        20090511        6581155
pagecounts-20090511-100000.gz.summary:en        20090511        6641253
pagecounts-20090511-110000.gz.summary:en        20090511        6826325
pagecounts-20090511-120000.gz.summary:en        20090511        7433542
pagecounts-20090511-130000.gz.summary:en        20090511        8560776
pagecounts-20090511-130001.gz.summary:en        20090511        8548498 <== duplicate
pagecounts-20090511-140000.gz.summary:en        20090511        9911342
pagecounts-20090511-150000.gz.summary:en        20090511        9708457
pagecounts-20090511-150001.gz.summary:en        20090511        10696488 <== duplicate
pagecounts-20090511-160000.gz.summary:en        20090511        11218779
```

```
pagecounts-20090511-170000.gz.summary:en        20090511        11241469
pagecounts-20090511-180000.gz.summary:en        20090511        11743829
pagecounts-20090511-190000.gz.summary:en        20090511        11988334
pagecounts-20090511-190001.gz.summary:en        20090511        10823951 <== duplicate
pagecounts-20090511-200000.gz.summary:en        20090511        12107136
pagecounts-20090511-210000.gz.summary:en        20090511        12145627
pagecounts-20090511-220000.gz.summary:en        20090511        11178888
pagecounts-20090511-230000.gz.summary:en        20090511        10131273
```

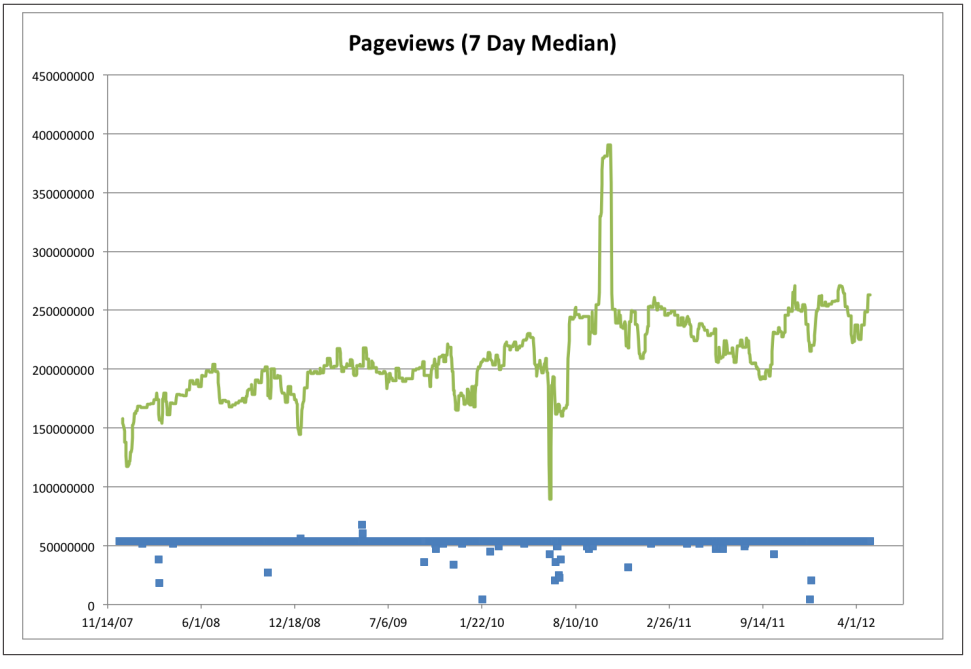


Figure 2-9. Wikipedia Seven Day Median with Hours

Removing these duplicate entries rationalizes the chart a bit more, but doesn't really address the largest variations in traffic volume, nor does estimating the data volumes for the missing files. In particular, there were 24 hourly files each day from June 12th through June 15th, 2010, yet that time period shows the largest drop in the graph. Similarly, the biggest positive outlier is the first three weeks of October, 2010, where the traffic surged from roughly 240M to 375M pageviews/day, yet there are no extra files during that period. So this points us to a need for deeper analysis of those time periods, and cautions us to be careful about drawing conclusions from this dataset.

Conclusion

Data seldom arrives neatly packaged with a quality guarantee. In fact, it often arrives with little or no documentation of where it came from, how it was gathered, or what to watch out for when using it. However, some relatively simple analysis techniques can provide a lot of insight into a dataset. This analysis will often provide interesting insights into an area of interest. At the minimum, these "sniff tests" provide areas to explore or watch out for.

CHAPTER 3

Data Intended for Human Consumption, Not Machine Consumption

Paul Murrell

This chapter describes issues that can arise when a dataset has been provided in a format that is designed mainly for consumption by human eyeballs.

Data is typically provided this way in order to allow a human to extract a particular message from the data.

The problem is that we inevitably end up wanting to do more with the data, which means working with the data using software, which means explaining the format of the data to the software, which in turn means that we end up wishing that the data were formatted for consumption by a computer, not human eyeballs.

The Data

The main high school qualification in New Zealand is called NCEA (National Certificate of Educational Achievement). A typical student will attempt to gain NCEA Level 1 in Year 11 (their eleventh year of formal education), NCEA Level 2 in Year 12, and Level 3 in Year 13. However, it is also possible for students to attempt NCEA levels in earlier years or to gain an NCEA level in a later year if they fail at the first attempt.

This leads to statistics on the number (or percentage) of students who have attained each level of NCEA by the end of each year of formal education (see Example 3-1).

Example 3-1. Number of students gaining NCEA in 2010 by level and year

```
               Year 11 Year 12 Year 13
NCEA (Level 1)   41072   46629   40088
NCEA (Level 2)    1050   37513   38209
NCEA (Level 3)      91     451   24688
```

The Problem: Data Formatted for Human Consumption

Tables of NCEA results can be obtained in the form of Excel spreadsheets, with a variety of filters and breakdowns available.[1] Figure 3-1 shows an example. In this case, the data is for boys-only schools and, for each combination of year and NCEA level, there are separate counts for male and female students.

	A	B	C	D	E	F	G	H	I
1	Qualifications by Year Level and Gender								
2					National				
3					Year 11		Year 12		Year 13
4	Qualificat	Gender							
5									
6	National Certificate of Educational Achievement								
7	NCEA (Level 1)								
8		Male			5,929		6,427		5,170
9		Female			0		60		38
10	NCEA (Level 2)								
11		Male			194		5,395		5,027
12		Female			0		58		38
13	NCEA (Level 3)								
14		Male			2		128		3,276
15		Female			0		0		36
16									

Figure 3-1. The number of students achieving NCEA at boys-only schools in 2010

This data is in a Microsoft Office Excel 97-2003 Worksheet format, which is unfortunate because that requires special software to open the file, ideally Microsoft Excel software. However, due to the prevalence of Microsoft Excel and Excel spreadsheets, many other software tools have been developed to work with this sort of file, so we will not focus on that particular problem here.

The issue that we wish to highlight instead is the *layout* of the data within the spreadsheet.

The Arrangement of Data

A human observer can interpret this table of information with relative ease. It is a simple matter to read off the number of male students who have attained NCEA Level 1 by Year

1. The New Zealand Qualifications Authority *http://www.nzqa.govt.nz/qualifications/ssq/statistics/provider-selected-report.do?reportID=1161649*

11 (5,929). If that is all that we want to extract from these data—if all we want to do is to read a few numbers from the table—then there is no problem. In other words, if this table of data is a useful *end result*, if all we want is a *textual presentation* of the data, then we can be satisfied with this spreadsheet.

However, data is much more useful to us if it can be used in other ways. This particular presentation of the data will only serve some people well for some purposes.

For example, some people would prefer to view these data as a plot or graph rather than as a table of text (see Figure 3-2). Different arrangements of the data also make it easier to make different sorts of comparisons. For example, it would be easier to focus on just the male students if the table was arranged as shown in Example 3-2.

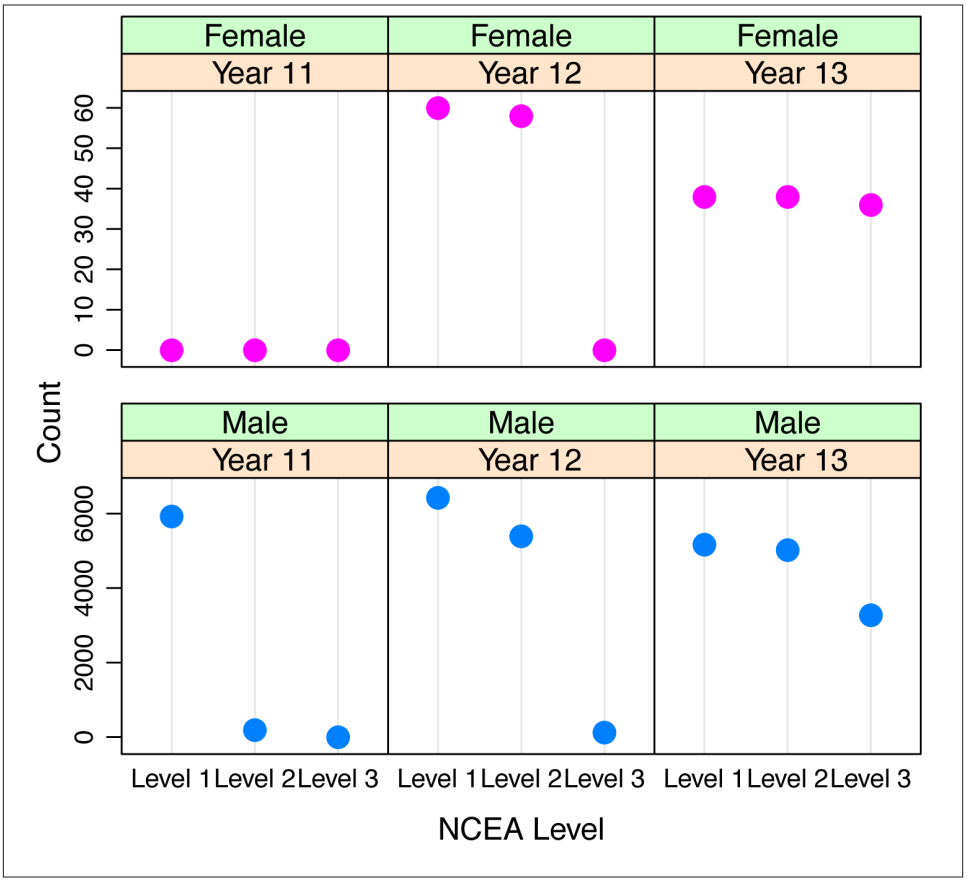


Figure 3-2. A plot of the data from Figure 3-1

Example 3-2. The data from Figure 3-1 rearranged so that all values for the same gender are next to each other

Gender	Level	Year 11	Year 12	Year 13
Female	NCEA Level 1	0	60	38
Female	NCEA Level 2	0	58	38
Female	NCEA Level 3	0	0	36
Male	NCEA Level 1	5929	6427	5170
Male	NCEA Level 2	194	5395	5027
Male	NCEA Level 3	2	128	3276

Unfortunately, this sort of reuse of the data is severely hampered by the layout of the data within the spreadsheet. The layout of the data is an obstacle because we need to be able to work with the data in software in order to do things like draw plots or rearrange the data and in order to do what we need to explain the arrangement of the data to the software. Explaining the arrangement of the data within this spreadsheet is not easy because it is not easy to specify where the data values are within the spreadsheet and what the different values represent.

To demonstrate this idea, Figure 3-3 highlights the location of the data values within the spreadsheet that correspond to the number of students who have attained NCEA Level 1. The association between the label "NCEA (Level 1)" and the corresponding data is not obvious because the label is neither on the same row nor in the same column as any of the data to which it refers.

It also does not help that the data values are not in a contiguous block. There are empty columns that do not represent missing data; the empty columns are just for visual appearance.

Given a label within the spreadsheet, it is difficult to describe to software the location of the data that are associated with that label.

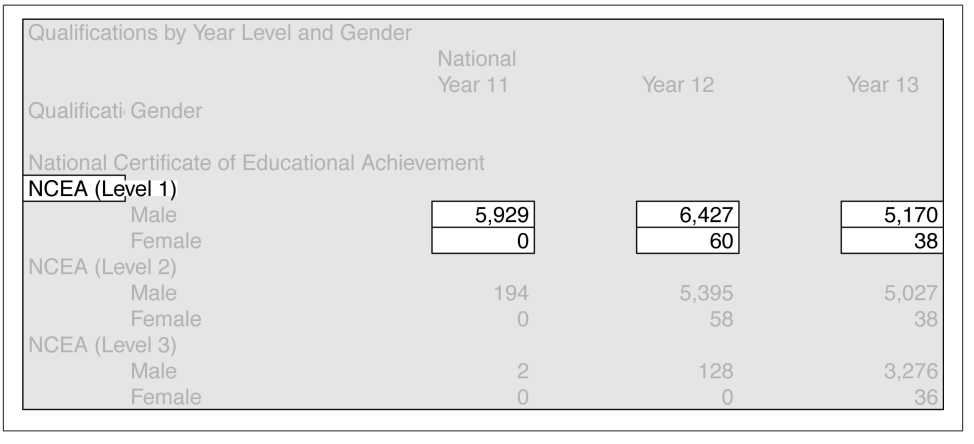


Figure 3-3. The data that corresponds to NCEA Level 1 in Figure 3-1

Another demonstration is shown in Figure 3-4. In this case, all of the data that relates to the top-left count in the spreadsheet, 5,929, have been highlighted. Some of the labels that correspond to this number, "Male" and "Year 11", are on the same row and column as the number, but even those are neither immediately adjacent to the number nor at some obvious limit location as the first row or first column in the spreadsheet.

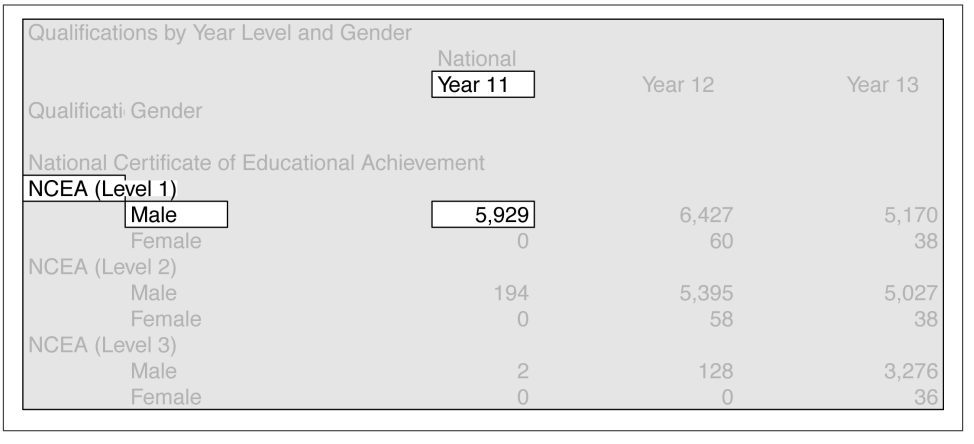


Figure 3-4. The data that corresponds to the top-left count in Figure 3-1

Given a data value, it is difficult to describe to software the location of the labels that are associated with that data value.

To drive the point home, Example 3-3 shows how the data could be arranged so that the association between labels and data values is very straightforward. In this arrangment, every label that corresponds to a data value lies on the same row as that data value.

Example 3-3. The data in a nicer format

```
Gender     Qualification       Year   Value
  Male    NCEA (Level 1)    Year 11    5929
Female    NCEA (Level 1)    Year 11       0
  Male    NCEA (Level 2)    Year 11     194
Female    NCEA (Level 2)    Year 11       0
  Male    NCEA (Level 3)    Year 11       2
Female    NCEA (Level 3)    Year 11       0
  Male    NCEA (Level 1)    Year 12    6427
Female    NCEA (Level 1)    Year 12      60
  Male    NCEA (Level 2)    Year 12    5395
Female    NCEA (Level 2)    Year 12      58
  Male    NCEA (Level 3)    Year 12     128
Female    NCEA (Level 3)    Year 12       0
  Male    NCEA (Level 1)    Year 13    5170
Female    NCEA (Level 1)    Year 13      38
  Male    NCEA (Level 2)    Year 13    5027
Female    NCEA (Level 2)    Year 13      38
  Male    NCEA (Level 3)    Year 13    3276
Female    NCEA (Level 3)    Year 13      36
```

One problem that still remains with the format in Example 3-3 is that we would still need to tell software how to separate labels and data values from each other on each row. An even better format is shown in Example 3-4. In this case, every single data value is not only clearly grouped with its corresponding labels, but the location of every label and data value is clearly delineated within the file. This sort of "self-describing" format can be read by software without any human assistance whatsoever.

Example 3-4. The data in an XML format

```
<NCEA>
  <gender="Male"   level="1" year="11" count="5929"/>
  <gender="Female" level="1" year="11" count="0"/>
  <gender="Male"   level="2" year="11" count="194"/>
  <gender="Female" level="2" year="11" count="0"/>
  <gender="Male"   level="3" year="11" count="2"/>
  <gender="Female" level="3" year="11" count="0"/>
  <gender="Male"   level="1" year="12" count="6427"/>
  <gender="Female" level="1" year="12" count="60"/>
  <gender="Male"   level="2" year="12" count="5395"/>
  <gender="Female" level="2" year="12" count="58"/>
  <gender="Male"   level="3" year="12" count="128"/>
  <gender="Female" level="3" year="12" count="0"/>
  <gender="Male"   level="1" year="13" count="5170"/>
  <gender="Female" level="1" year="13" count="38"/>
  <gender="Male"   level="2" year="13" count="5027"/>
  <gender="Female" level="2" year="13" count="38"/>
  <gender="Male"   level="3" year="13" count="3276"/>
  <gender="Female" level="3" year="13" count="36"/>
</NCEA>
```

Data Spread Across Multiple Files

In the previous section, we argued that one reason for wanting data in a reusable format is so that we can rearrange the data or display the data in a different way.

Another reason for wanting to have data in a reusable format is that data is very rarely an end point. What usually happens when we see a presentation of data like this is that it makes us think of other questions.

For example, in Figure 3-1 we have a table of the number of male and female students who have attained different levels of NCEA in different years *for boys-only schools*. Why are there *female* students attaining NCEA qualifications at boys-only schools?

It turns out that that result does have a sensible interpretation because there are schools in New Zealand that allow female students to enroll in Year 12 and Year 13 in boys-only schools. Kappa College is one example (see Figure 3-5).[2]

	A	B	C	D	E	F	G	H	I	J
1	Qualifications by Year Level and Gender									
2					Kappa College					
3					Year 11		Year 12		Year 13	
4	Qualificat	Gender								
5										
6	National Certificate of Educational Achievement									
7	NCEA (Level 1)									
8		Male			30		42		34	
9		Female			0		29		26	
10	NCEA (Level 2)									
11		Male			0		42		37	
12		Female			0		14		21	
13	NCEA (Level 3)									
14		Male			0		0		33	
15		Female			0		0		20	
16										

Figure 3-5. The number of students achieving NCEA at Kappa College in 2010. Female students are allowed to attend this school, but only at Year 12 and Year 13.

However, another question immediately springs to mind. What about *male* students at *girls-only* schools? Figure 3-6 shows a spreadsheet answering that question.

2. The real names of the New Zealand schools have been altered to avoid any privacy concerns.

	A	B	C	D	E	F	G	H	I
1	Qualifications by Year Level and Gender								
2					National				
3					Year 11		Year 12		Year 13
4	Qualificat	Gender							
5									
6	National Certificate of Educational Achievement								
7	NCEA (Level 1)								
8		Male			0		0		2
9		Female			7,432		7,665		6,872
10	NCEA (Level 2)								
11		Male			0		0		2
12		Female			70		6,958		6,751
13	NCEA (Level 3)								
14		Female			2		32		5,315
15									

Figure 3-6. The number of students achieving NCEA at girls-only schools in 2010

These data suggest that there were two male students who attained NCEA at girls-only schools—a result that has no simple explanation. And this result naturally leads to further questions, such as which girls-only schools are these male students attending?

The good news is that we can investigate this question because it is also possible to obtain a spreadsheet of results for each school in New Zealand, as was shown in Figure 3-5 for Kappa College. This means that we could answer our follow-up question by looking at the spreadsheet for each individual school in New Zealand and check for any schools that have only one or two male students at Year 13.

The bad news is that there are over 490 schools in New Zealand and the spreadsheet for each school does not tell us whether the school is coeducational or boys-only or girls-only. In order to find out which of these schools are contributing to the strange result, we would have to visually inspect over 490 separate spreadsheets!

One problem that we face now is that the dataset with which we want to work is spread across a large number of separate files. Plus we still have the problem that, within each file, the layout of the data is not very helpful.

Enough problems already! Time to look at some solutions.

The Solution: Writing Code

The short version of the solution is simply this: **write code**.

We want to check the spreadsheet for each New Zealand school to see if we can find these very few male students who are attending schools with lots of female students. Visually inspecting over 490 individual spreadsheets is not an option. Besides the fact that it would be very tedious, it is too error-prone, especially compared to the accuracy of a computer.

So we will write code to answer our question because that will be more efficient—the computer will complete the task much faster than we could if we had to inspect each spreadsheet by eye. And because writing code will also be more accurate, the computer will not make silly mistakes that we might make like accidentally confusing counts for male and female students.

Writing code also means we have a record of what we have done so that we can remember what we did and we can show others how we did the task. Furthermore, we can repeat the task very easily, including correcting our answer if we need to adjust the code because we find that it does not perform the task correctly.

In the following sections, we will write code in the R language, but many other computer languages could be used to carry out this task.

Reading Data from an Awkward Format

The first problem we want to solve is how to load data that is in an awkward format into software. Reading an entire spreadsheet into software is not difficult. The problem lies in how we identify the data values of interest within the spreadsheet. To keep things simple, we will start with just the spreadsheet for Kappa College.

The following code shows how to read an Excel spreadsheet into R using the **XLConnect** package:

```
> library(XLConnect)

> wb <- loadWorkbook("kappa.xls")
> kappa <- readWorksheet(wb, sheet=1, header=FALSE)
> kappa
```

The result is the NCEA data for Kappa College as an R "data frame," which is a set of columns of data:

```
                Col0   Col1 Col2 Col3          Col4 Col5    Col6 Col7    Col8
1  Qualificat ...
2                                    Kappa College
3                                          Year 11      Year 12      Year 13
4   Qualification Gender
5
6  National C ...
7  NCEA (Level 1)
8                    Male                    30.0         42.0         34.0
9                  Female                     0.0         29.0         26.0
10 NCEA (Level 2)
11                   Male                     0.0         42.0         37.0
12                 Female                     0.0         14.0         21.0
13 NCEA (Level 3)
14                   Male                     0.0          0.0         33.0
15                 Female                     0.0          0.0         20.0
```

What we want to do with these data is look at the data values associated with male students in Year 13. One way to identify these values is just to visually inspect the print out above to see that the numbers that we want reside in rows 8, 11, and 14 of column `Col8`. Using that knowledge, we can extract just those values from the data frame.

```
> maleRows <- c(8, 11, 14)
> year13boys <- kappa[maleRows, "Col8"]
> year13boys
[1] "34.0" "37.0" "33.0"
```

Having extracted the values of interest, we can check whether they are all less than 3. In order to do the comparison, we have to convert the character (text) values into numbers. The result of the comparison is `FALSE`, which indicates that at least one of the counts for male students in Year 13 is greater than (or equal to) 3.

```
> as.numeric(year13boys)
[1] 34 37 33
> as.numeric(year13boys) < 3
[1] FALSE FALSE FALSE
> all(as.numeric(year13boys) < 3)
[1] FALSE
```

The code above instructs the computer to carry out the task that we would otherwise have done by eye.

Reading Data Spread Across Several Files

The second problem that we have to solve is that there are more than 490 spreadsheets to check.

If all of the spreadsheets for individual schools had exactly the same format, this process could simply be repeated for all schools in New Zealand. Unfortunately, things are not that simple because some schools have different spreadsheet formats. For example, Alpha School only has results for male students (there are no rows of counts for female students) so the counts that we are interested in are in rows 8, 10, and 12 in that spreadsheet (see Figure 3-7).

	A	B	C	D	E	F	G	H	I	J
1	Qualifications by Year Level and Gender									
2					Alpha School					
3					Year 11		Year 12		Year 13	
4	Qualificat	Gender								
5										
6	National Certificate of Educational Achievement									
7	NCEA (Level 1)									
8		Male			121		199		138	
9	NCEA (Level 2)									
10		Male			0		140		136	
11	NCEA (Level 3)									
12		Male			0		1		99	
13										

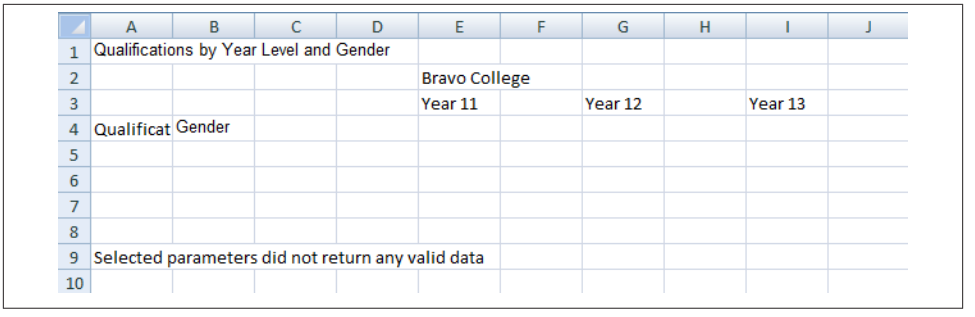
Figure 3-7. The number of students achieving NCEA at Alpha School in 2010

Another example is the Delta Education Centre, which only has *one* row of data for male students and that row has no values in it (see Figure 3-8).

	A	B	C	D	E	F	G	H	I	J
1	Qualifications by Year Level and Gender									
2					Delta Education Centre					
3					Year 11		Year 12		Year 13	
4	Qualificat	Gender								
5										
6	National Certificate of Educational Achievement									
7	NCEA (Level 1)									
8		Male								
9		Female			1		2		3	
10	NCEA (Level 2)									
11		Female			0		1		4	
12	NCEA (Level 3)									
13		Female			0		0		3	
14										

Figure 3-8. The number of students achieving NCEA at Delta Education Centre in 2010

Finally, Bravo College contains no results at all, so that spreadsheet has zero rows of interest (see Figure 3-9).



	A	B	C	D	E	F	G	H	I	J
1	Qualifications by Year Level and Gender									
2					Bravo College					
3					Year 11		Year 12		Year 13	
4	Qualificat	Gender								
5										
6										
7										
8										
9	Selected parameters did not return any valid data									
10										

Figure 3-9. The number of students achieving NCEA at Bravo College in 2010

Fortunately, we can still write code to cope with this task because another benefit of writing code is that it is *expressive*. We can express complex instructions to the computer through code. In this case, we need a smarter way to identify the rows that contain the data on male students.

One way to identify the rows with male student data is to search column `Col1` of the spreadsheet for the word "Male". The code below shows that this works for our original school, Kappa College, and also for Alpha School, Delta Education Centre, and Bravo College.

```
> grep("Male", kappa[, "Col1"])
[1]  8 11 14

> alpha <- readWorksheet(loadWorkbook("alpha.xls"), sheet=1, header=FALSE)
> delta <- readWorksheet(loadWorkbook("delta.xls"), sheet=1, header=FALSE)
> bravo <- readWorksheet(loadWorkbook("bravo.xls"), sheet=1, header=FALSE)

> grep("Male", alpha[, "Col1"])
[1]  8 10 12

> grep("Male", delta[, "Col1"])
[1] 8

> grep("Male", bravo[, "Col1"])
integer(0)
```

This code can be used to detect the location of the data values for male students across a variety of spreadsheet formats. Rather than using the fixed rows 8, 11, and 14, the code will calculate the rows we need in each case. The following code shows this working across two spreadsheets with different formats: the Kappa College data is being extracted from rows 8, 11, and 14, while the Alpha School data is extracted from rows 8, 10, and 12.

```
> kappaRows <- grep("Male", kappa[, "Col1"])
> kappa[kappaRows, "Col8"]
[1] "34.0" "37.0" "33.0"

> alphaRows <- grep("Male", alpha[, "Col1"])
> alpha[alphaRows, "Col8"]
[1] "138.0" "136.0" "99.0"
```

When there are no rows to extract, as is the case for Bravo College, we get no data.

```
> bravoRows <- grep("Male", bravo[, "Col1"])
> bravo[bravoRows, "Col8"]
character(0)
```

A better way to write this code is as a function. This will save on the amount of code that we have to write and will make it easier to update our code because there will only be one place to make changes. The following function is designed to extract whatever data exists for male students in Year 13 from a spreadsheet. The counts that we get from the spreadsheet are character values (text) so this function also converts the values to numbers.

```
> extractData <- function(ws) {
+     maleRows <- grep("Male", ws[, "Col1"])
+     as.numeric(ws[maleRows, "Col8"])
+ }
```

The code below shows this working for two spreadsheets with three counts, but in different locations (Kappa College and Alpha School) and a spreadsheet with no counts (Bravo College).

```
> extractData(kappa)
[1] 34 37 33

> extractData(alpha)
[1] 138 136  99

> extractData(bravo)
numeric(0)
```

The Delta Education Centre presents a different problem. In this case, although there is a row for males, the row is empty. When we try to extract the values from that row, we get a "missing value."

```
> extractData(delta)
[1] NA
```

We can write code to check for any missing values and remove them, as shown below.

```
> deltaCounts <- extractData(delta)
> is.na(deltaCounts)
[1] TRUE

> deltaCounts[!is.na(deltaCounts)]
numeric(0)
```

The following function is an extension of `extractData()`, which also removes any missing values.

```
> extractMale <- function(ws) {
+     maleRows <- grep("Male", ws[, "Col1"])
+     counts <- as.numeric(ws[maleRows, "Col8"])
+     counts[!is.na(counts)]
+ }
```

The following code shows the `extractMale()` function in action for all four schools that we have seen to this point.

```
> extractMale(kappa)
[1] 34 37 33

> extractMale(alpha)
[1] 138 136  99

> extractMale(bravo)
numeric(0)

> extractMale(delta)
numeric(0)
```

Once we have found some counts of male students in Year 13, we need to check whether the values are all less than three. The following function does this by using `extractMale()` to get the counts from the spreadsheet and then, if there are any counts, comparing the counts to three.

```
> allSmall <- function(ws) {
+     counts <- extractMale(ws)
+     if (length(counts)) {
+         all(counts < 3)
+     } else {
+         FALSE
+     }
+ }
```

This function works for all four of the schools that we have looked at so far. Unfortunately, the result in each case is `FALSE`, which indicates that none of these schools are the ones for which we are looking.

```
> allSmall(kappa)
[1] FALSE

> allSmall(alpha)
[1] FALSE

> allSmall(bravo)
[1] FALSE

> allSmall(delta)
[1] FALSE
```

The function above emphasizes the point that code allows us to convey complex decision processes to the computer, but we still need to take this a bit further.

So far, we have checked for schools with a small number of male students attaining NCEA in Year 13. This is probably not sufficiently strict because it will match any school that has only a small number of students attempting NCEA overall (if a school only has very few students overall, then it can only have at most very few male students who have attained NCEA by Year 13).

We can refine the search by also insisting that the school has *zero* male students attaining NCEA in Year 11 or Year 12 *and* insisting that the school must have *more than zero female* students attaining NCEA in Year 13.

To help us do this, the following code generalizes the `extractMale()` function so that we can get data values for either male or female students and so that we can get data values for any year.

```
> extractCounts <- function(ws, gender, year) {
+     col <- switch(year, "11"="Col4", "12"="Col6", "13"="Col8")
+     countRows <- grep(gender, ws[, "Col1"])
+     if (length(countRows) > 0) {
+         counts <- ws[countRows, col]
+         as.numeric(counts[!is.na(counts)])
+     } else {
+         numeric(0)
+     }
+ }
```

The following code demonstrates this function being used to get different sets of values from the Kappa College spreadsheet.

```
> extractCounts(kappa, "Male", "11")
[1] 30  0  0
> extractCounts(kappa, "Male", "12")
[1] 42 42  0
> extractCounts(kappa, "Male", "13")
[1] 34 37 33
> extractCounts(kappa, "Female", "13")
[1] 26 21 20
```

Now we can make our overall check more complex by extracting counts for male students from Year 11, 12, and 13 and counts for female students from Year 13. We will only be interested in the school if there are zero male students in Year 11 and Year 12 *and* there are fewer than three male students who have attained each NCEA level by Year 13 *and* there are some female students in Year 13.

```
> schoolWeWant <- function(ws) {
+     result <- FALSE
+     male11Counts <- extractCounts(ws, "Male", "11")
+     male12Counts <- extractCounts(ws, "Male", "12")
+     male13Counts <- extractCounts(ws, "Male", "13")
+     if (length(male13Counts)) {
+         female13Counts <- extractCounts(ws, "Female", "13")
+         if (length(female13Counts)) {
+             result <- all(male11Counts == 0) &&
+                       all(male12Counts == 0) &&
+                       all(male13Counts < 3) &&
+                       any(female13Counts > 0)
+         }
+     }
+     result
+ }
```

This function can now be used to check *all* school spreadsheets. The following code does this and prints out the names of any schools in which we are interested.

```
> for (i in list.files(pattern="[.]xls$")) {
+     wb <- loadWorkbook(i)
+     ws <- readWorksheet(wb, sheet=1, header=FALSE)
+     if (schoolWeWant(ws))
+       cat(i, "\n")
+ }
```

```
Golf Girls' High School.xls
Romeo Girls' High School.xls
Tango College.xls
```

The above code performs the tedious task of inspecting over 490 individual spreadsheets, seeking out the data within each spreadsheet and testing it to see whether it matches a set of criteria. The result is a small set of three spreadsheets that we can inspect visually to check whether we have got an answer to our question.

The first two results, Golf Girls' High School and Romeo Girls' High School, are the schools that we are looking for (see Figure 3-10 and Figure 3-11). Each school has one male student who had attained NCEA Level 1 and NCEA Level 2 by Year 13.

	A	B	C	D	E	F	G	H	I	J
1	Qualifications by Year Level and Gender									
2					Golf High School					
3					Year 11		Year 12		Year 13	
4	Qualificat	Gender								
5										
6	National Certificate of Educational Achievement									
7	NCEA (Level 1)									
8		Male			0		0		1	
9		Female			152		142		106	
10	NCEA (Level 2)									
11		Male			0		0		1	
12		Female			7		115		99	
13	NCEA (Level 3)									
14		Female			2		1		62	
15										

Figure 3-10. The number of students achieving NCEA at Golf Girl's High School in 2010

	A	B	C	D	E	F	G	H	I	J
1	Qualifications by Year Level and Gender									
2					Romeo High School					
3					Year 11		Year 12		Year 13	
4	Qualificat	Gender								
5										
6	National Certificate of Educational Achievement									
7	NCEA (Level 1)									
8		Male			0		0		1	
9		Female			126		162		147	
10	NCEA (Level 2)									
11		Male			0		0		1	
12		Female			1		106		136	
13	NCEA (Level 3)									
14		Female			0		2		71	
15										

Figure 3-11. The number of students achieving NCEA at Romeo Girl's High School in 2010

Tango College also matches our search (see Figure 3-12), but that is just because it is a very small school overall (it is not a girls-only school).

	A	B	C	D	E	F	G	H	I	J
1	Qualifications by Year Level and Gender									
2					Tango College					
3					Year 11		Year 12		Year 13	
4	Qualificat	Gender								
5										
6	National Certificate of Educational Achievement									
7	NCEA (Level 1)									
8		Male			0		0		1	
9		Female			1		1		2	
10	NCEA (Level 2)									
11		Male								
12		Female			0		0		2	
13	NCEA (Level 3)									
14		Female								

Figure 3-12. The number of students achieving NCEA at Tango College in 2010

Postscript

The code that we have written embodies a set of assumptions about the format of the data in the spreadsheets. We assume that the word "Male" always appears in column `Col1` (if it appears at all). We assume that the Year 11 data values (if there are any) always appear in column `Col4`, Year 12 data is always in `Col6`, and Year 13 is always in `Col8`. More importantly, we assume that a school that has few male students who had attained NCEA by Year 13 and some female students who had attained NCEA by Year 13 is a girls-only school.

We have written code based on these assumptions and that code has produced a list of schools, but we still had to check that these are in fact the schools that explain the original result (male students attaining NCEA at girls-only schools). What the code did for us was to reduce the workload by performing the routine checks that would take us too long to do by eye and we would be too likely to get wrong. However, there is still a need for a human brain to inspect the three schools to make sure that the answer is correct.

Other Formats

This chapter has focused on a particular example of data that are formatted for human consumption, which has demonstrated that the arrangement of data within a spreadsheet and the breaking up of a dataset across multiple spreadsheets can present significant obstacles to reusing the data.

There are, of course, many other ways that this general issue can arise. The spreadsheet examples used in this chapter do at least have the redeeming feature that it is still possible to access the individual data values unambiguously because each cell of the spreadsheet contains only one data value or data label. The situation is much worse if the data are stored in a format that does not allow data values to be clearly identified.

This problem arises if the data is presented as tables within a PDF document. For example, Figure 3-13 shows a table of mortality data from a PDF report published by the New Zealand Ministry of Health. Data in this format is purely for human consumption.

There are software tools for extracting textual content from PDF documents, but the result is unlikely to reflect the structure of the original table. For example, Example 3-5 shows the result from using `pdftotext` to extract the text from the table of data in Figure 3-13. The data values are intact, but it would be impossible to reliably recover the correspondence between data values and data labels from this result.

Mortality 2009—Numbers and Rates

Raw Numbers	2009 Deaths		
	Total	Male	Female
Māori deaths	3029	1649	1380
Non-Māori deaths	26,175	12,966	13,209
Total deaths	29,204	14,615	14,589

Figure 3-13. A table of mortality data from a PDF report published by the New Zealand Ministry of Health.

Example 3-5. The text content extracted from the PDF report using pdftotext.

```
Mortality 2009 -- numbers and rates
Raw numbers
2009 deaths
Total
Maori deaths
Male
Female
3029
1649
1380
Non-Maori deaths
26,175
12,966
13,209
Total deaths
29,204
14,615
14,589
```

Another common situation arises when data is presented in a table within a web page. For example, Figure 3-14 shows a table of rates of imprisonment from the New Zealand Department of Corrections.[3]

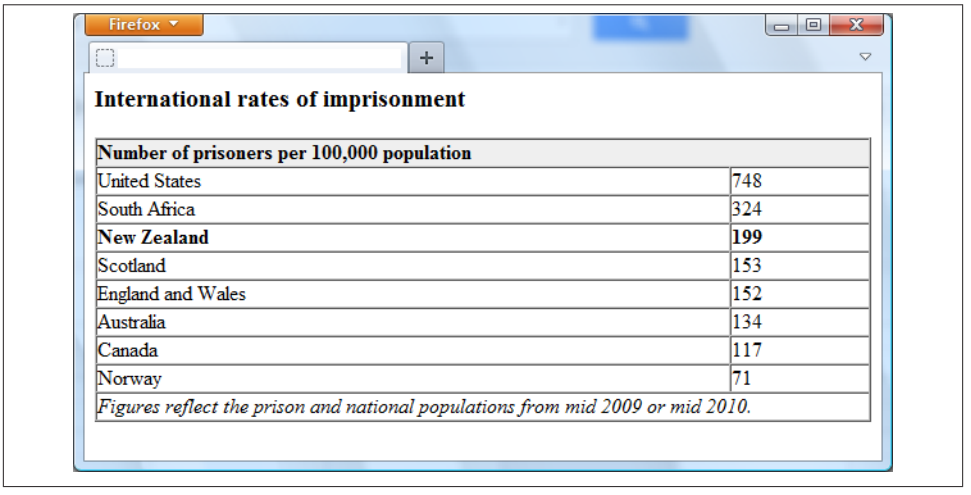


Figure 3-14. A table of international rates of imprisonment presented as a table on a web page

This sort of presentation of data lies somewhere between PDF documents and spreadsheets; there is some hope of reliably extracting the data from a web page table (because the underlying description of the table has a clear structure; see Example 3-6), though the extraction, known as "screen-scraping," is more work and less reliable than extracting cells from a spreadsheet.

3. © Crown 2003

Example 3-6. A portion of the HTML code underlying the web page table of rates of imprisonment.

```
<h3>International rates of imprisonment</h3>

<table>
  <tbody>
    <tr>
      <td colspan="2">
        <p><strong>Number of prisoners per 100,000 population</strong></p>
      </td>
    </tr>
    <tr>
      <td>United States</td>
      <td>748</td>
    </tr>
    <tr>
      <td>South Africa</td>
      <td>324</td>
    </tr>
    <tr>
      <td><strong>New Zealand</strong></td>
      <td><strong>199</strong></td>
    </tr>
```

As with the spreadsheets discussed in this chapter, further reuse of the data is also dependent on whether the layout of the table makes it easy or hard to associate data values with data labels such as column and row headers.

Summary

Data that is provided in a format designed for human consumption is awkward because it is harder to reuse. More effort is required to describe the data format to software so that software can be used to present the data in other formats or to explore the data further. On the positive side, computer code can be used to navigate around the obstacles that are created by an awkward data format.

One lesson to take away from this chapter is that it is worthwhile learning about data formats so that we can provide data to others in a way that encourages and enables better reuse of the data.

Another lesson to take away is that it is worthwhile learning about computer code so that we can work with data that is provided by others, regardless of the format that they use to store or present the data.

CHAPTER 4

Bad Data Lurking in Plain Text

Josh Levy, PhD

> *This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.*
>
> —Doug McIlroy

Bad data is often delivered with a warning or an apology such as, "This dump is a real mess, but maybe you'll find something there." Some bad data comes with a more vacuous label: "This is plain text, tab-delimited. It won't give you any trouble."

In this article, I'll present data problems I've encountered while performing seemingly simple analysis of data stored in plain text files and the strategies I've used to get past the problems and back to work. The problems I'll discuss are:

1. Unknown character encoding
2. Misrepresented character encoding
3. Application-specific characters leaking into plain text

I'll use snippets of Python code to illustrate these problems and their solutions. My demo programs will run against a stock install of Python 2.7.2 without any additional requirements. There are, however, many excellent Open Source libraries for text processing in Python. Towards the end of the article, I'll survey a few of my favorites. I'll conclude with a set of exercises that the reader can perform on publicly available data.

Which Plain Text Encoding?

McIlroy's advice above is incredibly powerful, but it must be taken with a word of caution: *some text streams are more universal than others*. A text encoding is the mapping between the characters that can occur in a plain text file and the numbers computers use to represent them. A program that joins data from multiple sources may misbehave if its inputs were written using different text encodings.

This is a problem I encountered while matching names listed in plain text files. My client had several lists of names that I received in plain text files. Some lists contained names of people with whom the client conducted business; others contained the names of known bad actors with whom businesses are forbidden from transacting. The lists were provided as-is, with little or no accompanying documentation. The project was part of an audit to determine which, if any, of the client's partners were on the bad actors lists. The matching software that I wrote was only a part of the solution. The suspected matches it identified were then sent to a team of human reviewers for further investigation.

In this setting, there were asymmetric costs for errors in the matching software. The cost of a false positive—a case where the investigative team rejects a suspected match—is the cost of the human effort to refute the match. The cost of a false negative is the risk to which the client is exposed when the investigative team is not alerted to a match. The client wanted the ability to tune the matching software's error profile, that is, to be able to choose between being exposed to fewer false positives at the expense of potentially more false negatives, or vice-versa. Had I not recognized and worked around the text encoding issues described below, the client would have underestimated its risk of false negative errors.

Let's discuss the basics of text encoding to understand why multiple text encodings are in use today. Then let's discuss how we can detect and normalize the encoding used on any given plain text file. I became computer literate in a time and place when plain text meant ASCII. ASCII is 7-bit encoding with codes for 95 printable characters shown in Table 4-1. ASCII encodes the letters of the English alphabet in upper- and lowercase, the Arabic numerals, and several punctuation and mathematical symbols.

When I write a number without a prefix, such as *65*, it can be assumed to be in decimal notation. I will use the prefix *0x* to indicate numbers in hexadecimal format, for example *0x41*.

Table 4-1. Printable ASCII characters

(32)	*!* (33)	*"* (34)	*#* (35)	*$* (36)	*%* (37)	*&* (38)	*'* (39)
((40)	*)* (41)	* (42)	*+* (43)	*,* (44)	*-* (45)	*.* (46)	*/* (47)
0 (48)	*1* (49)	*2* (50)	*3* (51)	*4* (52)	*5* (53)	*6* (54)	*7* (55)
8 (56)	*9* (57)	*:* (58)	*;* (59)	*<* (60)	*=* (61)	*>* (62)	*?* (63)
@ (64)	*A* (65)	*B* (66)	*C* (67)	*D* (68)	*E* (69)	*F* (70)	*G* (71)
H (72)	*I* (73)	*J* (74)	*K* (75)	*L* (76)	*M* (77)	*N* (78)	*O* (79)
P (80)	*Q* (81)	*R* (82)	*S* (83)	*T* (84)	*U* (85)	*V* (86)	*W* (87)
X (88)	*Y* (89)	*Z* (90)	*[* (91)	\ (92)	*]* (93)	^ (94)	_ (95)
` (96)	*a* (97)	*b* (98)	*c* (99)	*d* (100)	*e* (101)	*f* (102)	*g* (103)
h (104)	*i* (105)	*j* (106)	*k* (107)	*l* (108)	*m* (109)	*n* (110)	*o* (111)
p (112)	*q* (113)	*r* (114)	*s* (115)	*t* (116)	*u* (117)	*v* (118)	*w* (119)
x (120)	*y* (121)	*z* (122)	*{* (123)	| (124)	*}* (125)	~ (126)	(127)

ASCII omits letters and symbols such as *ñ*, *ß*, *£*, and *¡* that are used in Western European, or Western Latin, languages. Several 8-bit encoding schemes with support for Western European character sets were developed in the 1980s. These schemes were incompatible with each other. The incompatibilities between these competing standards were confounded by the historical coincidence that the Euro symbol € was invented after the 8-bit encodings had gained traction. Microsoft was able to inject € into a blank space in encoding known as Code Page 1252; IBM modified its encoding known as Code Page 850, creating Code Page 858 to add €; and the ISO-8859-15 standard was created from ISO-8859-1 to support €. Table 4-2 lists six of the most common 8-bit encodings of Western Latin character sets. Table 4-3 shows examples of the incompatibilities between these encodings, by showing how they represent some uppercase, non-English letters. For example, the character *Ÿ* is missing from IBM Code Page 858 and ISO-8859-1 and is represented by *159* in Windows Code Page 1252, by *190* in ISO-8859-15, and by *217* in MacRoman.

Table 4-2. 8-Bit Western Latin encoding schemes

Encoding	Operating System	Region	Has €
Code Page 437	DOS	USA	N
Code Page 850	DOS	Europe	N
Code Page 858	DOS	Europe	Y
Code Page 1252	Windows		Y
ISO-8859-1	ISO Standard		N
ISO-8859-15	ISO Standard		Y
MacRoman	Macintosh		Y

Table 4-3. Encodings of some non-ASCII Western Latin letters

	Code Page			ISO-8859-x		Mac Roman	Unicode
	437	858	1252	-1	-15		
À		183	192	192	192	203	192
Æ	146	146	198	198	198	174	198
Ç	128	128	199	199	199	130	199
È		212	200	200	200	233	200
Ì		222	204	204	204	237	204
Ð		209	208	208	208		208
Ñ	165	165	209	209	209	132	209
Ö	153	153	214	214	214	133	214
Ù		235	217	217	217	244	217
Þ		232	222	222	222		222
ß	225	225	223	223	223	167	223
Œ			140		188	206	338
Š			138		166		352
Ÿ			159		190	217	376

In the late 1980s, as the problems caused by these incompatible encodings were recognized, work began on Unicode, a universal character encoding. Unicode is now well supported on all major computing platforms in use today. The original version of Unicode used a 16-bit encoding for each character. That was initially believed to provide enough code points (numeric encodings of symbols) to represent all characters in all languages that had been printed in the previous year. Unicode has since been expanded to support more than one million code points, more than one hundred thousand of which have been assigned. There are Unicode code pages (groups of code points) that represent both living and historic languages from around the globe. I can keep a list of my favorite drinks: *קוקה-קולה*, *weißbier*, *清酒*, *piña colada*, and so on in a single Unicode plain text file.

Strictly speaking, a Unicode string is a sequence of code points. Unicode code points are often written with the prefix *U+* in front of a hexadecimal number. For example, *U+41* specifies code point 0x41 and corresponds to the letter *A*. A serialization scheme is needed to map between sequences of code points and sequences of bytes. The two most popular Unicode serialization schemes are UTF-16 and UTF-8. Unicode code points U+D800 - U+DFFF have been permanently reserved for the UTF-16 encoding scheme. In UTF-16, code points U+0000 - U+D7FF and U+E000 - U+FFFF are written as they are (i.e., by the same 16-bit value). The remaining assigned code points fall in the range U+010000 - U+10FFFF, and are serialized to a pair of 16-bit values: the first from the range 0xD800 - 0xDBFF and the second from the range 0xDC00 - 0xDCFF. In UTF-8,

code points are serialized to between one and four 8-bit values. The number of bytes needed increases with the code point and was designed so that the ASCII characters can be written in a single byte (the same value as in ASCII), the Western European characters can be written in two or fewer bytes, and the most commonly used characters can be written in three or fewer bytes.

The widespread adoption of Unicode, UTF-8, and UTF-16 has greatly simplified text processing. Unfortunately, some legacy programs still generate output in other formats, and that can be a source of bad data. One of the trickiest cases has to do with confusion between Windows Code Page 1252 and ISO-8859-1. As seen in Table 4-3, many characters have the same representation in Code Page 1252, ISO-8859-1, and ISO-8859-15. The differences we've seen so far are in the letters added to ISO-8859-15 that were not present in ISO-8859-1. Punctuation is a different story. Code Page 1252 specifies 18 non-alphanumeric symbols that are not found in ISO-8859-1. These symbols are listed in Table 4-4. € is found in ISO-8859-15, but at a different code point (164). The other 17 are not found in ISO-8859-15 either. Some of the symbols that are representable by Code Page 1252 but not by ISO-8859-1 are the Smart Quotes—quote marks that slant towards the quoted text. Some Windows applications replace straight quotes with Smart Quotes as the user types. Certain versions of those Windows applications had export to XML or HTML functionality that incorrectly reported the ISO-8859-1 encoding used, when Code Page 1252 was the actual encoding used.

Table 4-4. Code Page 1252 symbols (conflicts with ISO-8859-1 & ISO-8859-15)

€ (128)	‚ (130)	„ (132)	… (133)	† (134)	‡ (135)
‰ (137)	‹ (139)	‘ (145)	’ (146)	“ (147)	” (148)
• (149)	– (150)	— (151)	˜ (152)	™ (153)	› (155)

Consider the example in Example 4-1. In this Python code, *s* is a byte string that contains the Code Page 1252 encodings of the smart quotes (code points 147 and 148) and the Euro symbol (code point 128). When we erroneously treat *s* as if it had been encoded with ISO-8859-1, something unexpected happens. Code points (128, 147, and 148) are control characters (not printable characters) in ISO-8859-1. Python prints them invisibly. The printed string appears to have only 11 characters, but the Python *len* function returns 14. Code Page 1252 encoding masquerading as ISO-8859-1 is bad data.

Example 4-1. Smart quotes and ISO-8859-1

```
>>> bytes = [45,147, 128, 53, 44, 32, 112, 108, 101,
... 97, 115, 101, 148,45]
>>> s = ''.join(map(chr, bytes))
>>> print s
-??5, please?-
>>> print(s.decode('cp1252'))
-“€5, please”-
>>> print(s.decode('iso-8859-1'))
```

```
-5, please-
>>> print(len(s.decode('cp1252')))
14
>>> print(len(s.decode('iso-8859-1')))
14
```

We've now seen that text with an unknown encoding can be bad data. What's more, text with a misrepresented encoding, as can happen with the Code Page 1252 / ISO-8859-1 mix-up, can also be bad data.

Guessing Text Encoding

The Unix *file* tool determines what type of data is in a file. It understands a wide variety of file types, including some plain text character encodings. The Python script in Example 4-2 generates some text data in different encodings. The function *make_alnum_sample* iterates through the first *n* Unicode code points looking for alpha-numeric characters. The parameter *codec* specifies an encoding scheme that is used to write out the alpha-numeric characters.

Example 4-2. Generating test data

```
>>> def make_alnum_sample(out, codec, n):
    """
    Look at the first n unicode code points
    if that unicode character is alphanumeric
    and can be encoded by codec write the encoded
    character to out
    """
    for x in xrange(n):
        try:
            u = unichr(x)
            if u.isalnum():
                bytes = u.encode(codec)
                out.write(bytes)
        except:
            # skip u if codec cannot represent it
            pass
    out.write('\n')

>>> codecs = ['ascii', 'cp437', 'cp858', 'cp1252',
... 'iso-8859-1', 'macroman', 'utf-8', 'utf-16']
>>> for codec in codecs:
   with open('../%s_alnum.txt' % codec, 'w') as out:
      make_alnum_sample(out, codec, 512)
```

The results of running the files generated in Example 4-2 through *file* are shown in Example 4-3. On my system, *file* correctly identified the ASCII, ISO-8859, UTF-8, and

UTF-16 files. *file* was not able to infer the type of the files containing bytes that map to alphanumeric characters in Code Page 1252, Code Page 437, Code Page 858, or MacRoman. From the output of *file*, we know that those files contain some 8-bit text encoding, but we do not yet know which one.

Example 4-3. Output from file command

```
$ file *alnum.txt
ascii_alnum.txt:      ASCII text
cp1252_alnum.txt:     Non-ISO extended-ASCII text
cp437_alnum.txt:      Non-ISO extended-ASCII text,
↪ with LF, NEL line terminators
cp858_alnum.txt:      Non-ISO extended-ASCII text,
↪ with LF, NEL line terminators
iso-8859-1_alnum.txt: ISO-8859 text
macroman_alnum.txt:   Non-ISO extended-ASCII text,
↪ with LF, NEL line terminators
utf-16_alnum.txt:     Little-endian UTF-16 Unicode
↪ text, with very long lines, with no line
↪ terminators
utf-8_alnum.txt:      UTF-8 Unicode text,
↪ with very long lines
```

One way to resolve this problem is to inspect snippets of text containing non-ASCII characters, manually evaluating how they would be generated by different encoding schemes. Python code that does this is shown in Example 4-4. Let's revisit the byte string from Example 4-1. Pretend for a moment that we don't know the contents of that string. Running it through *test_codecs* and *stream_non_ascii_snippets*, we see that sequence of bytes is valid in Code Page 858, Code Page 1252, and MacRoman. A human looking at the results of *test_codecs* can make a judgment as to which encoding makes sense. In this case, smart quotes and € contextually make more sense than the other options, and we can infer that the text is encoded by Code Page 1252.

Example 4-4. Snippets of non-ASCII text

```
>>> def stream_non_ascii_snippets(s, n_before=15,
... n_after=15):
    """
    s is a byte string possibly containing non-ascii
    characters
    n_before and n_after specify a window size

    this function is a generator for snippets
    containing the n_before bytes before a non-ascii
    character, the non-ascii byte itself, and the
    n_after bytes that follow it.
    """
    for idx, c in enumerate(s):
        if ord(c) > 127:
            start = max(idx - n_before, 0)
```

```
            end = idx + n_after + 1
            yield(s[start:end])
>>> CODECS = ['cp858', 'cp1252', 'macroman']
>>> def test_codecs(s, codecs=CODECS):
    """
    prints the codecs that can decode s to a Unicode
    string and those unicode strings
    """
    max_len = max(map(len, codecs))
    for codec in codecs:
        try:
            u = s.decode(codec)
            print(codec.rjust(max_len) + ': ' + u)
        except:
            pass
>>> bytes = [45,147, 128, 53, 44, 32, 112, 108, 101,
... 97, 115, 101, 148,45]
>>> s = ''.join(map(chr, bytes))
>>> test_codecs(next(stream_non_ascii_snippets(s)))
   cp858: -ôÇ5, pleaseö-
  cp1252: -“€5, please”-
macroman: -ìÄ5, pleaseî-
```

The *stream_non_ascii_snippets* function in Example 4-4 lets the user explore the non-ASCII bytes sequentially, in the order in which they occur in the byte string. An alternative, presented in Example 4-5, is to consider the frequency of occurrence of non-ASCII bytes in the string. The set of unique non-ASCII bytes might be enough to eliminate some encodings from consideration, and the user may benefit from inspecting snippets containing specific characters. The test string with which we've been working isn't the most interesting, because it is short and no non-ASCII character repeats. However, Example 4-5 shows how these ideas could be implemented.

Example 4-5. Frequency-count snippets of non-ASCII text

```
>>> from collections import defaultdict
>>> from operator import itemgetter
>>> def get_non_ascii_byte_counts(s):
    """
    returns {code point: count}
    for non-ASCII code points
    """
    counts = defaultdict(int)
    for c in s:
        if ord(c) > 127:
            counts[ord(c)] += 1
    return counts
>>> def stream_targeted_non_ascii_snippets(s,
... target_byte, n_before=15, n_after=15):
    """
    s is a byte string possibly containing non-ascii
    characters
```

```
    target_byte is code point
    n_before and n_after specify a window size

    this function is a generator for snippets
    containing the n_before bytes before
    target_byte, target_byte itself, and the n_after
    bytes that follow it.
    """
    for idx, c in enumerate(s):
        if ord(c) == target_byte:
            start = max(idx - n_before, 0)
            end = idx + n_after + 1
            yield(s[start:end])
>>> sorted(get_non_ascii_byte_counts(s).items(),
... key=itemgetter(1,0), reverse=True)
[(148, 1), (147, 1), (128, 1)]
>>> it = stream_targeted_non_ascii_snippets(s, 148,
... n_before=6)
>>> test_codecs(next(it))
   cp858: pleaseö-
  cp1252: please”-
macroman: pleaseî-
```

Normalizing Text

When mashing up data from multiple sources, it is useful to normalize them to either UTF-8 or UTF-16 depending on which is better supported by the tools you use. I typically normalize to UTF-8.

Let us revisit the file *macroman_alnum.txt* that was generated in Example 4-2. We know from its construction that it contains 8-bit MacRoman encodings of various alphanumeric characters. Example 4-6 shows standard Unix tools operating on this file. The first example, using *cat*, shows that the non-ASCII characters do not render correctly on my system. *iconv* is a Unix tool that converts between character sets. It converts *from* the encoding specified with the `-f` parameter *to* the encoding specified with `-t`. Output from *iconv* gets written to *STDOUT*. In Example 4-6, we allow it to print, and we see that the non-ASCII characters display correctly. In practice, we could redirect the output from *iconv* to generate a new file with the desired encoding.

Example 4-6. Normalizing text with Unix tools

```
$ cat macroman_alnum.txt
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnop
↪ qrstuvwxyz??????????????????????????????????????
↪ ????????????????
$ iconv -f macroman -t utf-8 macroman_alnum.txt
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnop
↪ qrstuvwxyzªµºÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÑÒÓÔÕÖØÙÚÛÜßàáâãäåæçè
↪ éêëìíîïñòóôõöøùúûüÿıŒœŸƒ
```

Using *iconv* to re-encode text can be consistent with McIlroy's Unix philosophy. One might write a tool to perform some specific text processing operation to transform one UTF-8 document into another. We can design this tool to "do one thing and do it well." Its code can be simpler than a program that has to detect and normalize its input because it can always assume it is working with UTF-8. If it needs to process an non-UTF-8 dataset, that input can be piped through *iconv*. Similarly, its output needs to be consumed by a process with different encoding assumptions; it too can be piped through *iconv*.

Python has excellent support for decoding byte strings to Unicode. I often find it useful to write functions that "do one thing and do it well" operating on Unicode input, and then to wrap them with a bit of boilerplate to decode text from other formats. I tend to use this technique when I'm developing systems that allow their user to reference input that they do not control (for example, by passing in a URL). Two functions that decode byte strings to Unicode are *str.decode* and *codecs.open*. Both are illustrated in Example 4-7.

Example 4-7. Normalizing text from Python

```
>>> with open('macroman_alnum.txt') as f:
    print(f.readline())
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnop
↪ qrstuvwxyz?????? ????????????????□?????????????
↪ ??????????????????
>>> with open('macroman_alnum.txt') as f:
    print(f.readline().decode('macroman'))
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnop
↪ qrstuvwxyzªµºÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÑÒÓÔÕÖØÙÚÛÜßàáâãäåæçè
↪ éêëìíîïñòóôõöøùúûüÿıŒœŸƒ
>>> import codecs
>>> with codecs.open('macroman_alnum.txt',
... encoding='macroman') as f:
    print(f.readline())
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnop
↪ qrstuvwxyzªµºÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÑÒÓÔÕÖØÙÚÛÜßàáâãäåæçè
↪ éêëìíîïñòóôõöøùúûüÿıŒœŸƒ
```

The first example of Example 4-7 simply shows that the byte string for what we know to be MacRoman-encoded text does not render correctly within Python. In the second example, we use *str.decode* to produce the correct Unicode string, which in turn prints correctly. The last example of Example 4-7 uses *codecs.open* when reading the file. *codecs.open* is called in the same manner as the ordinary *open* function, and returns an object that behaves like an ordinary *file* except that it automatically decodes byte strings to Unicode when reading, and it automatically encodes Unicode to the specified encoding when writing. In the last example of Example 4-7, the call to *f.readline* returns a properly decoded Unicode string. The call to *print* handles the Unicode string correctly.

Thus far, we have discussed how text with an unknown or misrepresented character encoding can be bad data for a text processing application. A join operation using text

keys may not behave as expected if different encodings are used on its inputs. We have seen strategies for detecting character encoding using the Unix `file` command or custom Python code. We have also seen how text with a known representation can be normalized to a standard encoding such as UTF-8 or UTF-16 from Python code or by using *iconv* on the command line. Next, we will discuss another bad data problem: application-specific characters leaking into plain text.

Problem: Application-Specific Characters Leaking into Plain Text

Some applications have characters or sequences of characters with application-specific meanings. One source of bad text data I have encountered is when these sequences leak into places where they don't belong. This can arise anytime the data flows through a tool with a restricted vocabulary.

One project where I had to clean up this type of bad data involved text mining on web content. The users of one system would submit content through a web form to a server where it was stored in a database and in a text index before being embedded in other HTML files for display to additional users. The analysis I performed looked at dumps from various tools that sat on top of the database and/or the final HTML files. That analysis would have been corrupted had I not detected and normalized these application-specific encodings:

- URL encoding
- HTML encoding
- Database escaping

In most cases, the user's browser will URL encode the content before submitting it. In general, any byte can be URL encoded by % followed by its hex code. For example, *ȳ* is Unicode code point U+233. Its UTF-8 representation is two bytes: 0xC3A9, and the URL encoding of its UTF-8 representation is *%C3%A9*. URL encoding typically gets applied to non-ASCII characters if present, as well as to the following ASCII symbols, each of which has a special meaning when it appears in a URL: *;/?:@&=+$,*. Because *%* is used in the encoded string, raw *%* characters in the input need to be encoded. The ASCII code for *%* is *0x25*, so the encoding *%25* is used. The blank space is ASCII code 0x20, but it is typically encoded as + rather than *%20*. The server application must decode the URL encoding to recover exactly what the user has entered.

In general, URL encoding artifacts leaking into plain text is not a serious problem. URL encoding and decoding of form submissions happens behind the scenes in most web application frameworks. Even if URL encoded text did leak into other parts of an application, it would be easily detectable by the lack of spaces and abundance of + and

%XX codes. One notable exception is when analyzing a list of URLs or URL fragments. In that case, it may be worthwhile to ensure that all of the URLs have been decoded consistently. Example 4-8 uses Python's *urllib.urlencode* function to URL encode text, and then *urllib.unquote* functions to decode the URL encoded text.

Example 4-8. Decoding URL encoded text

```
>>> import urllib
# urlencode generates a query string from a dict
>>> urllib.urlencode({'eqn': '1+2==3'})
'eqn=1%2B2%3D%3D3'
# unquote decodes a URL encoded string
>>> s = 'www.example.com/test?eqn=1%2B2%3D%3D3'
>>> urllib.unquote(s)
'www.example.com/test?eqn=1+2==3'
```

Let's return to our application collecting user-generated content for inclusion on other web pages. Our users enter their content into a form, their browser URL encodes it at submission time, our application decodes it, and we have their content ready to display on one of our pages. If we write the submitted text as-is on our web pages, evil-doers will submit content containing their own HTML markup for things like link spamming, Rickrolling[1], or executing arbitrary JavaScript. Best practice for this type of application is to HTML encode user submissions so they get rendered as they were typed. HTML encoding a string replaces some characters with an entity reference: a sequence beginning with *&* followed by the name or code point of a character, followed by *;*. Example 4-9 gives an example of HTML encoding text.

Example 4-9. Decoding HTML encoded text

```
>>> import cgi
>>> import HTMLParser
>>> s = u'<script>//Do Some Évîl</script>'
>>> encoded = cgi.escape(s).encode('ascii',
... 'xmlcharrefreplace')
>>> print(encoded)
&lt;script&gt;//Do Some &#201;v&#238;l&lt;/script
↪ &gt;
>>> print(HTMLParser.HTMLParser().unescape(encoded))
<script>//Do Some Évîl</script>
```

The call to `cgi.escape` in Example 4-9 replaces the angle brackets < and > with the named entities `<` and `>` respectively. `unicode.encode(…, xmlcharrefre`

1. To "Rickroll" someone is to trick them into clicking a link that plays the music video for Rick Astley's song "Never Gonna Give You Up"

`place)` replaces the non-ASCII characters `É (U+C9)` and `î (U+EE)` with their numeric entities: `&201;` and `&238;` (0xC9 = 201, 0xEE=238). When a browser encounters the encoded string `<script>//Do Some &#201;v&#238;l</script>` it will display *<script>//Do Some Évîl</script>*, but it will not actually execute the evil script.

It is a reasonable decision to have a web application store HTML encoded strings in its database. That decision ensures that raw text submitted by the users won't appear in our other pages, and it may speed up the server-side rendering time for those pages. However, if we decide to text mine the user-submitted content, we'll need to understand how the content is formatted in database dumps, and we'll want to decode the HTML entity references before processing it.

I've actually seen redundantly HTML-encoded strings such as `&amp;lt;` in what was supposed to be a plain text dump. That data presumably passed through multiple web applications and databases before I got my hands on it. Example 4-10 expands on code from Example 4-9 to decode repeatedly HTML-encoded strings inside a `while` loop.

Example 4-10. Decoding redundantly HTML encoded text

```
>>> # add a few more layers of encoding
>>> ss = cgi.escape(encoded).encode('ascii',
... 'xmlcharrefreplace')
>>> ss = cgi.escape(ss).encode('ascii',
... 'xmlcharrefreplace')
>>> print(ss)
&amp;amp;lt;script&amp;amp;gt;//Do Some &amp;amp;
↪ #201;v&amp;amp;#238;l&amp;amp;lt;/script
↪ &amp;amp;gt;
>>> # now decode until length becomes constant
>>> while len(ss) != len(parser.unescape(ss)):
    ss = parser.unescape(ss)
    print(ss)
&amp;lt;script&amp;gt;//Do Some &amp;#201;v&amp;
↪ #238;l&amp;lt;/script&amp;gt;
&lt;script&gt;//Do Some &#201;v&#238;l&lt;/script
↪ &gt;
<script>//Do Some Évîl</script>
```

HTML encoding all user-submitted text is a step towards preventing malicious users from launching attacks when our pages are rendered. A well-engineered web application will also take precautions to protect itself from attacks that exploit its form submission handlers. A common example of such an attack is the SQL injection attack, where the attacker tries to trick a form handler into running user-supplied SQL statements. There is a brilliant example of a SQL injection attack in the famous XKCD comic about "Little Bobby Tables" (*http://xkcd.com/327/*).

The characters ', ;, --, and /* are often exploited in SQL injection attacks. They are used to terminate strings (') and statements (;), and to begin comments that span single (--)

or multiple lines (/*). There are two main strategies for defending against SQL injection attacks. The first uses a database feature called "prepared statements" that separates a SQL statement from its parameters, eliminating the possibility that a maliciously crafted parameter could terminate the original statement and launch an attack. When prepared statements are used, the special characters listed above can occur as-is in the database and dump files exported from the database. The second strategy is to detect and escape those special strings. When this technique is used, a text processing application operating on a dump from the database will need to decode the escaped strings back to their normal forms.

As we've seen, URL encoding, HTML encoding, and SQL escaping can all leak into text generated by a web application. Another case where encoding/decoding rules need to be implemented in text processing applications comes up when data is exported from a spreadsheet or database to a flat file such as CSV. Many tools will let the user specify which characters are used for field delimiters and which are used for quoting. Quoting is necessary if the export field delimiter happens to appear in the original data. In Example 4-11, we see a simple example simulating a dump of Name and Job Description from a database into a .CSV file

Example 4-11. Quoted CSV

```
>>> import StringIO
>>> import csv
>>> # s behaves like a file opened for reading
>>>  s = StringIO.StringIO('''Name,Job Description
"Bolton, Michael ""Mike""","Programmer"
Bolton,Michael "Mike",Programmer''')
>>> # When we count the fields per line,
>>> # str.split is confused by Name
>>> map(len, [line.split(',') for line in s])
[2, 3, 3]
>>> # csv.reader understands quoted name
>>> s.seek(0)
>>> map(len, csv.reader(s))
[2, 2, 3
>>> s.seek(0)
>>> data = [row for row in csv.reader(s)]
>>> # with quotes the comma in the name
>>> # is not a delimiter
>>> data[1][0]
'Bolton, Michael "Mike"'
>>> # without quotes all commas are delimiters
>>> data[2][0]
'Bolton'
```

The difference between the data rows of *s* in Example 4-11 is that values are quoted in the first, similar to what MySQL's `mysqldump --fields-enclosed-by="` would produce. Values in the second data row are not quoted. The Python functions `str.split` and

`unicode.split` are simple ways to extract fields from a line of comma-delimited text. They treat all commas as delimiters, a behavior that is incorrect for this data, where the name field contains a non-delimiting comma. Python's `csv.reader` allows the non-delimiting commas to occur within quoted strings, so it correctly parses the data line where the values are quoted. Mismatched columns when parsing delimited text is a bad data problem. I recommend quoting when exporting text from a database or spreadsheet, and using `csv.reader` rather than `str.split` as a parser.

If we don't understand the processes that create plain text files, application-specific characters may inadvertently leak in and affect text analysis. By understanding these processes and normalizing text before working with it, we can avoid this type of bad data.

Text Processing with Python

We've discussed bad data problems caused by unknown or misrepresented text encodings. We've also discussed bad data problems caused by application-specific encodings or escape characters leaking into plain text dumps. We've used small examples written in Python to expose these problems and their solutions. Table 4-5 summarizes the Python functions we've used in these demonstrations.

Table 4-5. Python reference

Function	Notes	Listings
str.decode	Converts byte string to Unicode string	Example 4-1
		Example 4-4
		Example 4-7
unicode.encode	Converts Unicode string to byte string	Example 4-2
unichr	Maps number (Unicode code point) to character	Example 4-2
ord	Gets number (code point) from byte string or Unicode strings	Example 4-5
codecs.open	Reads and decodes Unicode strings from a file of byte strings	Example 4-7
urllib.urlencode	URL encodes a dictionary	Example 4-8
urllib.unquote	Decodes URL-encoded string	Example 4-8
cgi.escape	HTML encodes characters. Used with unicode.encode('ascii', 'xmlcharrefreplace')	Example 4-9
		Example 4-10
HTMLParser.unescape	Decodes HTML-encoded string	Example 4-9
		Example 4-10
csv.reader	Parses delimited text	Example 4-11

The functions listed in Table 4-5 are good low-level building blocks for creating text processing and text mining applications. There are a lot of excellent Open Source Python libraries for higher level text analysis. A few of my favorites are listed in Table 4-6.

Table 4-6. Third-party Python reference

Library	Notes
NLTK	Parsers, tokenizers, stemmers, classifiers
BeautifulSoup	HTML & XML parsers, tolerant of bad inputs
gensim	Topic modeling
jellyfish	Approximate and phoenetic string matching

These tools provide a great starting point for many text processing, text mining, and text analysis applications.

Exercises

1. The results shown in Example 4-3 were generated when the *n* parameter to *make_alnum_sample* was set to *512*. Do the results change for other values of *n*? Try with *n=128*, *n=256*, and *n=1024*.
2. Example 4-4 shows possible interpretations of the string *s* for three character encodings: Code Page 858, Code Page 1252, and MacRoman. Is every byte string valid for all three of those encodings? What happens if you run *test_codecs(''.join(map(chr, range(256))))*.
3. Example 4-6 shows *iconv* converting text from MacRoman to UTF-8. What happens if you try to convert the same text from MacRoman to ASCII? What if you try to convert the same text from ISO-8859-1 to UTF-8?

The Office of the Superintendent of Financial Institutions, Canada publishes a list of individuals connected to terrorism financing. As of July 9, 2012, the list can be downloaded from *http://bit.ly/S1l9WK*. The following exercises refer to that list as "the OSFI list."

4. How is the OSFI list encoded?
5. What are the most popular non-ASCII symbols (not necessarily letters) in the OSFI list?
6. What are the most popular non-ASCII letters in the OSFI list?
7. Is there any evidence of URL encoding in the OSFI list?
8. Is there any evidence of HTML encoding in the OSFI list?

CHAPTER 5

(Re)Organizing the Web's Data

Adam Laiacano

The first, and sometimes hardest part of doing any data analysis is acquiring the data from which you hope to extract information. Whether you want to look at your personal spending habits, calculate your next trade in fantasy baseball, or compare a politician's investment returns to your own, the data you need is usually there on the web with some sense of order to it, but it's probably not in a form that's very useful for analysis. If this is the case, you'll need to either manually gather the data or write a script to collect the data for you.

The granddaddy of all data formats is the data table, with a column for each attribute and a row for each observation. You've seen this if you've ever used Microsoft Excel, relational databases, or R's `data.frame` object.

Table 5-1. An example data table

Date	Blog	Posts
2012-01-01	adamlaiacano	2
2012-01-01	david	4
2012-01-01	dallas	6
2012-01-02	adamlaiacano	0
2012-01-02	david	4
2012-01-02	dallas	6

Most websites store their data behind the scenes in tables within relational databases, and if those tables were accessible to the computing public, this chapter of *Bad Data Handbook* wouldn't need to exist. However, it's a web designer's job to make this information visually appealing and interpretable, which usually means they'll only present the reader with a relevant subset of the dataset, such as a single company's stock price over a specific date range, or recent status updates from a single user's social connections.

Even online "database" websites are vastly different from a programmer's version of a database. On web database websites such as LexisNexis or Yahoo! Finance, there are page dividers, text formatting, and other pieces of code to make the page look prettier and easier to interpret. To a programmer, a database is a complete collection of clean, organized data that is easily extracted and transformed into the form you see on the web. So now it's your job to reverse-engineer the many different pages and put the data back to a form that resembles the original database form.

The process of gathering the data you want consists of two main steps. First is a web crawler that can find the page with the appropriate data, which sometimes requires submitting form information and following specific links on the web page. The second part is to "scrape" the data off of the page and onto your hard drive. Sometimes these steps are both straightforward, often one is much trickier than the other, and then there is the all too common case where both steps are tricky.

In this chapter, I'll cover how and when you can gather data from the web, give an example of a simple scraper script, and provide some pitfalls to watch out for.

Can You Get That?

Writing the code to crawl a website and gather data should be your last resort. It's a very expensive undertaking, both in terms of developer salary and the opportunity cost of what you could be doing with the data once it's acquired. Before writing any web scraping code, I would consider some of the following actions.

First off, see if somebody else already did this work and made the data available. There are websites such as the Infochimps Data Marketplace and ScraperWiki that contain many useful datasets either for free or for a fee. There are also several government websites that put their data in common, easy to use formats with a searchable interface. Some good examples are *www.data.gov.uk* and *nycopendata.socrata.com*.

Also see if the data you want is available through an API (Application Programming Interface). You'll still have to extract the data programmatically, but you can request specific pieces of information, such as all of a certain user's Twitter followers or all of the posts on a certain Tumblr blog. Most major social networks offer an API and deliver the information in a standard format (JSON or XML) so there's no need to try to scrape data off of the rendered web pages.

In some cases, you can try contacting the website and tell them what you want. Maybe they'll sell you the data, or just send you a database dump if they're really nice. My colleagues and I were once able to acquire several months worth of Taxi and Limousine Commission (TLC) data for New York City via the Freedom of Information Law (FOIL). The folks at the TLC were very nice and mailed us a DVD with three months of pick up, drop off, fare, and other information about every taxi ride in the five boroughs.

General Workflow Example

Database architects spend a lot of time optimizing the ETL (Extract, Transform, Load) flow to take data out of a database and put it on a webpage as efficiently as possible. This involves making sure that you index the right columns, keep relevant data in the same table so that you don't have to do expensive `JOIN` operations, and many other factors.

To scrape the data off a webpage, there is a similar workflow. I'll work through an example of scraping No Child Left Behind teacher quality data from the Idaho state website. No Child Left Behind[1] requires an annual assessment of all schools that receive federal funding, and also requires the results to be made available to the public. You can easily find these results on each state's website. The URL for Idaho is *http://www.sde.idaho.gov/ReportCard*. The site is very useful if you want to see how your specific school district is performing, but as data scientists, we need to see how all of the schools are performing so that we can search for trends and outliers.

In this chapter, we'll be gathering the Teacher Quality for each of the school districts in Idaho for the year 2009. An example page that we'll be scraping is *http://bit.ly/XdPENa*, pictured below. This example is written entirely in Python, though there are tools in pretty much all major scripting languages for screen scraping such as this. Just because we are used to viewing this information in a web browser doesn't mean that's the only way to access it. The further you can stay from an actual web browser when writing a web scraper, the better. You'll have more tools available to you, and you can likely perform the whole process of finding, gathering, and cleaning data in fewer steps.

Extracting the data on all of these schools is a multistep process.

- Find a pattern in the URLs for the different school districts
- Store a copy of the web page's HTML source code locally whenever possible
- Parse web pages into clean datasets
- Append new data to an existing dataset

The entire data extraction and cleaning process should be as close to fully automated as possible. This way you can re-acquire the entire dataset should it become lost or corrupted, and you can re-run the script again with minimal changes if more data becomes available.

I also can't stress enough that you should separate these steps as much as possible within your code, especially if you're executing your script daily/weekly/monthly to append new data. I once had a web scraping script running for several days when it suddenly started throwing errors. The problem was that the website had been completely redesigned. Fortunately, I only had to rewrite the one function that parses the HTML

1. *http://en.wikipedia.org/wiki/No_Child_Left_Behind_Act*

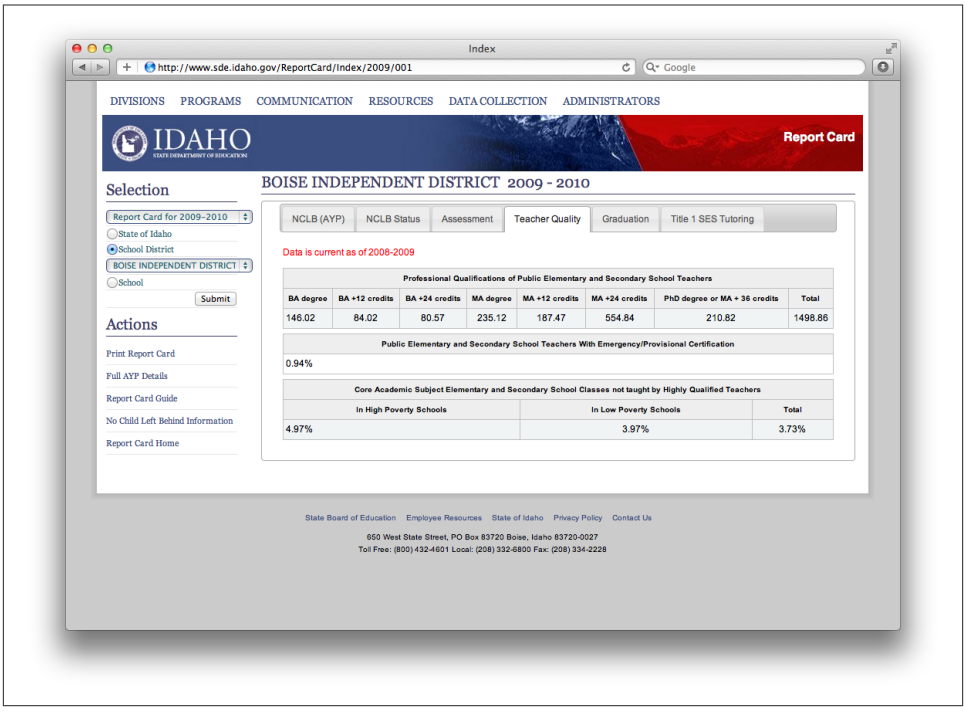


Figure 5-1. Example table to be gathered. There is an identical table for each school district in Idaho.

tables. As we make our way through the No Child Left Behind example, keep in mind that we're only gathering data for 2009, and what would be required to modify the program to collect data on a different year. Ideally, it would only be a line or two of code to make this change.

robots.txt

Nearly every website has a *robots.txt* file that tells web crawlers (that's you) which directories it can crawl, how often it can visit them, and so on. The rules outlined in *robots.txt* are not legally binding, but have been respected since the mid-1990s. You should also check the website's terms of service, which are certainly legally binding.

Here is an example *robots.txt* file from App Annie, which is a website that gathers and reports the reviews for your apps in the iOS App Store.

```
User-agent: *
Crawl-delay: 5
Sitemap: http://www.appannie.com/media/sitemaps/sitemapindex.xml
Disallow: /top/*date=
```

```
Disallow: /matrix/*date=
Disallow: /search/*
Disallow: /search_ac/
Disallow: */ranking_table/
Disallow: /top-table/*
Disallow: /matrix-table/*
Disallow: */ranking/history/
Disallow: */ranking/history/chart_data/
```

Let's break these down one section at a time.

`User-agent: *`
: The following rules apply to everybody. Sometimes a site will allow or block specific web crawlers such as Google's indexer.

`Crawl-delay: 5`
: Crawlers need to wait five seconds between each page load.

`Sitemap: http://www.appannie.com/media/sitemaps/sitemapindex.xml`
: Defines which pages a web crawler should visit and how often they should be indexed. It's a good way to help search engines index your site.

`Disallow:`
: Specifies which pages the bots are not allowed to visit. The `*` acts as the familiar wildcard character.

Robots.txt should be respected even for public data, such as in our example. Even though the data is public, the servers hosting and serving the traffic may be privately owned. Fortunately, there isn't actually a file at *http://www.sde.idaho.gov/robots.txt*, so we're all set to go.

For more specifics on how to interpret the rules of a *robots.txt* file, check out its Wikipedia page.

Identifying the Data Organization Pattern

We first need to figure out how to navigate to the page that contains the data. In our example, we want to loop over school districts. If we visit *http://www.sde.idaho.gov/reportcard*, we'll see a `select` object with each of the school districts.

Clicking on each of these options will bring us to a page with the pattern *http://www.sde.idaho.gov/ReportCard/Index/2009/<id number>*. So all we have to do is obtain the list of districts and their ID numbers and we can begin looping through the pages to extract their source HTML code. If we view the HTML for *http://www.sde.idaho.gov/reportcard* and find the `select` element with the school districts listed, we'll see that it will provide us with all of the information we need:

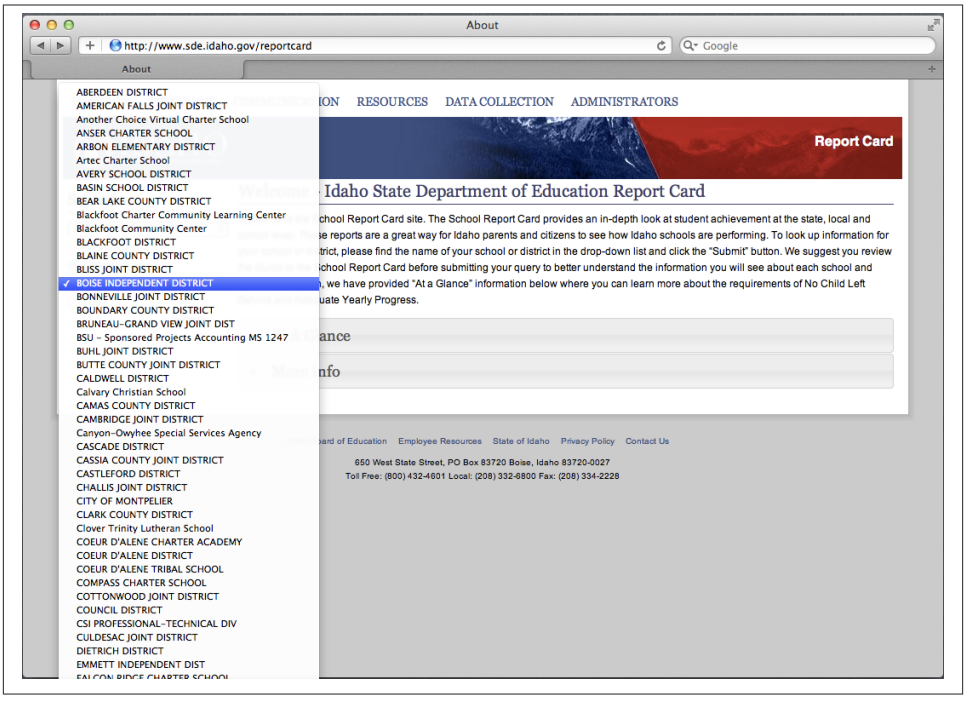


Figure 5-2. All of the school districts appear in one form element.

```
<select id="LEA" name="LEA">
    <option value="058">ABERDEEN DISTRICT</option>
    <option value="381">AMERICAN FALLS JOINT DISTRICT</option>
    <option value="476">Another Choice Virtual Charter School</option>
    <option value="492">ANSER CHARTER SCHOOL</option>
    <option value="383">ARBON ELEMENTARY DISTRICT</option>
    <option value="796">Artec Charter School</option>
    <option value="394">AVERY SCHOOL DISTRICT</option>
    ...more districts...
</select>
```

We'll be interested in the `value` attribute, which will appear in the URL pattern for the individual district pages, and of course we'll need the district name itself. We can extract a list of these values with the following function. We use the `BeautifulSoup` package[2] for Python, which makes it simple to parse and navigate HTML files.

```
def get_district_pages(index_url):
    """
    Takes a URL string and returns a list of tuples with page IDs and district
```

2. *http://www.crummy.com/software/BeautifulSoup/*

```
    names from the 'LEA' drop-down menu.
    """
    index_page = urllib2.urlopen(index_url).read()
    soup = BeautifulSoup(index_page)

    drop_down = soup.find(attrs={'id':'LEA'}) # ❶

    school_districts = []
    for district in drop_down.findAll('option'):  # ❷
        if district.text.strip() != "" and district['value'].strip() != "":
            school_districts.append(
                (str((district['value'])), str(district.text.lower()))
            )

    return school_districts

districts = get_district_pages('http://www.sde.idaho.gov/reportcard')
```

❶ This finds HTML elements of any type that have the `id="LEA"` attribute. In this case, it is the `<select>` object containing our list.

❷ Once we have extracted the `LEA` object, we loop through all `<option>` sub-elements and extract their values.

The result will be a list of tuples, with the district ID and name. The first few elements will look like this:

```
[('381', 'american falls joint district'),
 ('476', 'another choice virtual charter school'),
 ('492', 'anser charter school'),
 ('383', 'arbon elementary district'),
 ('796', 'artec charter school'),
 ...
 ]
```

Now we can create a function to convert this `(id, name)` tuple to a URL. This is a simple one-line function, but it's important to use a function here in case the URL pattern changes when the site is updated.

```
def build_url(district_tuple):
    return 'http://www.sde.idaho.gov/ReportCard/Index/2009/%s' % district_tuple[0]
```

Store Offline Version for Parsing

Now that we have the URLs generated, we can loop through and save an offline version of each. I highly recommend doing this because most websites don't like having their data scraped (whether it's public information or not) and some detect bots like this. As you debug the function to parse the actual page, you will probably have to run the script several times, and it's much faster and easier to load a local version of the file than to

download it from the Web again. Additionally, if you're gathering data from a page that changes daily (such as the front page of a media website), you will probably want to keep a history of the pages you've scraped in case you find a bug or decide you want more information at a later date.

Here's a function to load a web page and save the source in flat files with a simple `id_name.html` naming convention.

```
def cache_page(district, cache_dir):
    """
    Takes the given district tuple and saves a copy of the source code locally.

    This way we don't have to wait for pages to load and don't bother the website
    with requests.
    """
    url = build_url(district)

    # Create the cache directory if it doesn't already exist
    if not cache_dir in os.listdir('.'):
        os.mkdir(cache_dir)

    source = urllib2.urlopen(url).read()

    dest_file = os.path.join(cache_dir, "%s_%s.html" % district)
    open(dest_file, 'wb').write(source)
```

Scrape the Information Off the Page

The last step is of course to parse the page contents and get the information we want. It turns out that the Teacher Quality table that we'll scrape in is contained in a `div` tag with three total tables. The structure looks like this:

```
<div id="TeacherQuality">
    <p><font color="red">Data is current as of 2008-2009</font></p>
    <table>
        <thead>
            <tr>
                <th colspan="8">
                    Professional Qualifications of Public Elementary and Secondary
                    School Teachers
                </th>
            </tr>
            <tr>
                <th>BA degree</th>
                <th>BA +12 credits</th>
                ...
            </tr>
        </thead>
        <tbody>
            <tr>
                <td>58.45</td>
```

```
                <td>30.94</td>
                ...
            </tr>
        </tbody>
    </table>
    <table>
        ...
    </table>
    <table>
        ...
    </table>
</div>
```

Fortunately, `BeautifulSoup` will let us jump right to that `div` tag and then iterate through the `table` elements within it. I'm also taking a couple of other precautions here: first, I'm checking the local cache directory for a local copy of the source code before loading the page from the web. Second, I'm logging any errors to a log file. If any connection is lost or there is an anomaly on a specific web page that causes an error in the script, I can easily revisit those pages in a second pass.

```
def scrape_teacher_quality(district, cache_dir=None):
    """
    Takes a district id and name as a tuple and scrapes the relevant page.
    """
    from BeautifulSoup import BeautifulSoup
    # load data, either from web or cached directory
    if cache_dir is None:
        url = build_url(district)
        try:
            soup = BeautifulSoup(urllib2.urlopen(url).read())
        except:
            print "error loading url:", url
            return
    else:
        try:
            file_in = os.path.join(cache_dir, '%s_%s.html' % district)
            soup = BeautifulSoup(open(file_in, 'r').read())
        except:
            print "file not found:", file_in
            return

    quality_div = soup.find('div', attrs={'id':'TeacherQuality'}) # ❶

    header = ['district_id', 'district_name']
    data = list(district)

    for i, table in enumerate(quality_div.findAll('table')):
        header_prefix = 'table%s ' % i  # ❷

        header_cells = table.findAll('th')
        data_cells   = table.findAll('td')
```

```
        # some rows have an extra 'th' cell. If so, we need to skip it.
        if len(header_cells) == len(data_cells) + 1:
            header_cells = header_cells[1:]

        header.extend([header_prefix + th.text.strip() for th in header_cells])
        data.extend([td.text.strip() for td in data_cells])

    return {
        'header' : header,
        'data'   : data
    }
```

❶ On this page, all of the different tabs are in `div` tags with appropriate IDs, and all but the active `div` are commented out.

❷ Since there are three tables and some have the same title, I'll give the headers a prefix. In fact, headers here aren't necessary if the data within the tables are always the same. I included them here as an example.

This code will often be very specific to the way that a web page has been coded. If a website gets redesigned, this function will probably have to be amended, so it should rely on the other functions as little as possible. You can also write functions to scrape the other tables pertaining to this school district such as NCLB Status and Adequate Yearly Progress (AYP) and execute them at the same time as this one.

Now that we have all of the pieces, we can put them all together in the `main()` function. I will write the results to a comma-delimited flat file, but you could just as easily store them in a relational or NoSQL database, or any other kind of datastore.

```
from BeautifulSoup import BeautifulSoup
import urllib2, os, csv
def main():
    """
    Main funtion for crawling the data.
    """
    district_pages = get_district_pages('http://www.sde.idaho.gov/reportcard')

    errorlog = open('errors.log', 'wb')
    for district in district_pages:
        print district
        try:
            cache_page(district, 'data')
        except:
            url = build_url(district)
            errorlog.write('Error loading page %s\n' % url)
            continue


    fout = csv.writer(open('parsed_data.csv', 'wb')) # ❶

    # write the header
```

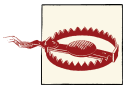
```
header = [
    'district_id',
    'district_name',
    'ba_degree',
    'ba_plus_12',
    'ba_plus_24',
    'ma_degree',
    'ma_plus_12',
    'ma_plus_24',
    'phd_degree',
    'degree_total',
    'emergency_cert',
    'poverty_high',
    'poverty_low',
    'poverty_total',
]
fout.writerow(header)

for district in district_pages:
    parsed_dict = scrape_teacher_quality(district, cache_dir='data')
    # ❷
    if parsed_dict is not None:
        fout.writerow(parsed_dict['data'])
    else:
        errorlog.write('Error parsing district %s (%s)' % district)

del(fout)
del(errorlog)
```

❶ Python's `csv` package is excellent at handling escaped characters and any other annoyances that may creep up when working with flat files and is a much better option than writing strings directly.

❷ If you wanted to scrape other tables from the page, just drop the function call to the table's parser function here.



This makes the strong assumption that the tables are identical on every page. This may not always be the case, so you should store the values as key/value in a `dict` or something similar.

The Real Difficulties

I spent a few years as a full-time research assistant at a prominent business school. Part of my duties included gathering and cleaning datasets for research projects. I crawled many websites to reproduce the clean dataset that I knew sat behind the web pages. Here are a few of the more creative projects I came up against.

Download the Raw Content If Possible

In my first web scraping application, I had to go through about 20GB of raw text files stored on an external website and do some simple text extraction that relied mostly on matching some regular expressions. I knew it would take a while to download all of the data, so I figured I would just process the files line-by-line as I download them.

The script I wrote was pretty simple. I used Python's `urllib2` library to open a remote file and scan through it line-by-line, the same way you would for a local file. I let it run overnight and came in the next morning to find that I couldn't access the Internet at all. It turns out that requesting 20GB of data a few hundred bytes at a time looks an awful lot like a college student downloading movies through a torrent service. It was my third day of a new job and I had already blacklisted myself. And that's why you always download a batch of data and parse it locally.

Forms, Dialog Boxes, and New Windows

The structure of most of the websites that I encountered had a similar data retrieval pattern: fill out a form, reverse-engineer the results page to extract the meaningful data, repeat. One online database, however, had a more complex pattern:

1. Log in to the website.
2. Fill out a form with a company name and date range.
3. Dismiss a dialog box ("Do you really want to submit this query?").
4. Switch to the new tab that the results open in; extract the data.
5. Close the new tab, returning to the original form.
6. Repeat.

For this, I used my favorite last-resort tool: Chickenfoot[3], which is a Firefox plug-in that allows you to programmatically interact with a web page through the browser. From the web server's perspective, all of the requests are as if a real person was submitting these queries through their browser. Here is a simple example that will load the Idaho NCLB page that we crawled and go to the first school's "Teacher Quality" page.

```
go('http://www.sde.idaho.gov/ReportCard');
select('Report Card for 2009-2010');
click('submit');
click('Teacher Quality');
```

3. *https://github.com/bolinfest/chickenfoot/*

Chickenfoot provides a few simple commands like `click`, `select`, and `find`, but also gives you full access to the Javascript DOM for extracting data and fully controlling the webpage. It is easy to make AJAX requests, modify and submit forms, and pretty much anything else. The disadvantage is that it is slow since it runs in the browser and has to render each page, and doesn't always fail gracefully if a page doesn't load properly.

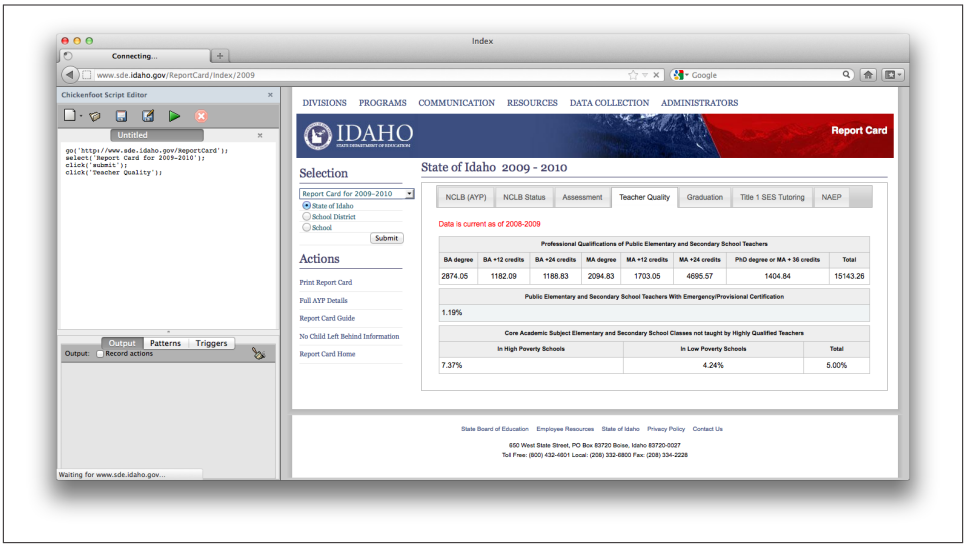


Figure 5-3. Screenshot of Chickenfoot in action.

Flash

We went through an example of scraping data from the Idaho No Child Left Behind report card website, but not all states are as easy to scrape as that one. Some states actually provide the raw data in `.csv` format, which is great, but others are trickier to get data out of.

One particular state (I forget which one, unfortunately) decided that Adobe Flash would be the best way to display this data. Each school has its own web page with a Flash applet showing a green (excellent), yellow (proficient), or red (failing) light for each of the subjects on which it is evaluated. Adobe Flash is an incredibly powerful platform, and for better or worse, it is a brick wall when it comes to accessing the underlying source code. That means that we had to get creative when gathering this data.

My colleagues and I put this state aside while we gathered the NCLB data for the others. We had already built the list of URLs that we would have to navigate to for each shool, but had to figure out how to get the data out of Flash. Eventually, it came to us: AppleScript. AppleScript can be a helpful tool for automating high-level tasks in OSX such as opening and closing programs, accessing calendar information in iCal, and visiting web

pages in the Safari web browser. We wrote an AppleScript to tell Safari to navigate to the correct URL, then tell OSX to take a screen shot of the page, saving it with an ID number in the file name. Then we could analyze the images afterwards and look at the color of specific pixels to determine the school's performance.

It would have to run at night, because we needed the green/yellow/red circles to be equally aligned on every page. That meant the browser window had to be active and in the same place for each screen shot. Each morning, for about a week, we came into the office to find about 2GB of screenshot images.

We used a MATLAB script to loop through all of these results and get the color of the pixels of interest. When MATLAB loads a `.jpg` image file that is `N` pixels tall and `M` pixels wide, it stores it as a three-dimensional array with dimensions `NxMx3`. Each of the *layers* holds the red, green, and blue color intensity for the relevant pixel with a number between 0 and 255. Red pixels would have the value `[255, 0, 0]`, green pixels were `[0, 255, 0]`, and yellow pixels were `[255, 255, 0]`.

The Dark Side

Sometimes you're trying to get data from a site that doesn't want to be crawled (even though you've confirmed that you're allowed to crawl the pages, right?). If your crawler is detected, your IP address or API key will likely be blacklisted and you won't be able to access the site. This once happened to a coworker of mine and the entire office was blocked from accessing an important website. If you're writing a web crawler similar to the example in this chapter on a page that you suspect will block you, it's best to insert pauses in the crawler so that your script is less likely to be detected. This way your page requests will look less programmatic and the frequency of the requests might be low enough that they don't annoy the website's administrators.

Conclusion

Any "screen scraping" program is subject to many factors that are beyond your control and which can make the program less reliable. Data changes as websites are updated, so saving an offline version of those pages is a priority over actually parsing and extracting the live data. Slow connections cause timeouts when loading pages, so your program has to fail gracefully and move on, keeping a history of what you were and were not able to save so that you can make a second (or third or fourth) pass to get more data. However, it is sometimes a fun challenge to reverse-engineer a website and figure out how they do things under the hood, notice common design approaches, and end up with some interesting data to work with in the end.

CHAPTER 6

Detecting Liars and the Confused in Contradictory Online Reviews

Jacob Perkins

Did you know that people lie for their own selfish reasons? Even if this is totally obvious to you, you may be surprised at how blatant this practice has become online, to the point where some people will explain their reasons for lying immediately after doing so.

I knew unethical people would lie in online reviews in order to inflate ratings or attack competitors, but what I didn't know, and only learned by accident, is that individuals will sometimes write reviews that completely contradict their associated rating, without any regard to how it affects a business's online reputation. And often this is for businesses that an individual likes.

How did I learn this? By using ratings and reviews to create a sentiment corpus, I trained a sentiment analysis classifier that could reliably determine the sentiment of a review. While evaluating this classifier, I discovered that it could also detect discrepancies between the review sentiment and the corresponding rating, thereby finding liars and confused reviewers. Here's the whole story of how I used text classification to identify an unexpected source of bad data...

Weotta

At my company, Weotta,[1] we produce applications and APIs for navigating local data in ways that people actually care about, so we can answer questions like: Is there a kid-friendly restaurant nearby? What's the nearest hip yoga studio? What concerts are happening this weekend?

1. *http://www.weotta.com*

To do this, we analyze, aggregate, and organize local data in order to classify it along dimensions that we can use to answer these questions. This classification process enables us to know which restaurants are classy, which bars are divey, and where you should go on a first date. Online business reviews are one of the major input signals we use to determine these classifications. Reviews can tell us the positive or negative sentiment of the reviewer, as well as what they specifically care about, such as quality of service, ambience, and value. When we aggregate reviews, we can learn what's popular about the place and why people like or dislike it. We use many other signals besides reviews, but with the proper application of natural language processing,[2] reviews are a rich source of significant information.

Getting Reviews

To get reviews, we use APIs where possible, but most reviews are found using good old-fashioned web scraping. If you can use an API like CityGrid[3] to get the data you need, it will make your life much easier, because while scraping isn't necessarily difficult, it can be very frustrating. Website HTML can change without notice, and only the simplest or most advanced scraping logic will remain unaffected. But the majority of web scrapers will break on even the smallest of HTML changes, forcing you to continually monitor and maintain your scrapers. This is the dirty secret of web mining: the end result might be nice and polished data, but the process is more akin to janitorial work where every mess is unique and it never stays clean for long.

Once you've got reviews, you can aggregate ratings to calculate an average rating for a business. One problem is that many sources don't include ratings with their reviews. So how can you accurately calculate an average rating? We wanted to do this for our data, as well as aggregate the overall positive sentiment from all the reviews for a business, independent of any average rating. With that in mind, I figured I could create a sentiment classifier,[4] using rated reviews as a training corpus. A *classifier* works by taking a *feature set* and determining a *label*. For sentiment analysis, a feature set is a piece of text, like a review, and the possible labels can be `pos` for positive text, and `neg` for negative text. Such a sentiment classifier could be run over a business's reviews in order to calculate an overall sentiment, and to make up for any missing rating information.

2. *http://en.wikipedia.org/wiki/Natural_language_processing*
3. *http://citygrid.com/*
4. *http://bit.ly/X9sqWR*

Sentiment Classification

NLTK,[5] Python's Natural Language ToolKit, is a very useful programming library for doing natural language processing and text classification.[6] It also comes with many corpora that you can use for training and testing. One of these is the `movie_reviews` corpus, [7] and if you're just learning how to do sentiment classification, this is a good corpus to start with. It is organized into two directories, `pos` and `neg`. In each directory is a set of files containing movie reviews, with every review separated by a blank line. This corpus was created by Pang and Lee,[8] and they used ratings that came with each review to decide whether that review belonged in `pos` or `neg`. So in a 5-star rating system, 3.5 stars and higher reviews went into the `pos` directory, while 2.5 stars and lower reviews went into the `neg` directory. The assumption behind this is that high rated reviews will have positive language, and low rated reviews will have more negative language. Polarized language is ideal for text classification, because the classifier can learn much more precisely those words that indicate `pos` and those words that indicate `neg`.

Because I needed sentiment analysis for local businesses, not movies, I used a similar method to create my own sentiment training corpus for local business reviews. From a selection of businesses, I produced a corpus where the `pos` text came from 5 star reviews, and the `neg` text came from 1 star reviews. I actually started by using both 4 and 5 star reviews for `pos`, and 1 and 2 star reviews for `neg`, but after a number of training experiments, it was clear that the 2 and 4 star reviews had less polarizing language, and therefore introduced too much noise, decreasing the accuracy of the classifier. So my initial assumption was correct, though the implementation of it was not ideal. But because I created the training data, I had the power to change it in order to yield a more effective classifier. All training-based machine learning methods work on the general principle of "garbage in, garbage out," so if your training data is no good, do whatever you can to make it better before you start trying to get fancy with algorithms.

Polarized Language

To illustrate the power of polarized language, what follows is a table showing some of the most polarized words used in the `movie_reviews` corpus, along with the occurrence count in each category, and the Chi-Squared information gain, calculated using NLTK's `BigramAssocMeasures.chi_sq()` function in the `nltk.metrics.association` module.

5. *http://nltk.org*
6. *http://en.wikipedia.org/wiki/Text_classification*
7. *http://www.cs.cornell.edu/people/pabo/movie-review-data/*
8. *http://www.cs.cornell.edu/home/llee/papers/sentiment.pdf*

[9] This Chi-Square metric is a modification of the Phi-Square measure of the association between two variables, in this case a word and a category. Given the number of times a word appears in the `pos` category, the total number of times it occurs in all categories, the number of words in the `pos` category, and the total number of words in all categories, we get an association score between a word and the `pos` category. Because we only have two categories, the `pos` and `neg` scores for a given word will be the same because the word is equally significant either way, but the interpretation of that significance depends on the word's relative frequency in each category.

Word	Pos	Neg	Chi-Sq
bad	361	1034	399
worst	49	259	166
stupid	45	208	123
life	1057	529	126
boring	52	218	120
truman	152	11	108
why	317	567	99
great	751	397	76
war	275	94	71
awful	21	111	71

Some of the above words and numbers should be mostly obvious, but others not so much. For example, many people might think "war" is bad, but clearly that doesn't apply to movies. And people tend to ask "why" more often in negative reviews compared to positive reviews. Negative adjectives are also more common, or at least provide more information for classification than the positive adjectives. But there can still be category overlap with these adjectives, such as "not bad" in a positive review, or "not great" in a negative review. Let's compare this to some of the more common and less polarizing words:

Word	Pos	Neg	Chi-Sq
good	1248	1163	0.6361
crime	115	93	0.6180
romantic	137	117	0.1913
movies	635	571	0.0036
hair	57	52	0.0033
produced	67	61	0.0026
bob	96	86	0.0024

9. *http://bit.ly/QibGfE*

Word	Pos	Neg	Chi-Sq
where	785	707	0.0013
face	188	169	0.0013
take	476	429	0.0003

You can see from this that "good" is one of the most overused adjectives and confers very little information. And you shouldn't name a movie character "Bob" if you want a clear and strong audience reaction. When training a text classifier, these low-information words are harmful noise, and as such should either be discarded or weighted down, depending on the classification algorithm you use. Naive Bayes[10] in particular does not do well with noisy data, while a Logistic Regression classifier (also known as Maximum Entropy)[11] can weigh noisy features down to the point of insignificance.

Corpus Creation

Here's a little more detail on the corpus creation: we counted each review as a single *instance*. The simplest way to do this is to replace all newlines in a review with a single space, thereby ensuring that each review looks like one paragraph. Then separate each review/paragraph by a blank line, so that it is easy to identify when one review ends and the next begins. Depending on the number of reviews per category, you may want to have multiple files per category, each containing some reasonable number of reviews separated by blank lines. With multiple files, you should either have separate directories for `pos` and `neg`, like the `movie_reviews` corpus, or you could use easily identified file-name patterns. The simplest way to do it is to copy something that already exists, so you can reuse any code that recognizes that organizational pattern. The number of reviews per file is up to you; what really matters is the number of reviews per category. Ideally you want at least 1000 reviews in each category; I try to aim for at least 10,000, if possible. You want enough reviews to reduce the bias of any individual reviewer or item being reviewed, and to ensure that you get a good number of significant words in each category, so the classifier can learn effectively.

The other thing you need to be concerned about is category balance. After producing a corpus of 1 and 5 star reviews, I had to limit the number of `pos` reviews significantly in order to balance the `pos` and `neg` categories, because it turns out that there's far more 5 star reviews than there are 1 star reviews. It seems that online, most businesses are above average, as you can see in this chart showing the percentage of each rating.

10. *http://en.wikipedia.org/wiki/Naive_Bayes_classifier*
11. *http://en.wikipedia.org/wiki/Maximum_entropy_classifier*

Rating	Percent
5	32%
4	35%
3	17%
2	9%
1	7%

People are clearly biased towards higher rated reviews; there are nearly five times as many 5 star reviews as 1 star reviews. So it might make sense that a sentiment classifier should be biased the same way, and all else being equal, favor `pos` classifications over `neg`. But there's a design problem here: if a sentiment classifier is more biased towards the `pos` class, it will produce more false positives. And if you plan on surfacing these positive reviews, showing them to normal people that have no insight into how a sentiment classifier works, you really don't want to show a false positive review. There's a lot of cognitive dissonance when you claim that a business is highly rated and most people like it, while at the same time showing a negative review. One of the worst things you can do when designing a user interface is to show conflicting messages at the same time. So by balancing the `pos` and `neg` categories, I was able to reduce that bias and decrease false positives. This was accomplished by simply pruning the number of `pos` reviews until it was equal to the number of `neg` reviews.

Training a Classifier

Now that I had a polarized and balanced training corpus, it was trivial to train a classifier using a classifier training script from nltk-trainer.[12] nltk-trainer is an open source library of scripts I created for training and analyzing NLTK models. For text classification, the appropriate script is `train_classifier.py`. Just a few hours of experimentation lead to a highly accurate classifier. Below is an example of how to use `train_classifier.py`, and the kind of stats I saw:

```
nltk-trainer$ ./train_classifier.py review_sentiment --no-pickle \
    --classifier MEGAM --ngrams 1 --ngrams 2 --instances paras \
    --fraction 0.75
loading review_sentiment
2 labels: ['neg', 'pos']
22500 training feats, 7500 testing feats
[Found megam: /usr/local/bin/megam]
training MEGAM classifier
accuracy: 0.913465
```

12. *https://github.com/japerk/nltk-trainer*

```
neg precision: 0.891415
neg recall: 0.931725
pos precision: 0.947058
pos recall: 0.910265
```

With these arguments, I'm using the `MEGAM` algorithm for training a `MaxentClassifier` using each review paragraph as a single instance, looking at both single words (unigrams) and pairs of words (bigrams). The `MaxentClassifier` (or Logistic Regression), uses an iterative algorithm to determine weights for every feature. These weights can be positive or negative for a category, meaning that the presence of a word can imply that a feature set belongs to a category and/or that a feature set **does not** belong to different category. So referring to the previous word tables, we can expect that "worst" will have a positive weight for the `neg` category, and a negative weight for the `pos` category. The `MEGAM` algorithm is just one of many available training algorithms, and I prefer it for its speed, memory efficiency, and slight accuracy advantage over the other available algorithms.

The other options used above are `--no-pickle`, which means to not save the trained classifier to disk, and `--fraction`, which specifies how much of the corpus is used for training, with the remaining fraction used for testing. `train_classifier.py` has many other options, which you can see by using the `--help` option. These include various algorithm-specific training options, what constitutes an *instance*, which ngrams to use, and many more.

If you're familiar with classification algorithms, you may be wondering why I didn't use Naive Bayes. This is because my tests showed that Naive Bayes was much less accurate than Maxent, and that even combining the two algorithms did not beat Maxent by itself. Naive Bayes does not weight its features, and therefore tends to be susceptible to noisy data, which I believe is the reason it did not perform too well in this case. But your data is probably different, and you may find opposite results when you conduct your experiments.

I actually wrote the original code behind `train_classifier.py` for this project so that I could design and modify classifier training experiments very quickly. Instead of copy and paste coding and endless script modifications, I was able to simply tweak command line arguments to try out a different set of training parameters. I encourage you to do the same, and to perform many training experiments in order to arrive at the best possible set of options.

After I'd created this script for text classification, I added training scripts for part-of-speech tagging[13] and chunking,[14] leading to the creation of the whole nltk-trainer project

13. *http://en.wikipedia.org/wiki/Part-of-speech_tagging*

14. *http://en.wikipedia.org/wiki/Chunking_(computational_linguistics)*

and its suite of training and analysis scripts. I highly recommend trying these out before attempting to create a custom NLTK based classifier, or any NLTK model, unless you really want to know how the code works, and/or have custom feature extraction methods you want to use.

Validating the Classifier

But back to the sentiment classifier: no matter what the statistics say, over the years I've learned to not fully trust trained models and their input data. Unless every training instance has been hand-verified by three professional reviewers, you can assume there's some noise and/or inaccuracy in your training data. So once I had trained what appeared to be a highly accurate sentiment classifier, I ran it over my training corpus in order to see if I could find reviews that were misclassified by the classifier. My goal was to figure out where the classifier went wrong, and perhaps get some insight into how to tweak the training parameters for better results. To my surpise, I found reviews like this in the pos/5-star category, which the classifier was classifying as `neg`:

> It was loud and the wine by the glass is soo expensive. Thats the only negative because it was good.

And in the neg/1-star category, there were `pos` reviews like this:

> One of the best places in New York for a romantic evening. Great food and service at fair prices.

> We loved it! The waiters were great and the food came quickly and it was delicious. 5 star for us!

The classifier actually turned out to be more accurate at detecting sentiment than the ratings used to create the training corpus! Apparently, one of the many bizarre things people do online is write reviews that completely contradict their rating. While trying to create a sentiment classifier, I had accidentally created a way to identify both liars and the confused. Here's some 1-star reviews by blatant liars:

> This is the best BBQ ever! I'm not just saying that to keep you fools from congesting my favorite places.

> Quit coming to my favorite Karaoke spot. I found it first.

While these at least have some logic behind them, I completely disagree with the motivation. By giving 1 star with their review, these reviewers are actively harming the reputation of their favorite businesses for their own selfish short-term gain.

On the other side of the sentiment divide, here's a mixed sentiment comment from a negative 5-star review:

> This place sucks, do not come here, dirty, unfriendly staff and bad workout equipment. MY club, do you hear me, MY, MY, MY club. STAY AWAY! One of the best clubs in the bay area. All jokes aside, this place is da bomb.

This kind of negative 5-star review could also be harming the business's reputation. The first few sentences may be a joke, but those are also the sentences people are more likely to read, and this review is saying some pretty negative things, albeit jokingly. And then there's people that really shouldn't be writing reviews:

> I like to give A's. I dont want to hurt anyones feelings. A- is the lowest I like to give. A- is the new F.

The whole point of reviews and ratings is to express your opinion, and yet this reviewer seems afraid to do just that. And here's an actual negative opinion from a 5-star review:

> My steak was way over-cooked. The menu is very limited. Too few choices

If the above review came with a 1 or 2 star rating, that'd make sense. But a 5-star rating for a limited menu and overcooked steak? I'm not the only one who's confused.

Finally, just to show that my sentiment classifier isn't perfect, here's a 5-star review that's actually positive, but the reviewer uses a double negative, which causes the classifier to give it a negative sentiment:

> Never had a disappointing meal there.

Double negatives, negations such as "not good," sarcasm, and other language idioms can often confuse sentiment analysis systems and are an area of ongoing research. Because of this, I believe that it's best to exclude such reviews from any metrics about a business. If you want a clear signal, you often have to ignore small bits of contradictory information.

Designing with Data

We use the sentiment classifier in another way, too. As I mentioned earlier, we show reviews of places to provide our users with additional context and confirmation. And because we try to show only the best places, the reviews we show should reflect that. This means that every review we show needs to have a strong positive signal. And if there's a rating included with the review, it needs to be high too, because we don't want to show any confused high-rated reviews or duplicitous low-rated reviews. Otherwise, we'd just confuse our own users.

Before adding the sentiment classifier as a critical component of our review selection method, we were simply choosing reviews based on rating. And when we didn't have a rating, we were choosing the most recent reviews. Neither of these methods was satis-

factory. As I've shown above, you cannot always trust ratings to accurately reflect the sentiment of a review. And for reviews without a rating, anyone could say anything and we had no signal to use for filtering out the negative reviews. Now some might think we should be showing negative reviews to provide a balanced view of a business. But our goal in this case is not to create a research tool—there's plenty of other sites and apps that are already great for that. Our goal is to show you the best, most relevant places for your occasion. If every other signal is mostly positive, then showing negative reviews is a disservice to our users and results in a poor experience. By choosing to show only positive reviews, the data, design, and user experience are all congruent, helping our users choose from the best options available based on their own preferences, without having to do any mental filtering of negative opinions.

Lessons Learned

One important lesson for machine learning and statistical natural language processing enthusiasts: it's very important to train your own models on your own data. If I had used classifiers trained on the standard `movie_reviews` corpus, I would never have gotten these results. Movie reviews are simply different than local business reviews. In fact, it might be the case that you'd get even better results by segmenting businesses by type, and creating classifiers for each type of business. I haven't run this experiment yet, but it might lead to interesting research. The point is, your models should be trained on the same kind of data they need to analyze if you want high accuracy results. And when it comes to text classification and sentiment analysis in particular, the domain really matters. That requires creating a custom corpus [15] and spending at least a few hours on experiments and research to really learn about your data in order to produce good models.

You must then take a critical look at your training data, and validate your training models against it. This is the only way to know what your model is actually learning, and if your training data is any good. If I hadn't done any model validation, I would never have discovered these bad reviews, nor realized that my sentiment classifier could detect inconsistent opinions and outright lying. In a sense, these bad reviews are a form of noise that has been maliciously injected into the data. So ask yourself, what forms of bad data might be lurking in your data stream?

Summary

The process I went through can be summarized as:

1. Get relevant data.

15. *http://www.slideshare.net/japerk/corpus-bootstrapping-with-nltk*

2. Create a custom training corpus.
3. Train a model.
4. Validate that model against the training corpus.
5. Discover something interesting.

At steps 3-5, you may find that your training corpus is not good enough. It could mean you need to get more relevant data. Or that the data you have is too noisy. In my case, I found that 2 and 4 star reviews were not polarizing enough, and that there was an imbalance between the number of 5-star reviews and the number of 1-star reviews.

It's also possible that your expectations for machine learning are too high, and you need to simplify the problem. Natural language processing and machine learning are imperfect methods that rely on statistical pattern matching. You cannot expect 100% accuracy, and the noisier the data is, the more likely you are to have lower accuracy. This is why you should always aim for more distinct categories, polarizing language, and simple classification decisions.

Resources

All of my examples have used NLTK, Python's Natural Language ToolKit, which you can find at *http://nltk.org/*. I also train all my models using the scripts I created in nltk-trainer at *https://github.com/japerk/nltk-trainer*. To learn how to do text classification and sentiment analysis with NLTK yourself, I wrote a series of posts on my blog, starting with *http://bit.ly/X9sqWR*. And for those who want to go beyond basic text classification, take a look at scikit-learn, which is implementing all the latest and greatest machine learning algorithms in Python: *http://scikit-learn.org/stable/*. For Java people, there is Apache's OpenNLP project at *http://opennlp.apache.org/*, and a commercial library called LingPipe, available at *http://alias-i.com/lingpipe/*.

CHAPTER 7

Will the Bad Data Please Stand Up?

Philipp K. Janert

Among hikers and climbers, they say that "there is no such thing as bad weather—only inappropriate clothing." And as anybody who has spent some time outdoors can attest, it is often precisely trips undertaken under more challenging circumstances that lead to the most noteworthy memories. But one has to be willing to put oneself out there.

In a similar spirit, I don't think there is really such a thing as "bad data"—only inappropriate approaches. To be sure, there are datasets that require more work (because of missing data, background noise, poor encoding, inconvenient file formats, and so on), but they don't pose fundamental challenges. Given sufficient effort, these problems can be overcome, and there are useful techniques for handling such situations (like tricks for staying warm during a late-November hike).

But basically, that's remaining within familiar territory. To discover new vistas, one has to be willing to follow an unmarked trail and see where it leads. Or equivalently, when working with data, one has to dare to have an opinion about where the data is leading and then check whether one was right about it. Note that this takes courage: it is far safer to merely describe what one sees, but doing so is missing a whole lot of action.

Let's evaluate some trail reports. Later, we'll regroup and see what lessons we have learned.

Example 1: Defect Reduction in Manufacturing

A manufacturing company had developed a rather clever scheme to reduce the number of defective items shipped to their customers—at no additional cost. The basic idea was to use a quantity that was already being measured for each newly manufactured item (let's call it the "size"—it wasn't, but it won't matter) as indicator of the item quality. If

the size was "off," then the item probably was not going to work right. To the manufacturer, the key benefit of this indirect approach was the low cost. The size was already being measured as part of the manufacturing process, so it did not impose an additional overhead. (The main problem with quality assurance always is that it has to be cheap.)

They put a system in place that flagged items that seemed "off" as candidates for manual inspection. The question was: how well was this system working? How good was it at actually detecting defective items? This was not so easy to tell, because the overall defect rate was quite low: about 1 item in 10,000 leaving the manufacturing line was later found to be truly defective. What fraction of defects would this new tagging system be able to detect, and how many functioning items would it incorrectly label as defective? (This was a key question. Because all flagged items were sent to manual testing, a large number of such false positives drove up the cost quickly—remember, the idea was for the overall process to be cheap!)

There's the assignment. What would *you* do?

Well, the central, but silent, assumption behind the entire scheme is that there is a "typical" value for the size of each item, and that the observed (measured) values will scatter in some region around it. Only if these assumptions are fulfilled does it even make sense to say that one particular item is "off," meaning outside the typical range. Moreover, because we try to detect a 1 in 10,000 effect, we need to understand the distribution out in the tails.

You can't tell 0.01% tail probabilities from a histogram, so you need to employ a more formal method to understand the shape of the point distribution, such as a probability plot. *Which* probability plot? To prepare one, you have to select a specific distribution. Which one? Is it obvious that the data will be Gaussian distributed?

No, it is not. But it is a reasonable choice: one of the assumptions is that the observed deviations from the "typical" size are due to random effects. The Gaussian distribution, which describes the sum of many random contributions, should provide a good description for such a system.

A typical probability plot for data from the manufacturing plant is shown in Figure 7-1. If the points fall onto a straight line in a probability plot, then this indicates that the data is indeed distributed according to the theoretical distribution. Moreover, the intercept of the line yields the empirical mean of the dataset and the slope of the line the standard deviation. (The size of the data was only recorded to within two decimal places, resulting in the step-like appearance of the plot.)

In light of this, Figure 7-1 may look like an excellent fit, but in fact it is *catastrophic!* Remember that we are trying to detect a 1 in 10,000 effect — in other words, we expect only about 1 in 10,000 items to be "off," indicating a possibly defective item. Figure 7-1 shows data for 10,000 items, about 20 of which are rather obviously outliers. In other

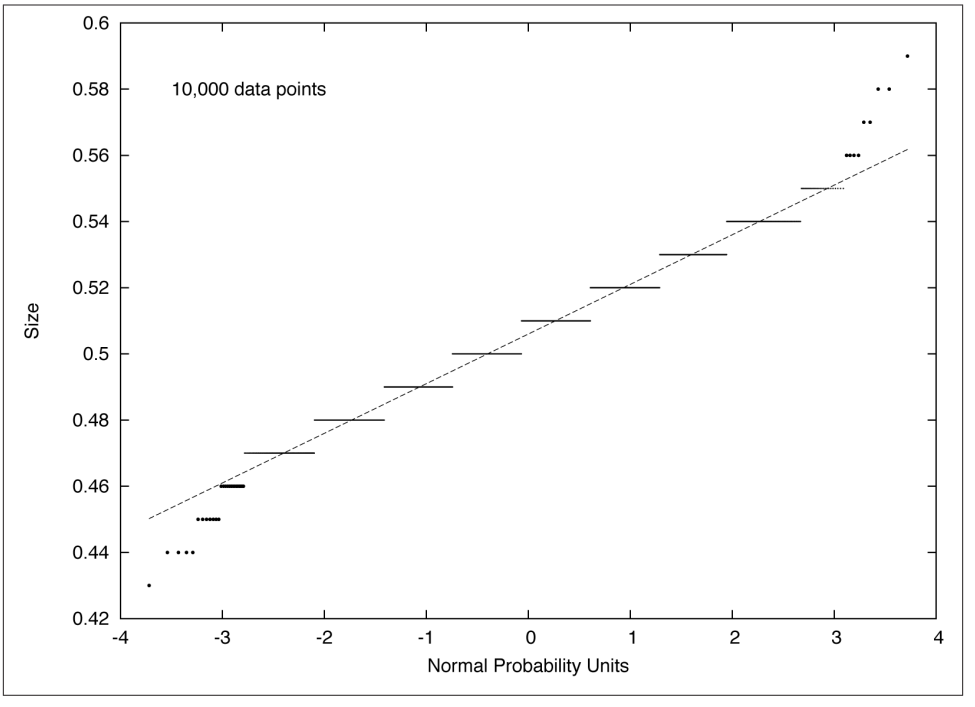


Figure 7-1. A normal probability plot. Note how the tails of the dataset do not agree with the model.

words, the number of outliers is about 20 times larger than expected! Because the whole scheme is based on the assumption that only defective items will lead to measurements that are "off," Figure 7-1 tells us that we are in trouble: the number of outliers is 20 times larger than what we would expect.

Figure 7-1 poses a question, though: what causes these outliers? They are relatively few in absolute terms, but much more frequent than would be allowed on purely statistical grounds. But these data points are there. Something must cause them. What is it?

Before proceeding, I'd like to emphasize that with this last question, we have left the realm of the data itself. The answer to this question cannot be found by examining the dataset—we are now examining the environment that produced this data instead. In other words, the skills required at this point are less those of the "data scientist," but more those one expects from a "gum shoe detective." But sometimes that's what it takes.

In the present case, the data collection process was fully automated, so we rule out manual data entry errors. The measuring equipment itself was officially calibrated and regularly checked and maintained, so we rule out any systematic malfunction there. We audited the data processing steps after the data had been obtained, but found nothing:

data was not dropped, munged, overwritten—the recording seemed faithful. Ultimately, we insisted on visiting the manufacturing plant itself (hard hat, protective vest, the whole nine yards). Eventually, we simply observed the measuring equipment for several hours, from a distance. And then it became clear: every so often (a few times per hour) an employee would accidentally bump against the apparatus. Or an item would land heavily on a nearby conveyor. Or a fork lift would go by. All these events happened rarely—but still more frequently than the defects we tried to detect!

Ultimately, the "cheap" defect reduction mechanism envisioned by the plant management was not so cheap at all: to make it work, it would be necessary to bring the data collection step up to the same level of accuracy and repeatability as was the case for the main manufacturing process. That would change the economics of the whole project, making it practically infeasible from an economic perspective.

Is this a case of "bad data?" Certainly, if you simply want the system to *work*. But the failure is hardly the data's fault—the data itself never claimed that it would be suitable for the intended purpose! But nobody had bothered to state the assumptions on which the entire scheme rested clearly and in time and to validate that these assumptions were, in fact, fulfilled.

It is not fair to blame the mountain for being covered with snow when one didn't bother to bring crampons.

Example 2: Who's Calling?

Paradoxically, wrong turns can lead to the most desirable destinations. They can take us to otherwise hidden places—as long as we don't cling too strongly to the original plan and instead pay attention to the *actual* scenery.

Figure 7-2 shows a histogram for the number of phone calls placed per business day to a small business—let's say, a building contractor. (The histogram informs us that there were 17 days with no calls, 32 days with one call, 20 days with two calls, and so on.) Knowing nothing else, what can we say about this system?

Well, we might expect the calls to be distributed according to a Poisson distribution:

$$\text{Probability for } k \text{ calls per day} = p(k, \lambda) = e^{-\lambda}\frac{\lambda^k}{k!}$$

The Poisson distribution is the natural choice to model the frequency of rare events. It depends only on a single parameter, λ, the average number of calls per day. Given the data in the histogram, it is easy to obtain a numerical estimate for the parameter:

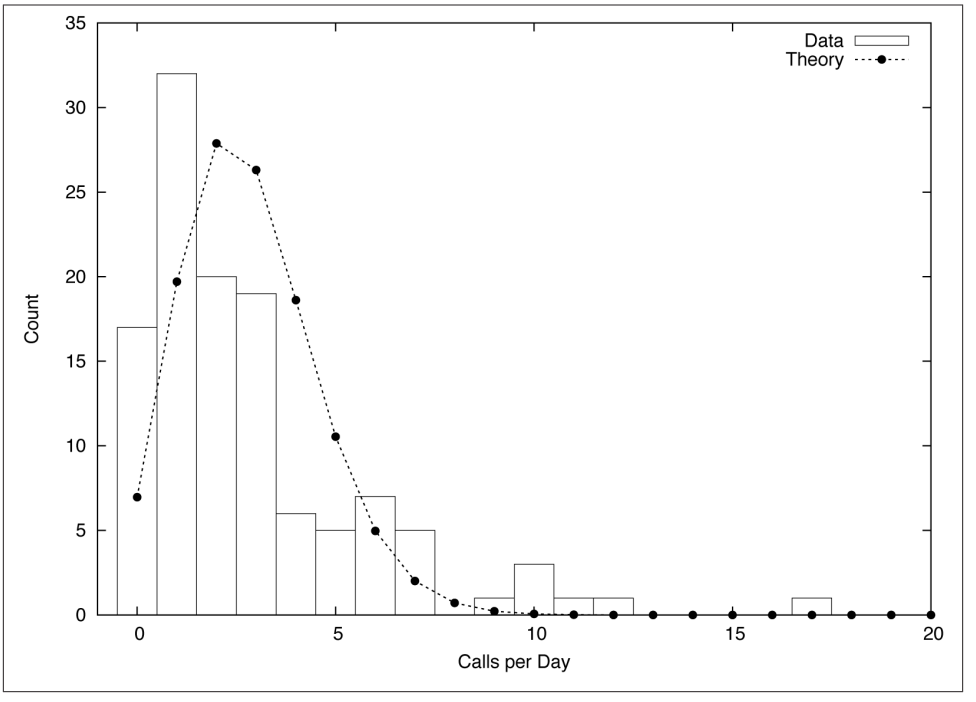


Figure 7-2. Histogram of calls received per business day by a small business, together with the best fit Poisson distribution.

$$\lambda = \frac{\text{total number of calls}}{\text{total number of days}}$$

Once λ is fixed, the distribution $p(k, \lambda)$ is completely determined. It should therefore fit the data without further adjustments. But look what happens! The curve based on the best estimate for λ fits the data very poorly (see Figure 7-2). Clearly, the Poisson distribution is *not* the right model for this data.

But how can that be? The Poisson model *should* work: it applies very generally as long as the following three conditions are fulfilled:

- Events occur at a constant rate
- Events are independent of each other
- Events do not occur simultaneously

However, as Figure 7-2 tells us, at least one of those conditions must be violated. If it weren't, the model would fit. So, which one is it?

It can't be the third: by construction, phones do not allow for this possibility. It could be the first, but a time series plot of the calls per day does not exhibit any obvious changes in trend. (Remember that we are only considering *business* days, and have thereby already excluded the influence of the weekend.) That leaves the second condition. Is it possible that calls are not independent? How can we know—is there any information that we have not used yet?

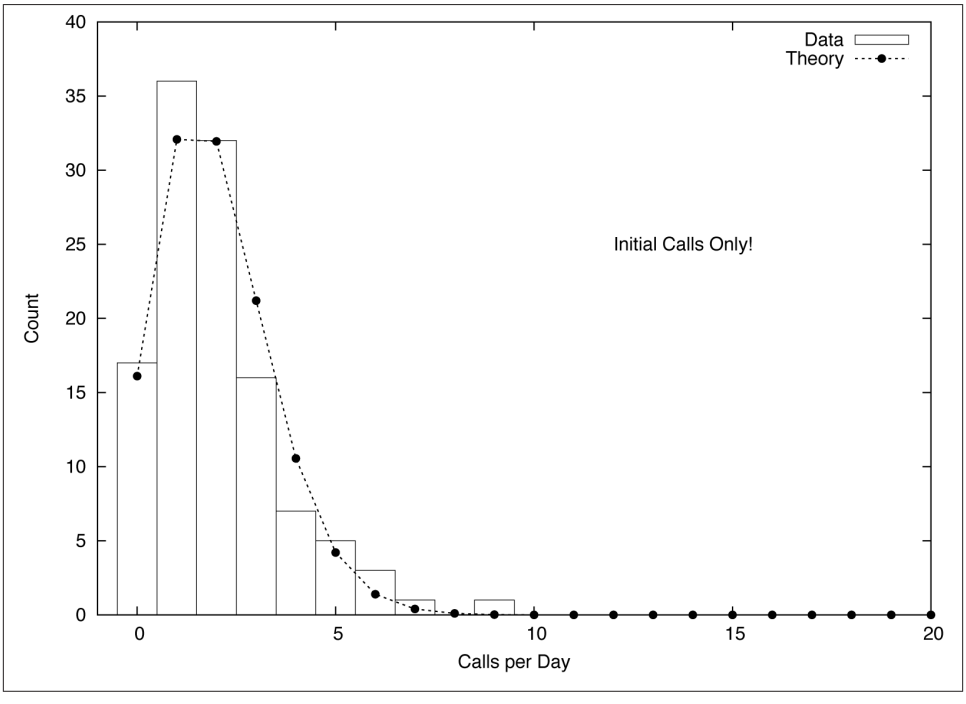


Figure 7-3. Same as Figure 7-2, but this time taking into account only the first call from each caller (that is, ignoring follow-up calls).

Yes, there is. So far we have ignored the identity of the *callers*. If we take this information into account, it turns out that about one third of all calls are *follow-up* calls from callers that have called at least once before. Obviously, follow-up calls are not independent of their preceding calls and we should not expect the Poisson model to represent them well. The histogram in Figure 7-3 includes only the *initial* call for each caller, and it turns out that the Poisson distribution now describes this dataset reasonably well.

The lesson here is that our innocuous dataset contains two different types of calls: initial calls and follow-ups. Both follow different patterns and need to be treated separately if we want to understand this system fully. In hindsight, this is obvious (almost all dis-

coveries are), but it wasn't obvious when we started. What precipitated this "discovery" was a failure: the failure of the data to fit the model we had proposed. But this failure could only occur because we had stuck our neck out and actually made a concrete proposition!

This is extremely important: to gain insight that goes beyond the merely descriptive, we need to formulate a prescriptive statement. In other words, we need to make a statement about what we *expect* the data to do (not merely what we already know it does—that would be descriptive). To make such a statement, we typically have to combine specific observations made about the data with other information or with abstract reasoning in order to arrive at a hypothetical theory—which, in turn, we can put to the test. If the theory makes predictions that turn out to be true, we have reason to believe that our reasoning about the system was correct. And *that* means that we now know *more* about the system than the data itself is telling us directly.

So, was this "bad data?" You betcha—and thankfully so. Only because it was "bad" did it help us to learn something new.

Example 3: When "Typical" Does Not Mean "Average"

Although it has never happened to me, I have heard stories of people getting on the highway in the wrong direction, and not noticing it until they ended up on the beach instead of in the mountains. I can see how it could happen. The road is straight, there are no intersections—what could possibly go wrong? One could infer from this that even if there are no intersections to worry about, it is worth confirming the *direction* one is going in.

There is a specific trap when working with data that can have an equally devastating effect: producing results that are *entirely* off the mark—and you won't even know it! I am speaking of highly skewed (specifically: power-law) point distributions. Unless they are diagnosed and treated properly, they will ruin all standard calculations. Deceivingly, the results will look just fine but will be next to meaningless.

Such datasets occur all the time. A company may serve 2.6 million web pages per month and count 100,000 unique visitors, thus concluding that the "typical visitor" consumes about 26 page views per month. A retailer carries 1 million different types of items and ships 50 million units and concludes that it ships "on average" 50 units of each item. A service provider has 20,000 accounts, generating a total of $5 million in revenue, and therefore figures that "each account" is worth $250.

In all these cases (and many, many more), the apparently obvious conclusions will turn out to be very, very wrong. Figure 7-4 shows a histogram for the first example, which exhibits the features typical of all such situations. The two most noteworthy features are the very large number of visitors producing only very few (one or two) page views per month, and the very small number of visitors generating an excessively large number

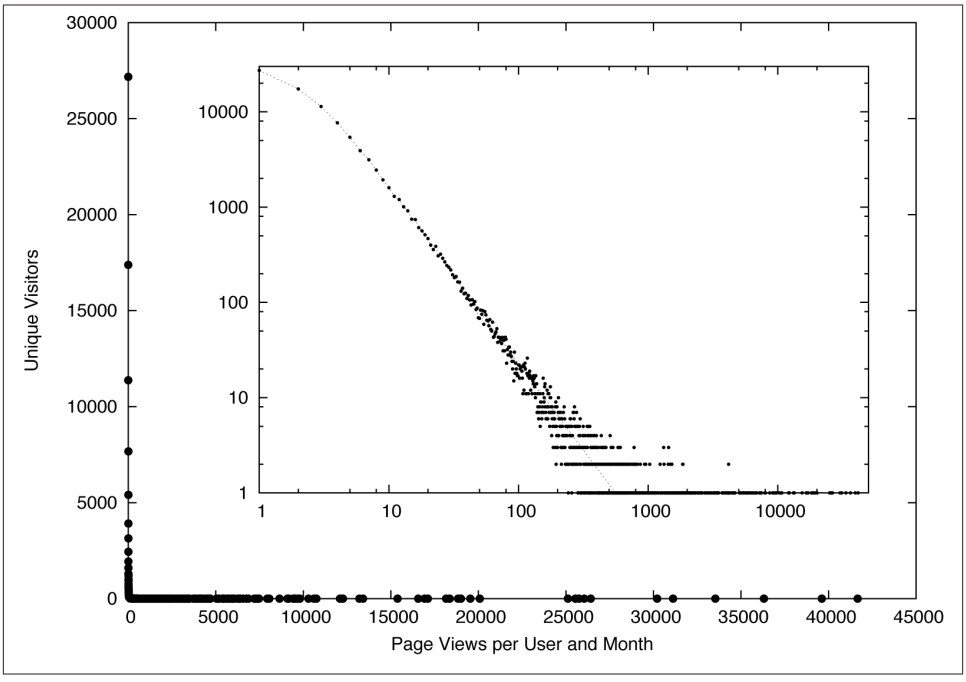


Figure 7-4. Histogram of the number of page views generated by each user in a month. The inset shows the same data using double-logarithmic scales, revealing power-law behavior.

of views. The "typical visitor" making 26 views is not typical for anything: not for the large majority of visitors making few visits and not typical for the majority of page views either, which stem from the handful of visitors generating thousands of hits each. In the case of the retailer, a handful of items will make up a significant fraction of shipped units, while the vast majority of the catalogue ships only one or two items. And so on.

It is easy to see how the wrong conclusions were reached: not only does the methodology seem totally reasonable (what could possibly be wrong with "page views per user"?), but it is also deceptively simple to calculate. All that is required are separate counts of page views and users. To generate a graph like Figure 7-4 instead requires a separate counter for each of the 100,000 users. Moreover, 26 hits per month and user sure *sounds* like a reasonable number.

The underlying problem here is the mistaken assumption that there is such a thing as a "typical visitor." It's an appealing assumption, and one that is very often correct: there really is such a thing as the "typical temperature in New York in June" or the "average

weight of a 30 year old male." But in the three examples described above, and in many other areas that are often (but not exclusively) related to human behavior, variations are so dominant that it does not make sense to identify any particular value as "typical." Everything is possible.

How, then, can we identify cases where there is no typical value and standard summary statistics break down? Although the ultimate diagnostic tool is a double-logarithmic plot of the full histogram as shown in Figure 7-4, an early warning sign is excessively large values for the calculated width of the distribution (for the visitor data, the standard deviation comes out to roughly 437, which should be compared to the mean, of only 26). Once the full histogram information is available, we can plot the Lorenz curve and even calculate a numerical measure for the skewness of the distribution (such as the Gini coefficient) for further diagnosis and analysis.

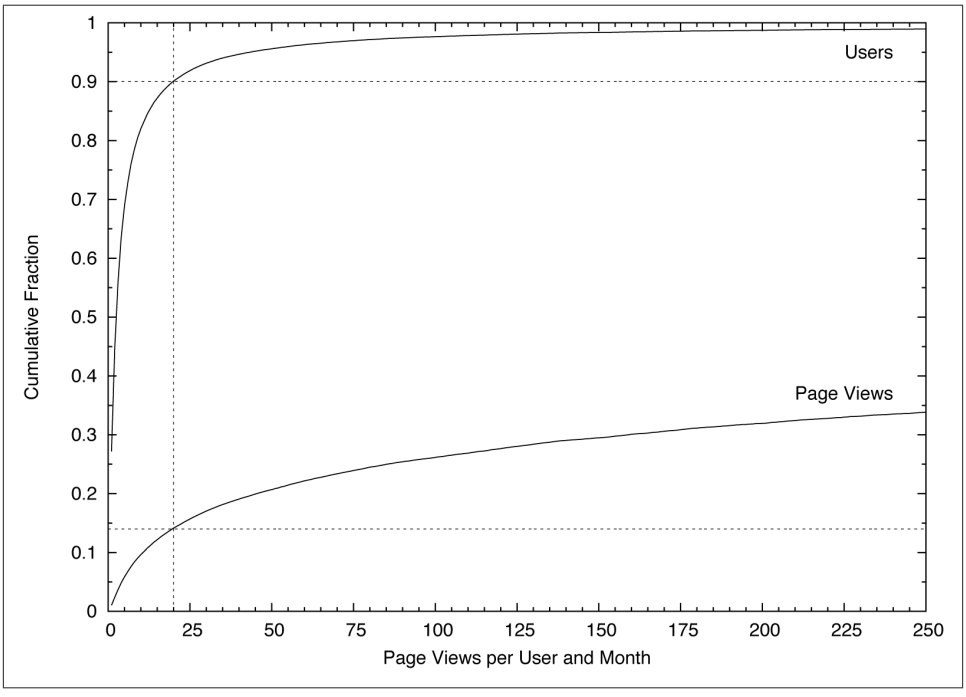


Figure 7-5. Cumulative distribution function for the data from Figure 7-4. Notice the reduced scale of the horizontal axis.

Figure 7-5 suggests a way to deal with such data. The graph shows a cumulative distribution plot, that is, the cumulative fraction of people and page views, attributable to visitors having consumed fewer than *x* pages per month. As we can see, the bottom 90% of visitors made fewer than 20 visits, and generated less than 15% of page views. On the other hand, the top 1% of users made more than 250 visits each, and together accounted

for more than 60% of page views. The graph suggests therefore to partition the population into three groups, each of which is in itself either relatively homogeneous (the bottom 90% and the middle 9%) or so small that it can almost be treated on an individual basis (the top 1%, or an even smaller set of extremely high-frequency users).

Datasets exhibiting power-law distributions come close to being "bad data": datasets for which standard methods silently fail and that need to be treated carefully on a case-by-case basis. On the other hand, once properly diagnosed, such datasets become manageable and even offer real opportunities. For instance, we can go tell the account manager that he or she doesn't have to worry about all of the 20,000 accounts individually, but instead can focus on the top 150 and *still* capture 85% of expected revenue!

Lessons Learned

What can we make of these disparate stories? Let's recap: the manufacturer's defect reduction scheme ran into trouble because they had failed to verify that the quality of the data lived up to their expectations. In the phone traffic study, the unexpected disagreement between the data and a theoretical model led to the discovery of additional structure and information in the data that would otherwise have gone unnoticed. And the third (and more generic) example points to a common failure mode in real-world situations where the most basic summary statistics (mean or median) fail to give a realistic representation of the true behavior.

What's common in all these scenarios is that it was not the *data* that was the problem. The problem was the discrepancy between the data and our ideas about what the data *should* be like. More clearly: it's not so much the data that is "bad," but our poor assumptions that make it so. However, as the second story shows, if we become aware of the disagreement between the actual data and our expectations, this discrepancy can lead to a form of "creative tension," which brings with it the opportunity for additional insights.

In my experience, failure to verify basic assumptions about the data (in regards to quality and availability, point distribution, and fundamental properties) is *the most common mistake* being made in data-oriented projects. I think three factors contribute to this phenomenon. One is wishful thinking: "Oh, it's all going to work just fine." Another is the absence of glory: verifying all assumptions requires solid, careful, often tedious work, without much opportunity to use interesting tools or exciting technologies.

But most importantly, I think many people are unaware of the importance of assumptions, in particular when it comes to the effect they have on subsequent calculations being performed on a dataset. *Every* statistical or computational method makes certain assumptions about its inputs—but I don't think most users are sufficiently aware of this fact (much less of the details regarding the applicable range of validity of each method). Moreover, it takes experience in a wide variety of situations to understand the various

ways in which datasets may be "bad"—"bad" in the sense of "failing to live up to expectations." A curious variant of this problem is the absence of formal education in "empirical methods." Nobody who has ever taken a hands-on, experimental "senior lab" class (as is a standard requirement in basically all physics, chemistry, biology, or engineering departments) will have quite the same naive confidence in the absolute validity of a dataset as someone whose only experience with data is "in a file" or "from the database." Statistical sampling and the various forms of bias that occur are another rich source of confusion, and one that not only requires a sharp and open mind to detect, but also lots of experience.

At the same time, making assumptions explicit can help to reduce basic mistakes and lead to new ideas. We should always ask ourselves what the data *should* do, given our knowledge about the underlying system, and then examine what the data actually *does* do. Trying to formulate such hypotheses (which are necessarily hypotheses about the system, not the data!) will lead to a deeper engagement and therefore to a better understanding of the problem. Being able to come up with good, meaningful hypotheses that lead to fruitful analyses takes a certain amount of inspiration and intellectual courage. One must be willing to stretch one's mind in order to acquire sufficient familiarity with background information about the specific business domain and about models and theories that might apply (the Poisson distribution, and the conditions under which it applies, was an example of such a "background" theory). If those hypotheses can be verified against the data, they lend additional credibility not only to the theory, but also to the data itself. (For instance, silly mistakes in data extraction routines often become apparent because the data, once extracted, violates some invariant that we know it must fulfill—provided we check.) More interestingly, if the data does *not* fit the hypothesis, this provides a hint for additional, deeper analysis—possibly to the point that we extend the range of our attention beyond the dataset itself to the entire system. (One might even get an exciting trip to a manufacturing plant out of it, as we have seen.)

Will This Be on the Test?

One may ask whether such activities should be considered part of a data scientist's job. Paraphrasing Martin Fowler: Only if you want your work to be relevant!

A scientist who is worth his salt is not there to crunch data (that would be a lab technician's job). A scientist is there to increase understanding, and that always must mean: understanding of the *system* that is the subject of the study, not just the data. Data and data analysis are merely means to an end, not ends in themselves. Instead, one has to think about the underlying system and how it works in order to come up with some hypothesis that can be verified or falsified. Only in this way can we develop deeper insights, beyond merely phenomenological descriptions.

Moreover, it is very much the scientist's responsibility to question the validity of one's own work from end to end. (Who else would even be qualified to do this?) And doing

so includes evaluating the validity and suitability of the dataset *itself*: where it came from, how it was gathered, whether it has possibly been contaminated. As the first and second examples show, a scientist can spot faulty experimental setups, because of his or her ability to test the data for internal consistency and for agreement with known theories, and thereby prevent wrong conclusions and faulty analyses. What possibly could be more important to a scientist? And if that means taking a trip to the factory, I'll be glad to go.

CHAPTER 8

Blood, Sweat, and Urine

Richard Cotton

A Very Nerdy Body Swap Comedy

I spent six years working in the statistical modeling team at the UK's Health and Safety Laboratory.[1] A large part of my job was working with the laboratory's chemists, looking at occupational exposure to various nasty substances to see if an industry was adhering to safe limits. The laboratory gets sent tens of thousands of blood and urine samples each year (and sometimes more exotic fluids like sweat or saliva), and has its own team of occupational hygienists who visit companies and collect yet more samples.

The sample collection process is known as "biological monitoring." This is because when the occupational hygienists get home and their partners ask "How was your day?," "I've been biological monitoring, darling" is more respectable to say than "I spent all day getting welders to wee into a vial."

In 2010, I was lucky enough to be given a job swap with James, one of the chemists. James's parlour trick is that, after running many thousands of samples, he can tell the level of creatinine[2] in someone's urine with uncanny accuracy, just by looking at it. This skill was only revealed to me after we'd spent an hour playing "guess the creatinine level" and James had suggested that "we make it more interesting." I'd lost two packets of fig rolls before I twigged that I was onto a loser.[3]

1. *http://www.hsl.gov.uk*
2. Creatinine is a breakdown of creatine phosphate, which is produced in your muscles. Dark yellow wee from dehydrated people will have higher concentrations of chemicals simply because it is less dilute, but it will also contain higher concentrations of creatinine. Dividing the concentration of a chemical by the concentration of creatinine adjusts for the state of your urine.
3. Translation of British idiom: I'd lost two packages of fig rolls on the wager before I realized I'd been had.

The principle of the job swap was that I would spend a week in the lab assisting with the experiments, and then James would come to my office to help out generating the statistics. In the process, we'd both learn about each other's working practices and find ways to make future projects more efficient.

In the laboratory, I learned how to pipette (harder than it looks), and about the methods used to ensure that the numbers spat out of the mass spectrometer[4] were correct. So as well as testing urine samples, within each experiment you need to test blanks (distilled water, used to clean out the pipes, and also to check that you are correctly measuring zero), calibrators (samples of a known concentration for calibrating the instrument[5]), and quality controllers (samples with a concentration in a known range, to make sure the calibration hasn't drifted). On top of this, each instrument needs regular maintaining and recalibrating to ensure its accuracy.

Just knowing that these things have to be done to get sensible answers out of the machinery was a small revelation. Before I'd gone into the job swap, I didn't really think about where my data came from; that was someone else's problem. From my point of view, if the numbers looked wrong (extreme outliers, or otherwise dubious values) they were a mistake; otherwise they were simply "right." Afterwards, my view is more nuanced. Now all the numbers look like, maybe not quite a guess, but certainly only an approximation of the truth. This measurement error is important to remember, though for health and safety purposes, there's a nice feature. Values can be out by an order of magnitude at the extreme low end for some tests, but we don't need to worry so much about that. It's the high exposures that cause health problems, and measurement error is much smaller at the top end.

How Chemists Make Up Numbers

Beyond the knowledge of how the experiments are set up, my biggest takeaway from the experience was learning about laboratory culture. Chemists (at least at the Health and Safety Laboratory) are huge on process. They have an endless list of documents and rules on good laboratory practice, how to conduct experiments, how to maintain the instruments, and how to choose your eleven o'clock snack. (Okay, the last one is made up, but there really are a lot of rules.) The formal adherence to all these rules was a huge culture shock to me. All the chemists are required to carry a lab book around, in which they have to record the details of how they conducted each experiment. And if they forget to write it down? Oops, the experiment is invalid. Run it again. I sometimes

4. For chemistry pedants, it's an inductively coupled plasma mass spectrometer, or ICP MS for short.

5. I also learned that you must never call a mass spectrometer a machine. Chemists are sensitive and become offended unless you call them instruments. I suspect it has something to do with wanting to be cool enough to be in a band.

wonder what would happen if the same principles were applied to data scientists. You didn't document this function. Delete. I can't determine the origin of this dataset. Delete. There's no reference for this algorithm. Delete, delete, delete. The outcry would be enormous, but I'm sure standards would improve.

On top of carrying around a lab book, numbers are peer-reviewed at each stage. Every number that comes out of an instrument is looked at by at least two people. Every number that goes into a database is also looked at by at least two people. This is a really good bad idea. It's really good because it exemplifies the chemists' commitment to generating accurate numbers, and peer-review is a great quality control technique. It's a bad idea for two reasons, which you may have spotted already.

"Every number gets looked at by two people?," you may be saying. "Does that really scale?" To which the short answer would be, "No, it doesn't." Peer-reviewing each number is hugely resource intensive, and a big expense. Because it's so time-consuming, one wonders how thoroughly it gets done when people are busy.

Secondly, "Every number gets looked at by *people*?" My background is in mathematics, so I love numbers more than most people. I even own a t-shirt with a maths joke on it. [6] Staring at thousands of numbers to look for mistakes, though, is not my idea of fun. For a start, I'm just not good at it. With five numbers, maybe even ten, I can quickly spot an outlier. Give me a thousand numbers, on the other hand, and I'm lost. I can probably spot a number that's one thousand times too big, just because the number is longer when printed (unless we use scientific formatting, of course). Beyond that, I'm wasting my time.

Here's a free business idea for you, based on a project idea that I've never gotten around to doing. Create software that accepts data generated by a mass spectrometer, and check that blank samples have a zero value, quality controllers are within range, and real samples have a sensible value. You can even throw in a few plots to visually display outliers and check for calibration drift. Once you've figured out how to connect to the mass spectrometer, the rest is easy, and laboratories around the world will sing your praises.

Human-based peer-reviewing wasn't the only problem I encountered. Once the data had been generated by the machines, although the formal adherence to process remained, the processes themselves were a little muddled. In a way, this is to be expected: chemists are naturally better at devising chemistry workflows than data handling workflows.

6. i says to π "Be rational." π replies "Get real!"

First, the data from each experiment is stored in Excel spreadsheets for the aforementioned peer-review process. I'll come back to the problems of spreadsheets later on, but as a storage medium for rectangular data (as produced by a mass spectrometer experiment), spreadsheets aren't too bad. The immediate problem is that to store the results of lots of experiments, spreadsheets won't cut it; they need to be stored in a database.

All Your Database Are Belong to Us

As a teaching tool for "how not to store data," the chemists' database was the gift that kept on giving. The database was created sometime in the pre-Cambrian period and had, over time, evolved into a hippopotamus: big, lumbering, and oddly prehistoric.[7]

The phrase "the chemists' database" betrays the first problem. The database was designed by chemists, maintained by chemists, and is "owned" by chemists. This "ownership" of data is a perennial problem, and one that I don't quite understand. On a corporate level, if a dataset has commercial value, then owning it makes sense. This is fine, but in my experience the sense of ownership goes way beyond corporate practicalities. It is really common for experimentalists to become personally attached to data that they create. To a lesser extent, this is true for data analysts as well. Flick back to the first page — even I subconsciously used the phrase "my data." While it's wonderful that people get passionate about data, too often it can lead to awkward office politics regarding who gets access to which dataset.

Furthermore, allowing individuals to control datasets creates data silos and incompatibility across your organization. Different teams will store data in different ways, and probably aren't aware of what data is available. This is a recipe for, at best, inefficiency, and at worst, lost datasets.

Though I've not experienced it in practice, I'm reasonably sure that a better system is for an organization to have a central data management team who controls access to the various datasets. The centralization encourages consistency of data structures (or at least storage technologies), and guarantees that your data is looked after by someone who knows about databases. A good analogy is with hard drive backups. If you let individual teams organize their own backups, then some teams will do it well, but others won't, and eventually you will lose data. On the other hand, if you outsource responsibility for the task to your IT support team, then failure will become rarer and you free up the rest of your staff to concentrate on their real job.

Returning to the chemists, their first problem with the database was how to get their data into it. Importing data from a spreadsheet into a database is automatable with a little effort, but this hadn't been done for all the types of datasets that they created. So

7. My friend who has been on safari suggests that this is unfair to hippopotamuses. They are actually rather agile, especially when running towards you.

for some experiments, the results had to be manually typed into the database. This should immediately set off warning bells in your mind. I can just about type my name correctly, most of the time. Give me a thousand numbers to type, and I'm going to make mistakes. Let me rephrase that. I'm going to make *a lot* of mistakes.

I think a good rule of thumb is that if you're having to type data, and you haven't been transported back in time, then you're doing it wrong. Again, this is the sort of problem that can easily be solved with a little thought about data management and workflow. Getting the data from the mass spectrometer, or whatever instrument you are using, into a database should be as near to automatic as possible.

In fact, automation should be a key goal of any workflow. Uwe's Maxim (named for statistician and R-core member Uwe Ligges) states

> Computers are cheap, and thinking hurts.

The more technology advances, and the older I get, the more I believe the maxim. When I think about the kind of science that I'd really like to be doing, the image consists of me lying on a beach, saying to my phone "Computer, tell me how many workers are at risk from occupational exposure to chromium VI." Then I sip a cocktail while the data is analyzed. I realize that this kind of Star Trek computing is a ways off, but I think that it's important to remember that messing about with data is not the end goal; what you really want to do is answer questions, and the more you can automate, the less burden there is on you to do that.

The structure of the chemists' database also created problems. As I previously mentioned, the database had evolved over time, in response to changing demands upon its use. This led to some quirks and inconsistencies in how things were stored.

Storing a number ought to be easy. You simply label the field as numeric, import the data, and ta-da! Everything should be fine. But what about special numbers? How do you store infinity, or ensure numbers are greater than zero? Can you distinguish not-a-numbers (NaNs) from missing values? With urine samples, there is an additional consideration. If the mass spectrometer records a zero result, you can't be sure that there really is none of the chemical that you were looking for in the sample. All you know is that the concentration is below some limit of detection (LOD). If the limit of detection is 5nmol/l, then the most natural way to write the value of a nondetect is <5nmol/l. Unfortunately, you can't store that value in a numeric field in a database. You need to split it into the numeric value of the LOD and a logical value denoting that the value wasn't detected. This sort of technical challenge is easy enough to overcome, provided that you think of it when you are designing the database. Leave it too late, and your users (that is, the chemists entering data) will have to hack a way of storing the nondetection information, and it won't be pretty. If you force concentrations to be stored as a plain number, with nowhere to store an optional "less than," the users have several choices. Naive users who aren't familiar with limits of detection will likely enter a zero

concentration. More educated users may enter the limit of detection itself. A third, smart-arse subset of users will try to be clever and enter half the limit of detection. In the worst case, the database will contain a mixture of the three types of value without any indication of which was which. At that point, you'll start wondering if your local exorcist can do databases. Unpicking such a mess can easily take longer than analyzing the dataset.

The solution to this needs to happen on three levels. The back-end data storage, as previously mentioned, needs to store a numeric concentration and a logical detection status. The user interface needs to check that the concentrations being entered are positive, finite numbers. In particular, disallowing zero values removes one potential user error, but this check also need to be made at the design stage. If you make the change later on, what do you do with all the zeroes that unwitting users already included? By that time, the LOD may be long forgotten, or never calculated, and removing rows of data will bias your statistics. (Measures of location, like the mean, will increase if you simply remove the low-end nondetects.)

User education is also important. It isn't intuitively obvious that when an instrument returns a zero value, you shouldn't enter a zero into the database. Making sure that chemists know some appropriate chemistry may sound obvious, but the point generalizes. If people entering data into a database don't understand that data, they will make mistakes. More importantly, technical fixes like checking user input can only take you so far—you shouldn't assume that technology will solve all your problems.

Deciding which pieces of information to store can also be problematic if you store the results of different experiments together. From a database designer's point of view, the simplest solution to storing experiment-specific results is to include a free-text field for miscellaneous information. When looking at occupational exposure to chemicals, a reasonably common experiment is to take a urine sample before people start work, and another one at the end of the day, then compare the two values to see if there is an increase throughout the work shift. The best solution to storing this information is to have a field in the database with the choices "pre-shift," "post-shift," or "not applicable." Maybe "unknown" as well for good measure.

By contrast, the free-text field solution will lead to a mess. Even if you are really strict with the people who enter the data and have idiot-proof guidelines, you will still end up with some records containing "pre" and "post," and some records containing "before" and "after," and some records containing the time of day of sample (in a variety of time formats). With a black belt in regular expressions, you may be able to unpick the mess, but it's far less neat than a dedicated field.

Using a dedicated field rather than free text does two things. It creates extra structure, and restricts the possibilities for input. Both these things allow you create additional

checks that your data is correct, which I'd argue are good things to strive for with data storage in general. (Of course, restricting possibilities for input can only be taken so far. If you don't allow users to store valid data, then you are creating problems for yourself rather than solving them.)

I realize that I've sounded harsh on the Health and Safety Laboratory and their data storage practices, but the truth is this sort of problem is so widespread that it is the norm rather than the exception. Data management is a genuinely hard problem because it mixes technical challenges with user education challenges, and throws in office politics for good measure.

Check, Please

The idea of checking input has been enshrined in good programming practise for as long as programming has been around, most notably in the maxim.[8]

> Fail early, fail often.

While this concept has its roots in software development, it is also applicable to data science. The "fail early" part means that checking the integrity of your data should be more or less the first thing you do. There is no point in waiting until you've written your conclusions to see if the dataset was nonsense. In fact, these checks should happen throughout the lifetime of the data, from the moment that they are generated. Bad data points should be identified, if possible, when they are generated by an instrument (or recorded in a survey, or otherwise created). They should be checked as they are stored in a database, and they should be checked again before they are analyzed.

The "fail often" part means that you need lots of checks. This is especially true if you are analyzing the data using a scripting language. The extreme flexibility of modern scripting languages is a wonderful thing, but it also provides you with more rope to hang yourself with in terms of allowing dubious data to run amok.

Features like dynamic typing mean that unless you explicitly check the type of your variable, the values that you thought were numbers might actually be strings, leading to bugs and confusion. On top of this, most scripting languages are permissive with variable sizes, so you need to watch out for vectors or arrays where you meant to have a scalar value. In fact, the number of possible checks can be fairly extensive.[9]I've taken to keeping a list of basic checks handy, in order to remember them all. Here's a shortened list.

8. The origin of the phrase is unclear, but it was famously the mantra of the engineering department in the Franklin Institute's Science Leadership academy.

9. R users can try my `assertive` package, which contains over one hundred (and counting) of the most common checks that you might want.

- Is your variable the correct type? (Integer, floating point, character, etc.)
- Does your variable have the correct dimensions?
- Is your number (in)finite/not a not-a-number/real/imaginary/positive/negative/in range?
- Is your string null or empty?
- Is your phone number, postal code, credit card number, or other special data type in the correct format?
- Does your file exist, and can you access it?
- Have you passed the right number of arguments to your function?
- What (version of the) software are you running?
- What units do your variables have?

The last check prevents a problem that is serious, but very hard to detect. In my younger and stupider days I fell victim to this when comparing variations between people in concentrations of a range of chemicals. When I collected the data together, I completely missed the fact that some concentrations were given in µmol/mol, and others were given nmol/mol. Several of the results that I excitedly reported to people were out by a factor of one thousand. Though it was very embarrassing at the time, I was lucky that a chemist spotted my mistake. Their experience correctly picked up that my numbers were nonsensical. The fix I've used to prevent this problem from happening again is to include units into variable names. Writing `concentration_nmol_per_mol` requires a little more typing than simply `concentration`, but it's infinitely preferable to producing incorrect results.

Live Fast, Die Young, and Leave a Good-Looking Corpse Code Repository

Including units in variable names is part of a more general idea that has helped me transform from a mathematician-who-writes-code to a "proper" programmer. That general idea is to use a style guide. Code that looks pretty is code you can read again more easily when you come back to it in six months time, and is code that you can give to others without a sense of shame. Sure, the substance is important too, but don't underestimate the importance of good-looking code.[10] In the same way that increased structure and restrictions on input are good ideas for data storage, they are also good ideas for the code you write.

Most style guides will specify how you should name your variables and functions (usually either `camelCase` or `lower_under_case`, with nouns for variables and verbs for

10. See also "brogramming," which involves you looking good while writing code.

functions), how spacing should be used (do you want `x + y` or `x+y`, and should the opening parenthesis be on the same line as the function declaration or the next?), and the order of contents in your code files. The specific rules will vary from style guide to style guide, and you may wonder which one you should pick. Don't worry. In the same way that it doesn't really matter how wide apart the rails are on train tracks—only that there is a standard to which all tracks conform—the fact that your code conforms to a style guide is more important than which one it conforms to.[11] (The same is true of version control software. Picking *anything* is a vast improvement over using nothing.)

Having said that, an important element of style is what you call your functions and their arguments. There's a fun game I learned called "Black Box" for testing code maintainability. It's really simple and very informative. Take a copy of your code and strip out everything except the function names and signatures.[12] (If you are working in an object-oriented language, you can keep the class names and signatures, too.) Pass your code to another programmer and have them guess the contents of the function. They don't need to give too much detail, just the general idea. As an example, if you give them a function called `mean`, which takes an argument `x`, they might say, "This takes a numeric input and calculates the arithmetic mean." Easy, right? The real beauty of this game is when you come across badly named functions. Will your friend know what `foo` does? How about the function you called `analyze`? What about the sensibly named function that grew and now takes 37 arguments? Or the one that you generalized, and now does six different things, depending upon the context?

Even if you code in a cabin in the woods with no connection to the outside world, Black Box is still a good game to play on your own. Pretend that you didn't write the contents of the functions, and look against at what you named them. Does anything make sense, six months down the line? If you gave your code to someone else, would they understand it or just snigger?

Rehab for Chemists (and Other Spreadsheet Abusers)

Returning to my job swap with James, after he'd shown me how strict the chemists are with their experimental processes, I was keen to demonstrate the data analyst's equivalent. The one point that I wanted to get across to James was that we need to be able to show where any statistics come from, providing a trail from raw data to results.

11. If you do want to pick one, I provide style guides for R and MATLAB code on my blog, at *http://4dpiecharts.com*.

12. R users can do this with my `sig` package.

As I mentioned earlier, the chemists are big fans (or at least routine users) of Excel spreadsheets. Most of them have taken an introductory course or two in statistics at some point. This means that, without intervention from a statistician, they'll have a go at their own data analysis, which consists of pointing and clicking and calculating a few means, then running T-tests on everything that they can think of.

There are three big problems with using spreadsheets for data analysis. Firstly, the numerical routines in all the major spreadsheets are currently surprisingly poor.[13] Secondly, they contain some semantics design flaws that make it easy to get muddled. For example, they don't properly distinguish data from analysis. Whereas more or less every programming language will contain variables for data and functions for analysis, in a spreadsheet, a cell can contain either a formula or data. In fact, it is worse than this, because you only get a representation on data. So a cell containing `1.23` could contain a number or some text, and you have to look closely to see the difference. The other semantic flaw is that spreadsheets don't easily deal with data that aren't rectangular, or don't have a location.

Unfortunately, experience tells me that neither of these arguments works very well at dissuading chemists from using spreadsheets. For most chemical health and safety purposes, the answer only needs to be right to the nearest order of magnitude, so poor numerical routines tend to be disregarded as the least of their worries. Depending upon your field of expertise, this argument may have more importance. Arguing about semantic flaws mostly just induces a glazed expression.

So in order to wean James off his spreadsheet addiction,[14] I chose to use the third argument: datasets always change halfway through your analysis, and your analysis will likely change several times as well, and editing code is quicker than having to point and click at the same thing over and over again.[15]

Having talked James through importing our urine sample dataset, calculating some statistics, and drawing some plots, I miraculously produced some extra samples that we'd "forgotten about." His initial disappointment changed to surprise at the fact that we could re-run the analysis with a simple key-press. Sometimes, technology just works.

All in all, trading places with the person creating my data was a useful experience. If you work in a multidisciplinary environment, then this sort of activity is something I'd

13. See Almiron et al, 2010. *http://www.jstatsoft.org/v34/i04/paper*.

14. As coined by Pat Burns. *http://www.burns-stat.com/pages/Tutor/spreadsheet_addiction.html*.

15. Technically, you can program things in spreadsheets, but that usually involves writing VBA code, and nobody wants that.

heartily recommend. Keep leaving copies of body-swap comedies on your boss's desk until she agrees to let you take part. (No one can cope with all of *Trading Places*, *Freaky Friday*, **and** *17 Again*!) If I get the chance again, I'd love to try and teach the person about reproducible research, but that's another story.

tl;dr

For those of you in a hurry, here's the distilled version.

- Working with the people who create or collect your data can be really informative. Try and organize a job swap or work shadowing.
- Laziness is a virtue. Try to automate as much of your job as possible.
- Having a central data management team can avoid data getting lost in the fringes of your organization, and help standardize formats across your organization.
- Write code that fails (early and often). The more checks you include in your code, the merrier.
- Stylish code is easier to maintain. Don't underestimate the importance of writing code thats looks good.
- Spreadsheets are easily abused. Use them sparingly.

CHAPTER 9

When Data and Reality Don't Match

Spencer Burns

It is common knowledge that beating the stock market is hard. But on the face of things, it seems like purely an analysis problem, not a data problem. Modeling is difficult; building a timeseries should be simple. For every day (or minute, or millisecond), for each unique stock symbol, there is a listed price at which you can buy shares, and you can sell those same shares later, comparing the two prices to calculate your profit.

Every assumption in the preceding statement is usually true, yet each of them fail often enough to ruin a model. A series of stock data may look structured and clean, but that neatness hides the idiosyncratic path of how a given stock got to where it is today. It is "good" data covering up for messy reality.

Stock data does not come out of independent observations of markets; the data is an integral part of the market. There is a tight feedback loop where data about the state of the market affects the market (e.g., rising prices may cause people to push prices up further). Furthermore, what happens in the market can change the nature of the market. Some examples of this are when companies with falling prices leave the market, or ones with rising prices buy other companies, or when market crashes trigger changes in trading rules.

What this means in practice is that the stock market of today is not exactly the same as the market of yesterday; every day is a new experiment. Data needs to be constantly adjusted in small ways to make yesterday comparable to today.

Magnifying this problem, most trading strategies are only just barely profitable (or unprofitable) on average, adding up a large number of small transactions to get significant results. Even tiny systematic errors can compound into grossly misleading results in analysis. The available data might represent reality closely, but the small mismatches matter.

Below are three examples of how the picture painted by the stock prices does not match the historical reality. These are not obscure domain-specific details, but would affect anyone looking at prices on common data sources like Yahoo! or Google Finance.

Whose Ticker Is It Anyway?

Imagine that at some point in the past, you wanted to know what the identifying ticker symbols were for Kmart, Sears, and Sprint.

Table 9-1. Stock symbols for Kmart, Sears, and Sprint over time

Start	End	Kmart	Sears	Sprint
20th Century	2002-12-18	KM	S	FON
2002-12-19	2003-06-09		S	FON
2003-06-10	2005-03-27	KMRT	S	FON
2005-03-28	2005-08-14		SHLD	FON
2005-08-15	Present		SHLD	S

For a long time, they were dependably KM, S, and FON, all listed on the New York Stock Exchange (NYSE).

Then in 2002, a bankrupt Kmart was delisted from the NYSE leaving just S and FON.

Kmart, having survived bankruptcy, was relisted in 2003 with the symbol of KMRT on NASDAQ. Technically, this was not the same stock that KM had been. It was the same company in any logical sense, with the same stores and branding. But it was an entirely new legal entity with fresh shares of stock.

In March 2005, Sears (S) and Kmart (KMRT) disappeared and "Sears Holdings" with a new NASDAQ listing of SHLD started trading. Sears and Kmart had merged. Without further information, the obvious conclusion would be that Sears had changed their symbol and company name at the time of a merger, which is not uncommon (e.g., when Hewlet Packard bought Compaq, it changed from HWP to HPQ).

Actually, the merger ran the other way, and Kmart bought Sears. This seems a trivial distinction, but a share of KMRT became a share of SHLD with no changes except the name. All shares of S were delisted, and the owners of those shares were given both cash and newly issued shares of SHLD.[1]

1. This does not address how to create a timeseries for an acquired company. The correct answer of having it become partially shares of the acquirer and partially cash is difficult to implement and requires that you know the terms of the acquisition. The lazy answer, which will give mostly correct results, is to simulate that you sold the stock for the market price on the end of the last trading day.

With Sears delisted, one of the great "blue chip" stocks listed on the markets since 1906 was gone. This meant that the coveted single-letter symbol of "S" was up for grabs. Sprint called dibs, and five months later they changed their ticker from FON to S when they completed a merger with Nextel (NXTL).

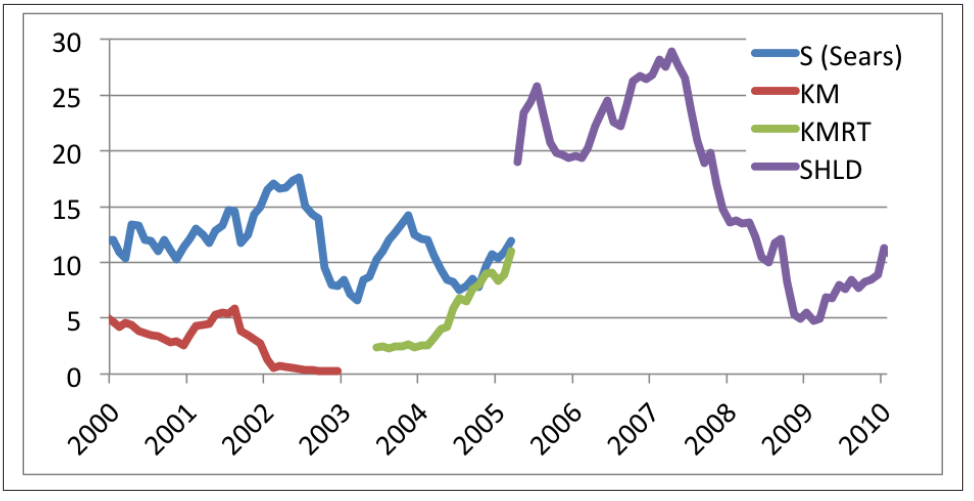


Figure 9-1. Total market value (billions of dollars) of stock for Sears and Kmart, 2000-2010

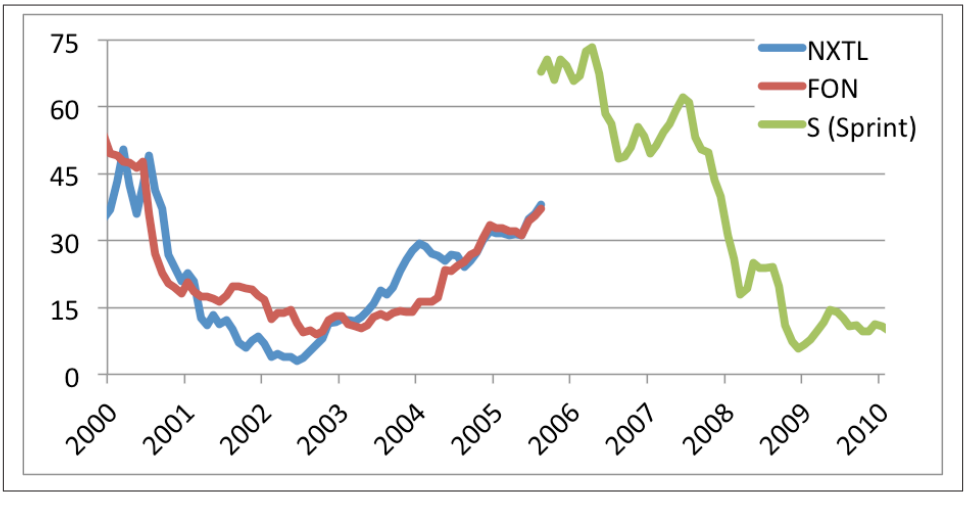


Figure 9-2. Total market value (billions of dollars) of stock for Sprint and Nextel, 2000-2010

In short, if you are looking at Sears Holdings today and ask what its symbol was in 2004, the answer is KMRT, not S. And if you ask what its ticker was in 2001, the answer is that it didn't exist yet. But if you want to look up the price for S at some point in the past, then you had better make sure you know whether you are looking at Sears or Sprint.

Most people working with equity data, even within the financial industry, want to use the ticker symbol as a unique identifier. They were originally created for that purpose, and at any given point in time they are unique. But once you start building timeseries to look at data from the past, they will trip you up.

The way that most data sources, such as Google or Yahoo! Finance, deal with this is to just provide the most recent symbol as the unique key. This means that whenever you reload fresh data, some of your identifiers will have changed.

More importantly, any company that is no longer traded does not have a current symbol. In most current datasets, if you look up "S" you get Sprint; if you look up "Sears" you get SHLD; and if you look up Kmart, you get nothing. This causes a major "survivorship bias" in data: the stock market looks much more profitable if you never look at companies that have gone bankrupt or been bought out.

There are universal unique identifiers with acronyms such as CUSIP, ISIN, and SEDOL. But these are proprietary and often not available. They also only solve half the problem: they do not get recycled the way ticker symbols do, but they will change over time as minor changes happen to a stock (e.g., the CUSIP changed when KMRT became SHLD).

The standard solution is to purchase data from a vendor that tracks all of these changes and provides unique identifiers. While convenient, it can be expensive and sticks you with vendor lock-in. It is also often the case that the vendor data has a high error rate in tracking these changes.

A more robust solution is to maintain your own mapping of symbols and unique identifiers. This involves a fair amount of hard work to build and maintain the dataset, and still requires getting mapping information from a vendor.

There is a limitless supply of other kinds of symbol and metadata chaos as one gets deeper into multiclass stocks, other asset classes, "over the counter" trading, stock reissues, foreign exchanges, and so on. Ticker changes are just the biggest and most common headache.

Splits, Dividends, and Rescaling

The prices of a stock between one day and the next are not always directly comparable. Prices can get rescaled or cash may get returned to stockholders when certain kinds of "corporate actions" occur.

The simplest kind of price level adjustment is a stock split. If a successful company's shares become expensive, they might declare a two-for-one (2:1) stock split and give out two new shares for every old one. Each such new share is half the price of the old share. A stock split can be for any ratio—3:2 or 3:1 are also common. A failing company might have a "reverse" split to push up its nominal price, sometimes 1:100.

Stock splits provide a valuable service with this sort of price renormalization. One company that has steadfastly refused to split is Berkshire Hathaway. At the time of this writing, the price of a share "Class A" stock (BRK.A) is $120,000. Not only does this make the stock difficult to purchase, but having to represent eight digits (including cents) wreaks havoc with display fields in visualizations and applications.

Many sources of stock data, such as Google or Yahoo! Finance, provide prices that have already been split-adjusted so that users can compare current to historic prices without making their own calculations. A problem with such adjusted data is that the entire price history of a company needs to be readjusted when a new split occurs; this makes maintaining and comparing datasets tricky. It also is often important to know what the actual price was in the past to understand how it would have been traded.

In general, splits are easy to deal with. The data is commonly available along with stock prices, and they will usually occur only once every few years. In all cases, the adjustment is a simple multiplication or division (technically, one gets cash for any fractional shares, but that can be ignored). But if you need to be able to have both corrected data for comparisons and uncorrected data for simulations, then multiple timeseries must be maintained. Often, the best solution is to use the original, uncorrected data as the main series, and also keep a series of adjustment factors around to adjust prices on the fly, as needed.

Dividends, the other common type of corporate action, are trickier to adjust for. When a company "declares a dividend," they are stating that anybody who holds a share of stock on a given date (the "ex-date") will be given a cash payment on another date a week or so later. In the meantime, the price of the stock will drop by approximately the amount of the dividend on the ex-date: it is worth that much less because any new owner of the stock would not receive that dividend.

Table 9-2. Microsoft splits, dividends, and adjustments

Ex-Date	Split Ratio	Dividend Value	Market Price	Split Adjustment	Dividend Adjustment	Adjusted Price
1996-01-02			89.75	1	1.0000	89.75
1996-12-09	2:1		81.75	2	1.0000	163.50
1998-02-23	2:1		81.63	4	1.0000	326.50
1999-03-29	2:1		92.38	8	1.0000	739.00
2003-02-18	2:1		24.96	16	1.0000	399.36
2003-02-19		0.08	24.53	16	1.0033	393.76

Ex-Date	Split Ratio	Dividend Value	Market Price	Split Adjustment	Dividend Adjustment	Adjusted Price
2003-10-15		0.16	29.07	16	1.0088	469.21
2004-08-23		0.08	27.24	16	1.0117	440.96
2004-11-15		3.08	27.39	16	1.1255	493.25
2005-02-15		0.08	25.93	16	1.1290	468.39
2005-05-16		0.08	25.49	16	1.1325	461.89
. . .	. . .	. . .	. . .	. . .	. . .	. . .
2012-02-14		0.20	30.25	16	1.2877	623.24
2012-05-15		0.20	30.21	16	1.2962	626.54

In 2004, Microsoft (MSFT) relented to shareholder demands and declared a one-time dividend of $3.00 per share on top of their regular dividend of $0.08 per share. This $3.08 was 11% of the value of each share. On the ex-date, MSFT stock opened trading in the morning at a price that was $2.61 (8.7%) less per share than it had the day before. Anyone who ignored the dividend would think the stock was significantly down for the day. Correctly accounting for the dividend, it is clear that Microsoft had gone *up* in value.

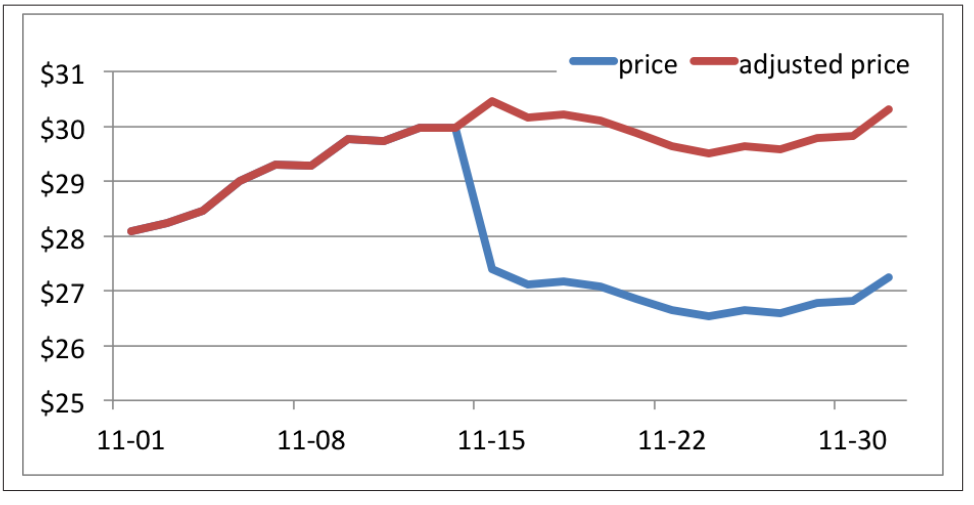


Figure 9-3. Microsoft (MSFT) stock price and dividend-adjusted stock price, November 2004

You cannot adjust for dividends by simply subtracting their value from historic prices—if the company had, in the past, been worth less than the current dividend, there would be negative prices. The "correct" way is to carefully account for the cash returned every time a dividend occurs; however, this kind of accounting makes it impossible to do timeseries analysis.

The best way to adjust the data is to pick the multiplicative factor that gives the same daily return as the correct accounting: (dividend) / (new price) - 1. This is the equivalent of taking the money from the dividend and immediately reinvesting it in the stock.

Unfortunately, even more so than for splits, there are many situations when you need to adjust for dividends and many where it is important not to do so. So it is important to track multiple timeseries, sometimes three of them: price, split adjustment, and dividend adjustment.

Bad Reality

Sometimes, "obviously" bad data is correct. Odd-looking data might be representing bad reality, whether from technical failure or irrational humans.

A great example of "accurately" bad data is a crash in United Airlines' stock (symbol at the time: UAUA, current symbol: UAL) that was triggered by Google News crawling and republishing a six-year-old story about United filing for bankruptcy.

On September 6th, 2008, Google News' spider came across an archived story in the *South Florida Sun Sentinel* about United's 2002 bankruptcy. The story had no date attached to it, so Google News tagged it with the current date by default. On September 8, an independent investment advisor saw the story, assumed it was current, and posted it to a subscription financial news feed. Traders saw the headline on the news feed and rushed to sell UAUA as fast as possible, not stopping to read the story. UAUA plunged from $12 a share to $3 in less than five minutes. A few minutes later, trading in UAUA was suspended for an hour to let everybody breathe and figure out what had happened. After that, trading resumed as if nothing had ever happened.

Many data-cleaning algorithms would throw out these few minutes of obviously wrong prices. Yet, these prices did happen; real money was exchanged. Within ten minutes, 15% of UAUA's shares and $156 million had changed hands. This is correct data that should be kept for any honest analysis.

Other times, the answer is not so clear cut. In the May 6th, 2010, "flash crash," many stocks suddenly dropped in price by more than half for no readily apparent reason before going back again when sanity prevailed. There is still no consensus on exactly what happened. After the crash had been resolved, the various stock exchanges retroactively canceled all trades that were more than 60% different from the pre-crash price. This caused an odd situation where people who got a really good bargain buying shares for half the correct price got to keep their gains, while those who had an even better deal at 60% off had to give the profits back.

So, what is the "reality" of the price data after the trades were canceled? Those trades officially never happened, and in the end no money changed hands. Most stock data sources have already had the canceled trades removed, so it is difficult to even see the

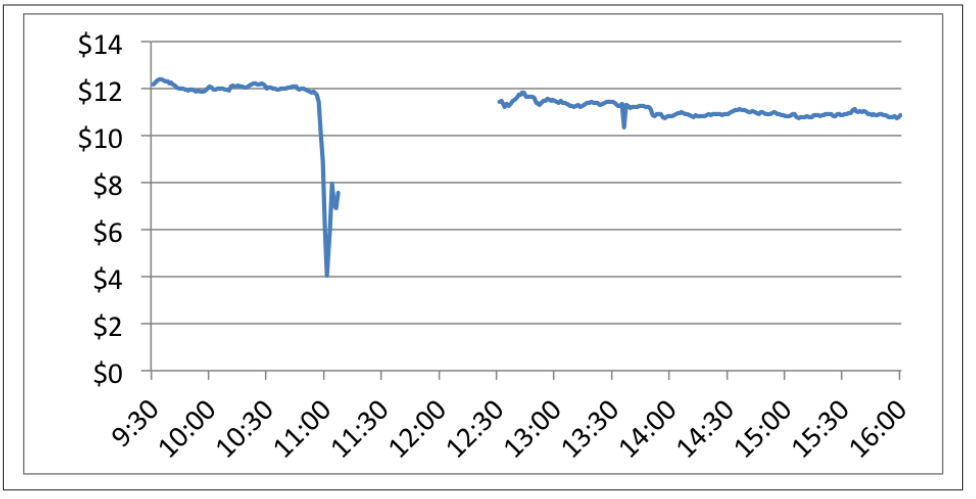


Figure 9-4. United Airlines (UAUA) Stock Price on 2008-09-08, Volume-Weighted Average over One-Minute Intervals

evidence of the flash crash in the historic record. In that sense, the flash crash contained bad data, not bad reality. Yet, the only difference between the flash crash and the United Airlines example is that human intervention undid the flash crash after it happened; it was "too big to fail."

To further cloud the issue, people believed that those trades were real when they happened. Stock trading is state-dependent; how you behave depends on the previous trades that happened.

To properly model trading during the flash crash, you would have to simulate making trades, updating your portfolio, and then having them canceled afterwards. In reality, you would just shrug your shoulders and hope that it is okay to ignore this sort of crash. This is fundamentally a human event, not a mathematically modelable one.

So, what can be done to deal with bad reality? Situations like the United Airlines example are more or less just outlier detection; a bad price that persists for one second should probably be thrown out while one that keeps going for a few minutes probably should not. It is a matter of judgment to decide what kind of thresholds to set and what error rates are acceptable, but that is a tractable data problem.

But there is little that can be done to deal with situations like the flash crash, except to know that they happened and work around them. There is no systematic way to model extraordinary events which have been shaped by human judgment and politics.

Conclusion

The theme of these examples is that clean-looking data often has additional complexity lurking under the surface. If you do not understand the data, where it comes from, and what it represents, then your conclusions may carry a bias. Financial data is especially sensitive to this problem, but similar problems can occur in any type of data source.

This is not to say that you need to be a domain expert to tackle a given data problem, but you do need to spend time understanding the inputs. Often, data scientists devote most of their effort towards building models and looking at results. Cleaning data and making sure it is well-formed often delivers better results for the effort spent than doing more analysis. There is no use in providing more accurate answers until you have made sure you know exactly the question you are asking.

CHAPTER 10

Subtle Sources of Bias and Error

Jonathan A. Schwabish

Please note: The views expressed in this chapter are those of the author and should not be interpreted as those of the Congressional Budget Office.

Before we get started, let me be clear: I get to work with some of the best socioeconomic data in the world. I have access to data provided by the U.S. Social Security Administration (SSA), which provides information on earnings, government benefits (specifically, Social Security benefits), and earnings for a huge number of people over a large number of years. The data is provided to the government through workers' W-2 tax forms or other government records. By comparison, survey data is often collected from interviews between an interviewer and respondent, but may also be collected online or through computer interfaces in which there is no interaction between the interviewer and interviewee. Administrative data is becoming more widely available in many social science fields, and while that availability is enabling researchers to ask new and interesting questions, that data has also led to new questions about various sources of bias and error in survey data.

Administrative data has both advantages and disadvantages over publicly available survey data. The major advantage is that the administrative data tends to be more accurate than survey data because it is not subject to the typical errors found in survey data. Such errors include:

- Nonresponse (the respondent fails to answer a question)
- Recall error (the respondent incorrectly answers a question or cannot recall information and fails to answer)
- Proxy reporting (one person responds to the survey for another)
- Sample selection (the survey sample does not represent the population)

Furthermore, administrative data is typically not imputed (that is, the firm or institution conducting the survey has not inserted values for missing observations) or top- or bottomcoded (that is, variables at the tails of the distribution are not changed to protect the identity of respondents). Finally, administrative data may include a larger number of observations for longer time periods than may be available in survey data.

Administrative data also have several disadvantages, however. First, information is typically available on only a limited set of demographic variables. For example, educational attainment, marital status, and number of children are almost always useful in economic analysis, but are not available in the SSA files I use. In addition, this data does not include earnings that are not reported to SSA (for example, earnings from cash-based employment or acquired "under the table") or earnings from workers who do not have, or do not report, a valid Social Security number. Unreported earnings may be particularly important for research on, say, immigration policy because many immigrants, especially illegal immigrants, do not have — or have invalid — Social Security numbers.[1]

Let's put these differences in datasets in some context. A few years ago, two co-authors and I were interested in examining year-to-year changes in individual earnings and changes in household incomes—so-called earnings and income "volatility."[2] The question of whether people's (or households') earnings (or incomes) had grown more or less volatile between the 1980s and 2000s was a hot topic at the time (and, to some degree, still is) and with administrative data at our disposal, we were uniquely suited to weigh in on the issue.[3] To track patterns in earnings and income volatility over time, we calculated the percentage change in earnings/income using three variables:

1. Earnings/income from the survey data.
2. Earnings/income from the survey data, excluding observations for which the institution conducting the survey replaced the earnings or income variable with values from a similar person (so-called imputation).
3. Earnings/income from administrative data matched to survey data (using respondent's Social Security numbers).

On the basis of previous research, we suspected that the results from the three variables would be roughly the same and that we would find an increase in volatility over time. When we used the raw survey data, we found an increase in volatility roughly between the mid-1990s and the mid-2000s. However, when we dropped observations with im-

1. See, for example, Lubotsky (2007) and Schwabish (2011).
2. Perhaps the most technical presentation of our work was published in Dahl, DeLeire, and Schwabish (2011). Interested readers should also see Congressional Budget Office (2007a, 2007b).
3. I would be remiss if I didn't add citations to at least a few seminal works in this area: Gottschalk and Moffitt (1994), Moffitt and Gottschalk (1995), Haider (2001), and Shin and Solon (2009).

puted earnings or income, we found a slightly slower rate of growth. Then, when we matched the survey data to the administrative records and recalculated the percentage change using the administrative earnings, we found almost no change in volatility over that roughly 20-year period.

This apparently contradictory result made us keenly aware of two shortcomings in the survey data we were using, specifically imputation bias and reporting bias. The strategies you can use to overcome data shortcomings are somewhat limited, but being aware of their existence and taking as many precautionary steps as possible will make your research better and your results more reliable.

The rest of this chapter describes these two issues—imputations and reporting error—which you might encounter in working with your data. I also discuss four other potential sources of bias—topcoding, seam bias, proxy reporting, and sample selection. The overall message here is to be careful with your data, and to understand how it is collected and how the construction of the survey (or the firm or institution who conducted it) might introduce bias to your data. To address those sources of bias may require time and energy, but the payoff is certainly worth the effort.

Imputation Bias: General Issues

High rates of data imputation complicate the use of survey data. Imputation occurs when a survey respondent fails to answer a particular question and the firm or institution conducting the survey may replace values for that record. For example, while a respondent may be perfectly happy to give her name, age, sex, educational attainment, and race, she may be less willing to share how much she earned last year or how many hours she worked last year. There are a number of ways in which the firm or institution can impute missing data; the Census Bureau, for example, uses a variety of methods, the most common being a hot-deck imputation method. (Hot-deck imputation replaces missing values with the value for that same variable taken from a complete record for a similar person in the same dataset.[4]) Imputation rates in the Survey of Income and Program Participation (SIPP; a panel dataset conducted by the Census Bureau) have grown over time, which could further bias research that tracks patterns over multiple years. For example, in work with co-authors, we found that in an early panel of the SIPP, roughly 20% of households had imputed earnings data over a two-year period; in a separate panel about twenty years later, that percentage had risen to roughly 60%.[5]

Although imputing missing data can often result in improved statistics such as means and variances, the use of imputed data can be problematic when looking at changes in

4. Bureau of the Census (2001).

5. Dahl, DeLeire, and Schwabish (2011). In that study, we found that 21% of households had imputed earnings in 1985, 28% in 1991, 31% in 1992, 33% in 1993, 35% in 1994, 54% in 1998, 60% in 2002, and 46% in 2005.

certain variables. For example, in our volatility project, we were interested in seeing how people's earnings and incomes changed over time. Using observations that have been imputed in such an analysis is problematic because observed changes in earnings are not "real" in the sense that they are not calculated from differences in an individual's reported values between one year and another. (Datasets that follow the same person — or family or household — over time are called *longitudinal* or *panel* datasets.) Instead, the calculated change is the difference between actual reported earnings in one period and the imputed earnings in the other. Thus, under a scheme in which earnings are imputed on the basis of some sample average, what you are actually measuring is the deviation from that average, not the change in that individual's earnings from one period to the next. Alternatively, one could impute the change in the variable of interest, a strategy that might work well for data that has a small variance with few outliers. The exact method by which imputed values are constructed might result in estimates that more strongly reflect the mean or that otherwise introduce bias.

When first exploring a dataset, you should become familiar not only with the data structure and variables available but also with how the survey was conducted, who the respondents were, and the technical approaches that were used to construct the final data file. For example, what happens when people fail to answer a question? Do the interviewers try to ask follow-up questions or do they move quickly to the next question? Was the survey conducted using one-on-one interviews where the interviewer recorded responses, or were respondents asked to fill out a form on their own?

You can deal with imputed data in a number of ways. One can:

- Use the imputed data as it is (which is problematic for longitudinal analysis, as previously explained);
- Replace the potentially imputed data with administrative earnings records (which, for most researchers, is not a viable option);
- Reweight the data to better reflect the population using the sample of survey respondents for whom the variable(s) of interest were not imputed;
- Drop the imputed observations (which may bias the results because the resulting sample may no longer be representative).

Economists have approached imputed data in a number of ways. Most studies use imputed data when it is available because it requires less work sifting through the data and maintains the existing sample size. Others, however, drop observations with imputed data, especially those who use longitudinal data.[6]

6. See, for example, Bound and Krueger (1991); Bollinger (1998); and Ziliak, Hardy, and Bollinger (2011); Lillard, Smith, and Welch (1986); Hirsch and Schumacher (2004); and the discussion in Dahl, DeLeire, and Schwabish (2011).

For longitudinal analysis, the best course of action is probably to drop imputed observations unless an administrative data substitute is available. For cross-sectional analysis, you should tread carefully and test whether the inclusion of imputed observations affects your results. In all cases, you should at least be aware of any imputations used in the survey and how the imputation procedure may have introduced bias to those variables.

Reporting Errors: General Issues

Reporting errors occur when the person taking the survey reports incorrect information to the interviewer. Such responses are considered "mistakes" and could be accidental or intentional, but it is impossible to know. Because reporting errors are more difficult to identify, the strategies you can use to address the problem are more limited. In one study, for example, researchers *re*-interviewed a sample of people and showed that only about 66% of people classified as unemployed in the original interview were similarly classified in the re-interview.[7] Another study compared reports of educational attainment for people over an eight-month period and found that those respondents were most likely to make mistakes in reporting their educational attainment the first time they were surveyed, which suggests that people may become better at taking surveys over time.[8] Of course, in cases in which the survey is conducted only once, there is no chance for the survey respondent to become better at taking the survey and little chance for the researcher to assess the accuracy of responses.

When it comes to measuring earnings, there are two ways in which earnings might differ between survey and administrative data. In the first case, a person works a part-time job for two months, say, at a hardware store early in the year and earns $1,500. When interviewed in December of that year, she tells the interviewer that she had no earnings in the past year; in the administrative data, however, her record shows earnings of $1,500 because that amount was recorded on her W-2. In the second case, a person works full-time shoveling driveways for cash early in the year and earns $500. When interviewed in March, she tells the interviewer that she earned $1,000 shoveling driveways; in the administrative data, however, her record shows no earnings in that year because the shoveling job was paid in cash and no W-2 was filed. In this second case, both datasets are incorrect—earnings were misreported in the survey data and earnings were not reported to the government in the administrative data. These two cases illustrate how errors can occur in either (or both) types of data, and which dataset is better (that is, provides more accurate answers) depends on the question you're asking. If you're wor-

7. McGovern and Bushery (1999).

8. Feng (2004).

ried about capturing earnings in the underground economy, an administrative dataset is probably not the best source of information; however, if you're interested in calculating the Social Security benefits for a group of workers, errors in survey data might cause problems.

These problems are most likely to occur among people at the bottom of the earnings distribution.[9] Figure 10-1 shows the cumulative distribution of earnings in 2009 at $25 increments in the SSA administrative earnings data and the March Current Population Survey (CPS; a monthly survey of over 50,000 households conducted by the U.S. Census Bureau) for people earning less than $5,000 a year. Two observations are immediately evident: First, the cumulative shares of people in the administrative data in each $25 range are about 8 to 15 percentage points above the same point in the CPS, suggesting that the first case described above is probably more prevalent than the second (that is, people underreport their earnings in a survey).[10] The second observation is that respondents in the survey data tend to round reports of their earnings (see the circled spikes in the CPS at thousand-dollar increments), which, depending on the research question may not be a terribly large problem, though in some cases rounding can lead to different estimates.[11]

The challenge a researcher faces with reporting error is that, generally, it is not obvious that it exists. Without comparing one dataset directly with another (for example, merging administrative and survey data together), how can one know whether someone misreported his or her earnings or educational attainment or how many weeks he or she worked last year? Part of dealing with reporting error is digging deep into your data; for example, do you really think a person earned *exactly* $5,000 last month or worked *exactly* 2,000 hours last year (50 weeks times 40 hours)? Another part of dealing with reporting error is realizing the strengths and weaknesses of your research question. In some previous work, for example, I was interested in predicting rates of emigration among foreign-born workers (that is, the rate at which people leave the United States). In that case, I inferred emigration rates by following longitudinal earnings patterns over time using administrative data; although that was the strength of the analysis (and, to my knowledge, was the first attempt to use administrative data in that way), the weakness of such an approach is that I clearly missed foreign-born workers who were living and working in the country without authorization (that is, illegal immigrants) and thus may not have filed a W-2.[12]

9. They are also problematic at the top of the distribution, but the percentage differences are most likely smaller.
10. That finding is supported in, for example, Cristia and Schwabish (2007).
11. See Roemer (2000), Schwabish (2007), and Cristia and Schwabish (2007).
12. Schwabish (2011).

Although determining whether the dataset you are using is riddled with reporting errors (and whether those errors actually matter) is difficult, being aware of such data shortcomings will take your research further and, importantly, make the validity of your conclusions stronger. Thus, the basic strategy is to be aware and ask key questions:

1. Is reporting error in your data likely to not be random? If so, your results will be biased.
2. Do you think reporting error is more likely in some variables than in others?
3. Can you use other variables to inform sources of potential bias? For example, does everyone with 2,000 hours of annual work also have round earnings amounts? (Perhaps the survey only allows people to check a box of round values instead of asking for exact amounts.)
4. Does the structure of the survey itself change the probability of reporting error? For example, does the order in which the survey asks questions suggest that people are less likely to answer certain questions because of that order, or might people have some reason for giving less reliable answers at one point in the survey than at some other point?

In sum, reporting errors are difficult to detect, but it's important to be aware of both the strengths and weaknesses of your data and to try to adjust your model or research questions accordingly.

Other Sources of Bias

While imputation bias and reporting error may be two of the largest sources of bias in survey data, there are a number of other issues that can affect the validity and reliability of your data. In this section, I briefly discuss four other sources of bias:

Topcoding/bottomcoding
: To protect respondents' identities, some surveys will replace extremely high or extremely low values with a single value.

Seam bias
: Longitudinal (or panel) surveys in which changes in survey respondents' behavior across the "seam" between two interviews are different than the changes within a single interview.

Proxy reporting
: Surveys in which a single member of the interview unit (for example, a family) provides information for all other members.

Sample selection
: Cases in which the very structure or sample of the survey is biased.

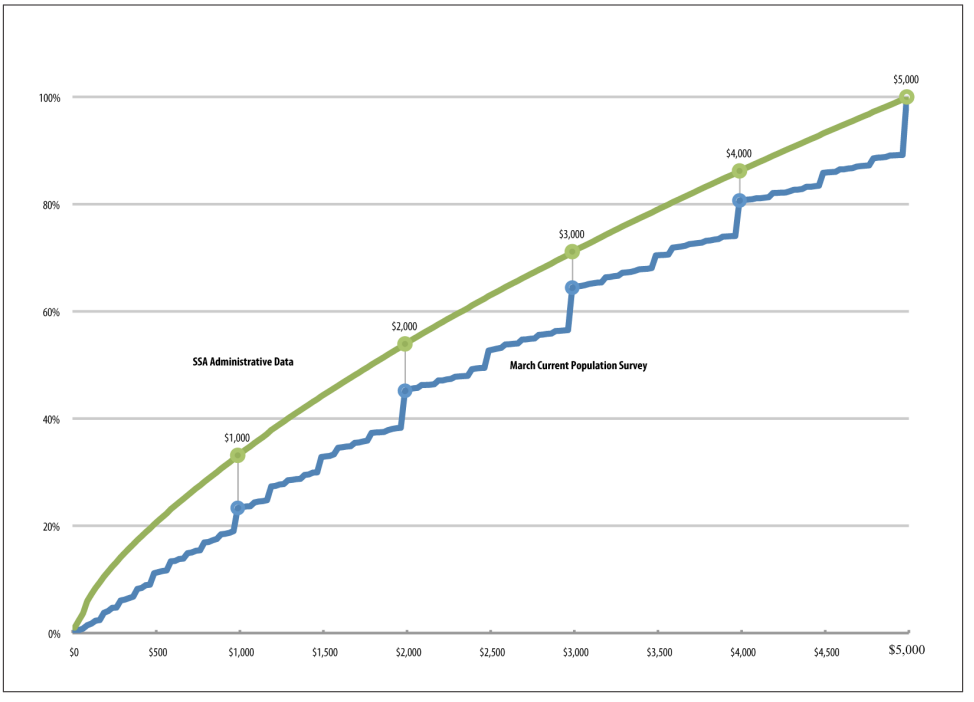


Figure 10-1. Cumulative distribution of earnings for people with positive earnings below $5,000 in the SSA administrative data and CPS, 2009

There are certain strategies you can use to address or even overcome some of these biases, but in cases in which fixing the source of bias is not possible, once again, it is important to be aware of how the issue might affect your results.

Topcoding/Bottomcoding

Tails exist in any distribution, and in the case of earnings or income, for example, those tails can be quite long. To protect the identity of its respondents, the firm or institution conducting a survey might replace extremely high values (topcoding) or extremely low values (bottomcoding) of a variable with a single value. Take, as an example, the Current Population Survey (CPS), a monthly survey of over 50,000 households conducted by the U.S. Census Bureau. In past years, the Census Bureau replaced earnings that exceeded the topcode amount with the topcode amount. For example, prior to 1996, the topcode on earnings was set to $99,999 and so anyone who reported earnings above that cutoff was assigned earnings equal to that value. The effect of that procedure is to create a spike in the distribution of earnings at that level. In more recent years, the Census Bureau has replaced earnings that exceeded some (higher) threshold with average earnings calculated across people with a similar set of characteristics (specifically, gender,

race, and work status). This approach has two advantages: one, it does not create a single spike of earnings at the topcode (although it does create several smaller spikes above the topcode); and two, the sum of all earnings is the same with or without the topcode. In the most recent CPS, earnings above the topcode were swapped between survey respondents; this preserves the tail of the distribution, but it distorts relationships between earnings and other characteristics of the earner.

Without access to alternative data sources (either administrative or the raw survey before it is top- or bottomcoded), it is impossible to ascertain the true value of topcoded or bottomcoded earnings. In the case of the CPS, one group of researchers has used administrative data to make a consistent set of topcoded values over time.[13] Other researchers have used administrative data to construct parameters that can be used to assign new values that better reflect the tail of the distribution.[14] If your data has topcodes or bottomcodes and you can infer the expected distribution of the variable, imputing values in this way is a viable approach.

Seam Bias

The Survey of Income and Program Participation (SIPP) is a panel dataset (that is, one in which the same people are followed over time) and contains information on approximately 30,000 people for about three to four years. In each panel, respondents are interviewed at four-month intervals about their experiences during the prior four months. One effect of this interviewing scheme is something known as "seam bias," in which changes measured across the "seam" (that is, from one interview to the next) are much larger than changes measured within a single interview. This can lead variables to *jump* to a different value every four months (when a new interview occurs) rather than transitioning smoothly as might actually occur. So you might have something closer to Figure 10-2 (a), when instead the pattern should look like Figure 10-2 (b).

To address seam bias, some researchers have suggested making changes to the actual survey, such as encouraging the interviewer to ask respondents to think more carefully about their behavior during the interview interval.[15] Other researchers have suggested strategies that you can use with your data and include choosing specific observations (i.e., specific months), aggregating several observations, or using other more sophisticated statistical techniques.[16]

13. See, for example, Jenkins et al. (2011) and Burkhauser et al. (2012).
14. Feng, Burkhauser, and Butler (2006) and Jenkins (2009).
15. Bureau of the Census (1998).
16. See, for example, Pischke (1995) and Ham, Li, and Shore-Sheppard (2009).

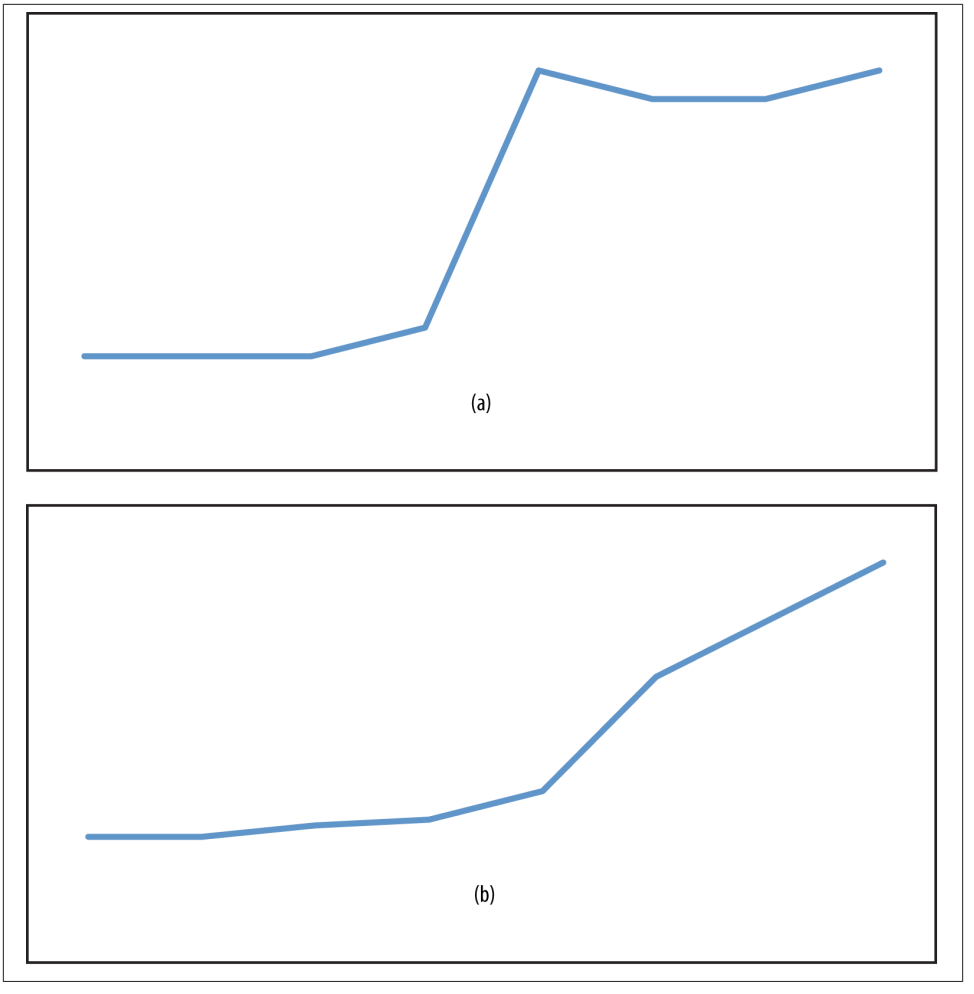


Figure 10-2. With seam bias, variables can jump to different values (a) instead of changing smoothly between surveys (b)

Proxy Reporting

The SIPP and the CPS both allow proxy reporting; that is, interviews in which a single member of the household (or family) provides information for all other members of the household (or family). For example, the husband of a family of four might provide information on his own earnings and labor force status as well as that of his wife, and the educational attainment of his children. Because the proxy may not know the exact

details for everyone in the household, the responses may be incorrect or missing (and thus lead to imputations). If your data includes a variable noting whether proxy reporting exists and who the proxy is, then you can compare nonproxy responses to proxy responses to test whether such reporting biases your estimates.

Sample Selection

Another source of bias that can occur is the very type of people surveyed. Some researchers have questioned whether traditional surveys conducted today provide reliable information for a population that is more technologically engaged than in the past. A recent report by the Pew Research Center suggests that in traditional phone-based surveys, contacting potential survey respondents and persuading them to participate has become more difficult. According to Pew, the contact rate (the percentage of households in which an adult was reached to participate in a survey) fell from 90% in 1997 to 62% 2012, and the response rate (the percentage of households sampled that resulted in an interview) fell from 36% to 9% over that same period. Importantly, Pew concludes that telephone surveys (which include landlines and cell phones) that are weighted to match the demographic makeup of the population "continue to provide accurate data on political, social, and economic issues." However, they also note that "survey participants tend to be significantly more engaged in civic activity than those who do not participate… This has serious implications for a survey's ability to accurately gauge behaviors related to volunteerism and civic activity."[17] Again, it is important to ask detailed questions about your data, including how it was constructed and in some cases, and whether the firm or institution conducting the survey has an agenda, in order to increase the validity and reliability of your results.

Conclusions

As mentioned, I get to work with some of the best data you can get your hands on. The comparisons researchers can make between survey and administrative data, which are becoming more widely available (at least in the field of economics), allow researchers to look at people's behavior in ways that were not available in the past. But those advances have also led to new questions about the accuracy of survey data and the results researchers can reach from models based on that data.

I've tried to show you a number of things to be aware of in your data. I typically work with economic data, which contains information on earnings, labor force participation, and other similar behaviors, but the strategies extend to any survey. Methods that firms

17. Pew Research Center (2012). Sample selection might also occur in the form of different technologies used to ask questions of survey respondents. For example, surveys that ask respondents to use a computer to respond might result in less-reliable responses from elderly respondents, or surveys that only ask questions in a single language might not be representative of certain communities.

or institutions use to produce data from surveys—such as imputations, topcoding, and proxy reporting—should always be documented, and as the researcher, you should always be aware of those methods and then decide how they might affect your results. Similarly, respondents may simply make an error in answering the survey, but it is your job to try to determine—or at least to understand—whether those errors impart bias to your results or are simply a hazard of using the best information you have at your disposal.

References

Bollinger, Christopher R. 1998. "Measurement error in the CPS: A nonparametric look." *Journal of Labor Economics* 16, no. 3:576-594.

Bound, John, and Alan B. Krueger. 1991. "The extent of measurement error in longitudinal earnings data: Do two wrongs make a right?" *Journal of Labor Economics* 9, no. 1:1-24.

Bureau of the Census. 1998. *Survey of Income and Program Participation Quality Profile 1998*. SIPP Working Paper, Number 230 (3rd edition).

Bureau of the Census. 2001. *Survey of Income and Program Participation Users' Guide (Supplement to the Technical Documentation)*, 3rd ed. Washington, D.C.

Burkhauser, Richard V., Shuaizhang Feng, Stephen Jenkins and Jeff Larrimore. 2012. "Recent Trends in Top Income Shares in the United States: Reconciling Estimates from March CPS and IRS Tax Return Data." *Review of Economics and Statistics* 94, no. 2 (May): 371-388.

Congressional Budget Office. 2007a. "Economic Volatility." CBO Testimony Before the Joint Economic Committee (February 28).

Congressional Budget Office. 2007b. "Trends in Earnings Variability Over the Past 20 Years." CBO Letter to the Honorable Charles E. Schumer and the Honorable Jim Webb (April 17).

Cristia, Julian and Jonathan A. Schwabish. 2007. "Measurement Error in the SIPP: Evidence from Administrative Matched Records." *Journal of Economic and Social Measurement* 34, no. 1: 1-17.

Dahl, Molly, Thomas DeLeire, and Jonathan A. Schwabish. 2011. "Year-to-Year Variability in Workers Earnings and in Household Incomes: Estimates from Administrative Data." *Journal of Human Resources* 46, no. 1 (Winter): 750-774.

Feng, Shuaizhang. 2004. "Detecting errors in the CPS: A matching approach." *Economics Letters* 82: 189-194.

Feng, Shuaizhang, Burkhauser, Richard, & Butler, J.S. 2006. "Levels and long-term trends in earnings inequality: Overcoming Current Population Survey Censoring Problems Using the GB2 Distribution." *Journal of Business & Economic Statistics* 24, no. 1: 57 - 62.

Gottschalk, Peter, and Robert Moffitt. 1994. "The growth of earnings instability in the U.S. labor market." *Brookings Papers on Economic Activity* 2: 217-254.

Haider, Steven J. 2001. "Earnings instability and earnings inequality in the United States, 1967-1991." *Journal of Labor Economics* 19, no. 4: 799-836.

Ham, John C., Xianghong Li, and Lara Shore-Sheppard. 2009. "Correcting for Seam Bias when Estimating Discrete Variable Models, with an Application to Analyzing the Employment Dynamics of Disadvantaged Women in the SIPP," *http://web.williams.edu/Economics/seminars/seam_bias_04_06_lara_williams.pdf* (April).

Hirsch, Barry T., and Edward J. Schumacher. 2004. "Matched bias in wage gap estimates due to earnings imputation." *Journal of Labor Economics* 22, no. 3: 689-772.

Jenkins, Stephen P. 2009. "Distributionally-Sensitive Inequality Indices and the GB2 Income Distribution." *Review of Income and Wealth* 55, no. 2: 392-398.

Jenkins, Stephen P., Richard V. Burkhauser, Shuaizhang Feng, and Jeff Larrimore. 2011. "Measuring inequality using censored data: A multiple imputation approach." *Journal of the Royal Statistical Society (A)*, 174, Part 1: 63-81.

Lillard, Lee, James P. Smith, and Finis Welch. 1986. "What do we really know about wages? The importance of nonreporting and Census imputation." *Journal of Political Economy* 94, no. 3: 489-506.

Lubotsky, Darren. 2007. "Chutes or ladders? A Longitudinal analysis of immigrant earnings." *Journal of Political Economy* 115, no. 5 (October): 820-867.

McGovern, Pamela D. and John M. Bushery. 1999. "Data Mining the CPS Reinterview: Digging into Response Error." 1999 Federal Committee on Statistical Methodology (FCSM) Research Conference. *http://www.fcsm.gov/99papers/mcgovern.pdf*.

Moffitt, Robert A., and Peter Gottschalk. 1995. "Trends in the Covariance Structure of Earnings in the U.S.: 1969-1987." Johns Hopkins University. Unpublished.

Pew Research Center. 2012. "Assessing the Representativeness of Public Opinion Surveys." *http://bit.ly/SIhhKP* (May).

Pischke, Jorn-Steffan. 1995. "Individual Income, Incomplete Information, and Aggregate Consumption." *Econometrica*, 63: 805-840.

Roemer, Marc I. 2000. *Assessing the Quality of the March Current Population Survey and the Survey of Income and Program Participation Income Estimates, 1990-1996*. Income Surveys Branch, Housing and Household Economic Statistics Division, U.S. Census Bureau (June 16).

Schwabish, Jonathan A. 2007. "Take a penny, leave a penny: The propensity to round earnings in survey data." *Journal of Economic and Social Measurement* 32, no. 2-3: 93-111.

Schwabish, Jonathan A. 2011. "Identifying rates of emigration in the U.S. using administrative earnings records." *International Journal of Population Research* vol. 2011, Article ID 546201, 17 pages.

Shin, Donggyun, and Gary Solon. 2009. "Trends in Men's Earnings Volatility: What Does the Panel Study of Income Dynamics Show?" Michigan State University. Unpublished.

Ziliak, James P; Hardy, Bradley; Bollinger, Christopher. 2011. "Earnings Volatility in America: Evidence from Matched CPS," *University of Kentucky Center for Poverty Research Discussion Paper Series, DP2011-03 http://www.ukcpr.org/Publications/DP2011-03.pdf* (September).

CHAPTER 11

Don't Let the Perfect Be the Enemy of the Good: Is Bad Data Really Bad?

Brett Goldstein

As a data scientist, your gut and your training tell you to use perfect data for an analysis. This is often a function of classical statistics education, with an intent to submit research and analysis for publication. This is fine and noble, but upon encountering real-world data, the cold reality of dirty data becomes prominent and one must learn to abandon hope of perfection or face an endless loop of frustration.

My wife Sarah, who did her graduate work in public health, has often used the phrase: "Don't let the perfect be the enemy of the good."[1] When confronted with imperfect data, my classical training would say that this data is beyond hope, that it is could never be cleaned sufficiently, and that we would be unable to obtain anything that was truly meaningful. However, this is where we get to the key principle that this should not have to be a zero-sum decision. It is not good, nor is it bad, but it certainly is viable. How can we improve our policies and strategies in absence of perfect data? If it doesn't meet the pristine standards of the classical approach, we must find ways to make the data work so that it can inform the critical decisions that are necessary to move ahead.

But First, Let's Reflect on Graduate School . . .

To explain, I need to step back in time.

In graduate school, I had the same professor for all of my statistics classes. She was thorough, excellent, and meticulous. While it may be easier to have such kind memories long after the courses ended, she taught me a great deal about classical statistics.

1. This phrase, I later learned, is an adaptation of a Voltaire quote: *Le mieux est l'ennemi du bien.*

The professor taught each of the techniques in a very specific way. The majority of the time would be spent on preparing and cleaning the data. There was a remarkably rigorous procedure and series of steps depending on the application of the test. In order to maximize the power of the given technique, each of the tests had a required set of assumptions. As we worked through each and every one of these methods, she had us meticulously document each step of our work (which, in this case, was in SPSS syntax). We would submit a huge amount of work, which she would tirelessly critique. Each time we took a path or made a decision that was incorrect, she would return our pile of syntax to us with the instructions to do it over with the modifications she recommended.

This was a remarkably tedious process because you ended up with 100-plus pages for a single technique. There was little point in arguing about whether there was in fact a meaningful difference between minute changes in skewness. So, through many iterations, I would make it as perfect as I could. The obscure and creative transformation function became my new friend and I took copious notes. They remain in stacks on the back porch in my house, awaiting filing. I am scared to let this work go.

While this classical training remains invaluable, the cold reality of the real world was waiting.

Moving On to the Professional World

My history with data has been varied throughout my career. Early on at OpenTable, we lacked holistic insight into the value of our data; we were completely focused on creating a new company. During those early years, it seemed as if decisions and strategies changed daily. Software development was in high gear and the competition was always nipping at our heels. There was little thought as to what we should collect, the form it should come in, and certainly not how we should extract it. However, as OpenTable matured and repeatedly worked through our mistakes, we started to smarten up. A few years in, we hired an experienced engineering vice president who started to add rigor and a plan to the chaotic system. Concepts such as *schema* and *metadata* joined the vernacular, and with this came the ability to actually extract information from the data. Data analytics came of age as we began to mature as a company.

As OpenTable developed as a company, we had the nimbleness, vision, and motivation to ensure that we were collecting the right data in the correct form, so we had the flexibility to build smart analytic outputs. Certainly, there were mistakes made along the way, but the data was input and output in a usable and strategic way. Still, it is important to note that our data goals were very modest. The types of data problems were typically based on traditional business questions, such as, "what type of support calls would you get for a certain classification of hardware?" This would produce a very standard set of descriptive statistics. We also evaluated our web conversions, but as we used third-party tools for this, there was little need to directly collect and evaluate the data.

Looking back, two aspects of my OpenTable period stand out. First, as was previously noted, we asked only limited questions of our data. We also conducted very little of the inductive research that we now know to be the heart of data mining. Second, which seems so silly now, was that we had an enormous concern about storage. The cost of disk space weighed heavily in our hardware purchase desicions. Furthermore, it was not atypical to have discussions about what data we should keep and for how long as a function of disk cost.

There was also the concern about the underlying data structure and how it would perform as it grew. Single machine instances would rapidly become growing clusters. The cost of keeping data—good or bad—was high. As a function of having to make these choices, often we would only keep what was good and known, rather than keeping more data with the hope of mining other useful information.

From this comes a critical point about bad data and why a text of this nature has become necessary. While bad data has always existed, for a significant amount of time, classical techniques could easily manage it and meet the needs of those standard techniques. Furthermore, as there were limits to what one could retain, these choices often framed and limited the type of quality leading to easier datasets.

As my career at OpenTable was ending, many critical points in the world of data came together. First, we started to see a steep decline in the price of hardware and the notion of white-box computing starting to emerge. Additionally, cloud computing became an immediate game-changer in the realm of provisioning. Lastly, a few years later, we started to see some of the brilliance that would lead to the possibility of truly big data. [2] When you mix these concepts with architectures like Hadoop and the NoSQL database family, you realize that you have the ability to capture enormous amounts of data, much of which may be "bad."

It is certainly speculative—and, as always, hindsight is 20/20—but I can think of many things that we might have done differently at OpenTable if the tools of today were a reality when we started to build the company. However, in as much as OpenTable proved to be a successful venture of which we are all very proud, I will refrain from speculating about this alternative reality.

2. Consider the Google MapReduce paper, "MapReduce: Simplified Data Processing on Large Clusters," by Jeffrey Dean and Sanjay Ghemawat.

Moving into Government Work

Making the transition into the public sector offered an entirely different perspective on data. OpenTable was a small and nimble company, in which we could easily change course. As I entered the enormous bureaucracy of one of the biggest cities in the United States, which had decades of information legacy systems, I was confronted with an entirely different platform and a nontrivial amount of dirty data.

What is dirty data? It has several definitions. First, it is data that is simply incorrect. Value *X* should equal 1, but in in your structure it is equal to 2. That is bad. This can be a function of any number of broken pieces in the business or technology process feeding the data pipeline, from the input sensor to the ETL job that feeds from the transactional system. Second, we see data in which the actual entry is correct, but its associated metadata is either incorrect, or in many cases, nonexistent. This means that you have the value, but you do not accurately know what it means. Third, there may be problems with data as part of the broader set or system. This occurs when you are trying to use the tools of data science to extract information from the data, but it is not adhering to the rules or assumptions of the procedure or algorithm.

From this, we see that "dirty data" is in fact an ambiguous term. It has meanings that can impact the entire enterprise or the individual data scientist trying to run a regression in R. However, the importance of cleaning and fixing data must be understood by all involved, and the principles that apply throughout the organization can mitigate the potential damage it can cause.

Government Data Is Very Real

Given the notion of the "correct" way to manage and prepare data, it is simply not realistic to always apply those techniques to government administrative data. Many datasets that we have in government—such as 911 or 311 calls, crimes, or permits—are inherently not clean because they represent the data of day-to-day life in the urban ecosystem.

When I entered into government service, I had very noble ideas about how data could be used. As a computer scientist, I felt that we could take any problem, break it down into a set of variables, apply a method or code block to the problem, and there would be a solution. However, I had also made assumptions as to how the data would look. For the naive outsider, one would expect that government data would be rock-solid. It would all be contained within a single system, bound by a strict schema and overlaid with comprehensive governance. I thought I would be able to tap into this source of wealth, apply a little computer science magic, and produce remarkable results.

The technical reality of what I found was quite different than my expectations. The systems had been built over time and incrementally. The end product was a system that lacked early functional vision for the design. (When you are building a scalable system,

you need to have that architectural vision or you will continue to build layers upon layers.) From a strictly data-centric perspective, this often leads to an overly complicated schema. You end up with 500 tables instead of 50. Rather than having a good plan from the beginning, you start adding fields over time. These were the types of systems that I encountered as I started on my path to develop models for predicting homicides and shootings.

Should I have been surprised by this? Absolutely not. What the government had done was something I had seen during the early days at OpenTable. We emphasized creating systems and focusing on the market rather than the technology; we raced ahead without giving the underlying architecture enough attention. In addition, business intelligence was still in its infancy. A good report would often just consist of output in a spreadsheet. At that point, data mining and predictive modeling were completely absent from the strategy.

Then there was the perception component. Data that was of slightly questionable quality, or data that could possibly have contained any errors, could not be used. It was assumed if a small piece of data lacked veracity, all of the data needed to be disqualified.

Service Call Data as an Applied Example

Consider 911 service call data.[3] Historically, the core of public safety data was driven by reported crime information, as derived from the official police reports that are filed by residents. (Many issues warrant discussion from the idea of reported crime, but would be out of scope for this chapter.) That said, it is important to discuss whether there is a more robust set of data that could be used or at the very least complement the crime data. And that is 911 data.

911 data is a fascinating set. The volume of calls that are recorded within that set far exceed the number of entries in the crimes set. When I first started working on this set, I realized that we had a robust amount of data. As a result, we could provide more insight into the ecosystem of a neighborhood. A very interesting and specific case came from my time working as a sworn officer with the Chicago Police Department.

The 11th District, where I worked, received a substantial number of calls reporting the sale of narcotics. People would note that narcotics were being sold at specific locations. Of course, we would respond to these calls. As an officer responding in a marked vehicle, more often than not, the individual selling the narcotics would see us or be notified by one of his counterparts. By the time we arrived at the site, the perpetrators had dispersed,

3. Editor's note: for readers who don't live in the USA, we call 911 to reach emergency services, such as police or firefighters. It is the equivalent of 999 in the United Kingdom, 15/17/18 in France, or the Europe-wide 112. (Many European countries continue to maintain their old emergency numbers in addition to 112, but that's another story.)

so we could take no action. When classified, the service essentially received a code that is similar to an unfounded disposition. This leaves one with the quandary: what does this point of data mean? Does it mean that no incident took place? That's an incorrect assumption. Does the converse apply? That's debatable. The reality is that neither of those options worked. Nevertheless, we do not need to ignore this example as data that is not usable.

In order to make informed strategic and tactical decisions in an environment with imperfect data, one must make compromises. From an academic perspective, these compromises make little sense. We are dropping the quality of the analytic and adhering to the gold standard of research. Still, I have repeatedly noted that it is better to have an informed decision built on imperfect data than to have a decision built on no data at all. When one accepts that imperfection, it opens up the ability to integrate data into all supports of projects and policies. The ability to even have incremental gain in their decisions may lead to substantial improvements in policy.

I believe that there is a dichotomy between standard research and operational reality. Classical research requires everything from very tight and structured statistical tests to randomized sampling. Reality doesn't always allow for random sampling, and often the data from the ecosystem won't permit the meeting of all the requisite assumptions. That said, this is not just a choice between absolute accuracy and complete inaccuracy. This instead is an opportunity to extend our toolkit to use tools that are beyond your typical "gold standard."

Moving Forward

Going forward, government needs to tread carefully as we define how we should be using data. We need to be mindful of the lessons from the past. In particular, there are two areas that have frequently created problems:

Fear of academia. Traditionally, academia is the home of some of the top research of data and assorted problems in the sciences. While this may produce gold standard research, it often come with two associated concerns. Such detailed and controlled research tends to move slowly. Receiving data and analyzing it for multiple years is not helpful in the terms of a nimble government that is looking to make smart tactical and strategic moves.

Fear of a surprise outcome. In this scenario, the evaluators review an initiative and return at a later date announcing to the world that it was a failure and all for naught. This is a very legitimate concern. Long timelines do not help government do business better in the near term. Furthermore, the surprise outcome both causes awkwardness and prevents future collaborations.

To avoid these problems, I have developed close relationships with academia and opened clear channels of communication. More recently, I have been working with Carnegie Mellon University and the University of Chicago. Our relationship is founded on a couple of key principles.

The first is open communication. With that involvement comes a lack of surprises. Both sides are aware of concerns and are focused on making the relationship work over the long term.

Second, and one of critical importance, is our determination to make small and incremental gains. In the world of big city issues, making small changes has huge value. Single-digit percent changes in a given problem can have huge returns. Imagine if you reduce robberies by two percent or cut unemployment by a single percentage point. As my academic partners have realized the enormous values of these returns, it has led to more active involvement. We may in fact be working toward a very high goal, but we also understand the value of more immediate results. Allowing better access to data leads to better near term answers and leverages some of the deep talent that is available in our communities.

The third principle is is the marriage of social scientists and computer scientists. Classic social scientists are often hesitant to work with dirty data, which can complicate our collaborations. That changes if we, as computer scientists, introduce a new set of tools. This is a kit—a machine learning arsenal, really—that is better equipped to handle the nuances of dirty data. Historically, these have been two very different worlds. In Chicago, we are actively working to merge these groups. That will enable us to solve problems in very different ways.

In general, it is often believed that government information technology is ten years behind the private sector. However, this lag is not acceptable. In order to serve our residents who depend on the government for critical services, our technology must be nimble and innovative.

Lessons Learned and Looking Ahead

Dirty data is there for a reason. Sometimes there is nothing you can do about it, while other times, it is absolutely your fault. Whatever the case, we have to resist the temptation to simply throw away imperfect data.

Just as the reality of daily life and complex ecosystems have high levels of entropy and thus "dirtyness," so does the data that surrounds it. We cannot use this as an excuse to avoid solving problems, but instead it should motivate us as data scientists to continue to explore the power of these new techniques and apply them to the problems that are critical to how we live.

As our world becomes more data-driven, it is critical that we understand what we can and cannot do. On a closing note, I offer some ideas that can help us move forward.

Bring data into the enterprise spotlight. From the enterprise and business perspective, data planning and data science must be planned at an enterprise scale. Similar to how we brought information technology security to the executive staff over the past decade, we need to make similar moves for data.

This is about planning one's data strategy across the entire enterprise rather than solely within the business unit. As new projects emerge, we must determine how they relate to the other pieces of data being collected. Whether it is within a broad schema or creating a comprehensive lexicon for keying attributes, creating the framework for consistency and broader planning provides huge returns.

Improve the inputs. Beyond the storage and tagging of data, we must improve the integrity to the inputs and sensors. We should make efforts to validate and ensure the understanding of data as it enters the enterprise. This ranges from interaction with the subject-matter expert to allow for smart metadata, to ensuring the validity and reliability of the inputs. Though these are common-sense components of statistics and methods, it may not be obvious within the business. This early investment may yield huge returns.

Determine which data is truly unusable (but save it for later). There are times when we simply cannot fix bad data. It is critical that we truly exhaust our options before we finally give up. That said, I am rarely in favor of discarding data (does that make me a data hoarder?). Our methods and tools will only improve over time, so we may as well hold on to our dirty data, and save it for a time when we can make use of it.

CHAPTER 12

When Databases Attack: A Guide for When to Stick to Files

Tim McNamara

My Masters dissertation still feels likes a personal defeat. At least four months of the nine-month project were sabotaged because I didn't understand the implications of the technology choices that I was making. This chapter will be a bit of a postmortem of the project and a walk-through of a strategy I should have used: storing plain-text data on-disk, instead of in a database.

History

But first, a little more about my story. In 2010, I was undertaking a Masters in Public Policy at Victoria University of Wellington. I was focusing my efforts on the arguments surrounding open data within the science sector. Specifically, I wanted to know: do the arguments that academics, officials, politicians, and the public align? I had the sense that open data and open government meant quite different things to different people, and I wanted to quantify that.

Getting access to information about what officials thought about open data was fairly easy. There was a major review of New Zealand's publicly funded research bodies being undertaken at the time. Policy advice between departments is available under the Official Information 1982, which is New Zealand's freedom of information statute. Information from politicians are even easier to find, as they talk all the time. All press releases are syndicated via scoop.co.nz. But what about the views of the public at large?

The political blogosphere is very active. While blogs have diminished in quality and depth with the maturity of social media, passionate people seem to draw themselves willingly into the flame of political debate.

Building My Toolset

I needed to learn text analysis and natural language processing (NLP) techniques, but my supervisor and I were confident that it would be possible to draw some inferences from this large corpus of material. The general approach was to crawl the New Zealand political blogosphere, identify discussion threads about open data, identify people who comment on this topic, then pull out the themes of these comments. I was also intending on spending some time classifying commenters. Perhaps we could see if there were things special about commenters interested in open data.

The dissertation itself was a success, but none of my work from the blogosphere could be included. Partway through the project, I believed the web crawling would be my greatest challenge. It looked easy at first. Blogs have articles. I wanted those articles. Sadly, they also have more than articles. They have comments, dates, locations, tags, and links. Commenters often have counts associated with them. Comments themselves have ratings.

This complexity in the data extraction process really slowed the development of the crawlers. I had several problems with missing data and exceptions. A breakage in the extraction process would break the crawl. A broken crawl that was not cached requires restarting the crawl from the beginning. I was able to clean up and streamline my extraction processes, though, so the crawling became less off a problem over time.

As it turned out, the *real* problem was storage, because I needed more ad-hoc query support than the tool could handle.

The Roadblock: My Datastore

The database that I chose to use was CouchDB. There were several reasons why I made this decision:

- I do not have a computer science background, and I liked the idea of sending JSON strings to the database and for it to figure out how to store things appropriately.
- It used an HTTP API, which I felt like I could understand.
- The MapReduce paradigm promised high performance.
- Documents are versioned.
- CouchDB's schemaless nature was a good fit, as I was consistently adding new fields to the JSON I generated from sucking it out of web pages.
- I was quite interested in getting a deep understanding for the database that looked likely to be rather influential, if not preeminent within the web world.

CouchDB was approachable, friendly, and fast to get data into the system. The wider community seemed to be doing some very impressive things with it. While I am sure

that fashion had a part of it, there were genuinely good reasons from a newcomer's point of view that CouchDB was the right system. MapReduce is purportedly very scalable, and as CouchDB supports MapReduce queries, I had assumed that this would be the perfect fit. That said, there were two hurdles that I wasn't able to surmont.

CouchDB provided a poor debugging experience for developing mappers and reducers. I had no idea how to interpret the Erlang/OTP stacktraces. Errors tended to pop up after some number of problems had occurred. I found little guidance about what was wrong with my JavaScript. The error messages were, effectively, "something broke." So I would try again. And again. Finally, something would work. That said, the only reduce functions I ever implemented were sums. I could not do anything more complex. Because errors in reduce functions would only appear once all of the maps had been undertaken, the develop/run/break/debug/develop cycle would take (at least) several minutes per iteration. In the end, I decided to just create mapping fuctions and would write reducers manually with the resulting views.

The debugging story can be improved by slicing a small portion of the database out and developing new queries there. That way, with orders of magnitude fewer documents, problems arise much more quickly.

The real problem for me came for when I wanted to run a new query on the dataset I had created. There ended up being about 70GB work of blog posts, comments, and metadata within the database. (You may recall, I'd already stripped out the HTML formatting during the crawl stage. This was 70GB of actual *text*.) I learned that, to conduct NLP work within CouchDB's mapping functions, building an index took days on my inadequate hardware. Worse still, things could still fail right near the end. That's what happened to me. At that point, I decided to cut my losses. The data still sits on an old hard drive somewhere, silently taunting me.

This is not a chapter about the failings of CouchDB. It's a chapter about how not understanding the tools that you're running on can have very negative consequences. CouchDB is designed for a read-heavy workload, such as a high volume website. It processes new data for a view once that view has been requested by a client. In a high-volume website, new hits will appear all the time. It only needs to process a few documents at the most in order to serve the request. Under a web crawling scenario, this is not the case.

In case you were wondering, my dissertation passed well using the material from traditional sources.

In the sections that follow, I'll explain how to use files and the filesystem as your database. Surprising though it may be, the filesystem can make for a very useful datastore.

Consider Files as Your Datastore

Sometimes, you're better off skipping the database and using the filesystem as your datastore. Consider the following:

Files Are Simple!

Simplicity and ubiquity are virtues that are very hard to find in technology products. There is simply less bureaucracy to deal with when you use files.

Files Work with Everything

Just about everything will (sort of) know how to process your data. Utilities will need coaching to understand how to extract structure from your files. This is generally not a significant barrier to entry, though. If you stick with a few conventions, such as using common and consistent delimiters, then you will be fine.

Compare this to a traditional database system, for which you will often need a specific adapter, which may in turn have its own API. If you're using an abstraction layer, such as an object-relational mapper (ORM), then you need to understand its unique syntax. Recent databases often provide an HTTP API, but this is still more of a burden than opening and reading a file.

Being able to have everything read your data makes life as a data wrangler easier. It's quite easy to swap between programming languages and tools. (Perhaps you want to use R's `ggplot2` for plotting, Python's NLTK for natural language processing, Octave for familiar matrix processing, and Fortran for numerical efficiency.) Hadoop and other implementations of MapReduce will typically work with files. It is easy to serve the data in ad-hoc fashion. A single line of Python can create a (temporary) web server: `python -m SimpleHTTPServer 8080` will make the current directory contents available, via web browser, on your machine's port 8080.

Also, OS tools can replicate functionality you may miss when you forego a traditional database. For example, the `inotifywait` tool can watch for changes and run a command when a given file is modified.

Files Can Contain Any Data Type

Traditional database engines do not handle large, unstructured file data (often referred to as binary blobs) well. Databases would prefer to work with fairly regularly sized records in order to efficently store tables and documents on disk. If records vary greatly in size, it is far more difficult to optimize how things are stored. Binary blobs get in the way.

By comparison, a file can contain any type of data, structured or unstructured.

This flexibility does, however, incur a cost when storing or retrieving typed data. For example, while a person will see that `3.14159` is a number, in a file this data is just a series of characters. Something has to tell your program that this is a number, and that may involve some translation overhead. Similarly, you can store objects in a file by defining a serialization process, but you then need to write a routine to *_de_*serialize that data in your application. That can be a burden, both in terms of developer time and processing time. Despite this, files can be surprisingly useful for getting off the ground.

Data Corruption Is Local

In case of a hard shutdown, any corruption to a database's files can create a deep mess. Damage to files is generally localized to those files. Furthermore, file systems work hard to prevent damage in the case of power failure. (The disk contollers will sometimes lie to the file system and tell it that they've already written to disk when they haven't. This can cause problems but they're reasonably rare.)

File systems are really smart. They take care of many things for you. File systems are able to handle data which varies in size much better than databases. Databases don't really like binary blobs that vary considerably in size. In general, database relations (aka tables) are optimized for rows of roughly consistent size. This enables them to optimize search strategies and so forth.

For example, journaling is a really useful feature for being able to ensure the stability of what's happening. They just don't care too much about data types. Everything is just bytes.

They Have Great Tooling

When you use a database system, you miss out on the powerful, built-in OS utilities to cut and slice the contents. Many such tools are heavily tested (having been run billions of times over the years), reliable, and are generally implemented in C for performance. `grep` is one of the fastest ways to hunt for things. Remote replication is only an `rsync` away. A word count for a file takes two characters: `wc`. A version control system such as Subversion or Git will help you maintain a history of your changes over time.

There's No Install Tax

Working with files has a very low barrier to entry. You can eliminate a lot of friction by not needing to install client libraries, compile drivers, or worry about a schema.

Files certainly aren't ideal for every situation. They're best suited for cases that are read-heavy, require few modifications, and incur minimal cost to translate the data into a typed representation. That means storing logs (such as webserver logs), recorded events, web crawlings, large binary data, and sensor readings, which are all suitable use cases.

Sensor readings do not change over time. History is fixed. Therefore, there is often little need to require features such as multiwrite concurrency. A similar case involves serving static web pages. Dynamic websites, which have been converted to files by a static site generator, are much more scalable than serving dynamically generated content.

File Concepts

Working with file datastores isn't all roses. You'd do well to consider different file types, formats, and possible issues before you move to an all-file setup.

Encoding

An *encoding* is a standard for converting zeros and ones to characters. Traditional encodings, such as ASCII, have a translation table that converts a bit sequence such as 111100 (60 in decimal) to the capital letter `A`. Unfortunately, there are only 256 bit sequences within a byte. Unicode adds some complexity, but offers a lot more flexibility. One Unicode encoding, UTF-8, is variable length and can support annotating base characters with accents with information from supplementary bytes.

Text Files

Text files are, simply, files that adhere to an encoding. They are typically printable, in that the human eye can read their contents.

Binary Data

Think of this as files that do not conform to a well known-encoding for representing text. Therefore, while bytes from binary files may contain printable characters, the text printed will be gibberish. Some binary formats require special tools to translate the raw bytes into meaningful representations.

Memory-Mapped Files

Memory-mapped files mirror the in-memory representation of data, to improve performance. While CPU does not distinguish between data that comes from memory and data that comes from the disk, your users will, because the former can be tens of thousands of times faster than the latter.

File Formats

Formats set standards for how data is represented within a stream of consecutive bytes. Table 12-1 includes a few different representations of some public data about me:

Table 12-1. Format overview

Python literal syntax	`{'country': 'New Zealand', 'name': 'Tim McNamara'}`
JSON	`{"country": "New Zealand", "name": "Tim McNamara"}`
YAML	`{country: New Zealand, name: Tim McNamara}`
CSV	`name,country` `Tim McNamara,New Zealand`

Python pickle format	`(dp0`
	`S'country'`
	`p1`
	`S'New Zealand'`
	`p2`
	`sS'name'`
	`p3`
	`S'Tim McNamara'`
	`p4`
	`s.`

S expressions	`((country, "New Zealand"), (name, "Tim McNamara"))`

Each of these representations performs roughly the same function, with many details at the edges. There are several choices for what might be optimal. This lends itself to trivial debate, sometimes known as bikeshedding.[1]

One way to decide about which format should be used is to consider the needs of its audience. If people are expected to read and potentially even edit the files, then one should optimize for human readability and editability. Syntax errors are fustrating to experienced programmers and intimidating to novices. If your audience for your data are computer parsers, then you should use something like Google Protocol Buffers,[2] Apache Thrift,[3] or JSON. If your audience is your future self or someone looking at the data in ten years' time, you should use an open format that has existed for decades. When you use open formats, you increase the chances that you'll actually be able to read the files in the future.

A file format is a method for translating files into information. There is no "best" file format. None of the representations above are very useful for storing image data, for example.

Lastly, it is often useful to wrap one format into another. When interacting with a web API, the JSON objects you receive are likely to be sent in a compressed state, such as gzip. (Please review the following sidebar for things to consider before you adopt JSON.)

1. *http://en.wikipedia.org/wiki/Parkinson%27s_Law_of_Triviality*
2. *http://code.google.com/p/protobuf/*
3. *http://thrift.apache.org/*

JSON Is Not the World's Best File Format

It is very likely that you will be using JSON all over the place. It is not necessary. Feel free to invent your own file format for particular jobs. If you really think that JSON is best, you should probably use YAML. YAML is a format that is designed primarily for human readability.

JSON should not be used in configuration files. Its syntax is far too strict to be helpful. JSON looks too similiar to other languages' serialization formats, which leads to little bugs when you are actually writing things by hand.

Spot the bug here:

```
{
    "id": 10020,
}
```

JSON does not allow a comma within the last element. This is an unnecessary hurdle. Coming from Python, you may be even tempted to use single quotes. Or even worse, comments.

There are several advantages of YAML over JSON:

Readability
: YAML omits most double quote characters. Fewer symbols make other little data things easier to read.

A defined comment syntax
: JavaScript comments are illegal in JSON.

Named entities
: This allows a great deal of compression because rather than including "University of Washington" everywhere, you can use the much shorter @wc or some other convenient variable name.

Decent streaming capabilities
: YAML documents can be processed without reading everything into memory. This capability does not exist within JSON. If you are reading data in from a stream, then your parser needs to slurp everything into memory before it can deserialize it.

JSON is a very common format, but it is not well-suited for many of the tasks for which it has been allocated.

Delimiters

Delimiters separate fields and records within a file. For example, in a comma-separated file, the newline character separates records, and commas separate fields.

Picking a delimiter lives in a class of problems that sucks more energy than is possible to imagine. I know of a production system that uses four bars (||||) to delimit strings, to work around not being able to modify the database schema and a fear of modifying the underlying data to escape text.

Paying homage to *http://xkcd.com/927/*, the ASCII standard includes delimiter characters that are almost never used. ASCII 30 is the record separator and ASCII 31 is designated for delimiting fields. In principle, user-facing systems reading in this data could neatly align things together in a tabular form. In practice, there are no user-facing systems that I'm aware of that even support these delimiters, let alone do something nice for the user.

Probably the biggest reason for the lack of adoption is that it's impossible to type the delimiters on a keyboard. CSV files look ugly and are misaligned, but we can all find the comma key.

You are best to split records on new lines and either use tabs or commas. Escaping with double quotes and back slashes is very common and likely to be easy enough to implement in bespoke systems. Whatever you do, try to avoid delimiter creep. If someone recommends |||| because it is "unlikely to cause problems," then please steer them towards the light of common standards.

Finally though, whitespace delimiters are fairly prone to error and are generally clunky. In an increasingly international world, whitespace itself won't work at all. Many Asian languages do not use whitespace to delimit characters. Perhaps it could be time to start thinking about actually implementing an old standard.

A Web Framework Backed by Files

As an example, consider a hypothetical content management system that uses files for its underlying datastore. While seemingly ludicrious, the concept may be quite suitable for small teams of technically able people. While we lose out on an attractive admin interface, we gain speed, security, and scalibility in exchange.

The concept is quite simple. We create plain text documents like this:

```
Title: Music to my ears

Was excited to read that people didn't simply move to the next greatest
database and build web framework to create a new website that no one will
read.
```

The computer then turns those documents into something that looks like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Music to my ears</title>
```

```
  </head>
  <body>
    <p>Was excited to read that people didn't simply move to the next
      greatest database and build web framework to create a new website
      that no one will read.</p>
  </body>
</html>
```

Let's discuss why this might be a useful thing to do, and then talk a little bit about how to actually implement it.

Motivation

Why would we want to do this? Here are a few answers:

Speed
: Flat files imply that we will be serving static content. Static content is very easy and fast for web servers to serve to a client, in part because there's no need to call a separate library, program, or remote system to generate content on-the-fly.

  Static content is also very easy to cache. The file system provides a lot of metadata for its files, such as the time they were last modified. This can be provided to web clients, who will only ask for the page again if it has not been modified. The same rules apply for scaling out. If you are having difficulty handling requests, a content delivery network (CDN) will be very happy serve your static files for you. Dynamic content is much more difficult to hand off to an external, third-party service.

  Many web developers will mention that the database is the bottleneck. Let's take the bottleneck away: we'll do the processing up front, rather than hitting the database with reads every time the website has a new request.

Security
: Because there is no way to touch the database from the web content, our website will be resistant to intrusion, defacement, or worse.

  Many security vulnerabilities occur because the outside world can manipulate services that are running on the server. For example, SQL injection is a means to run arbitrary commands on the database via things like web forms. The static file architecture prevents this. (Granted, someone could hack into the server itself—either through another, unrelated service running on the machine, or by infiltrating the data center—but this is a vulnerability of *any* computer system.)

Familiar tools
: People are accustomed to using text editors. They know how to save files to disk. This process enables users to retain the tools with which they are familiar, rather than having to fiddle with clunky WYSIWIG editors built in JavaScript.

Remote users may have a few difficulties at first. They will need to use a tool like FTP, `scp`, or `rsync` to push their local content to the server. That might be overwhelming for some. However, these commands are fairly trivial for anyone who is comfortable working on a command line. You could provide these people with any cloud-based service for syncing files between computers.

Easy backup
: Backups and restores are now easy, because the backup system only needs to talk to the standard filesystem. Every backup tool can handle plain text files.

  By comparison, backing up and restoring databases can be a pain. It typically requires that you use special, database-specific tools, and some of those create special formats.

Implementation

Given what we've outlined above, the implementation is fairly straightforward. We have a directory of plain text documents that store our raw data. That directory is monitored by `inotifywait` or an equivalent tool. When the contents of that directory are modified, `inotifywait` calls a processing script with the file that has changed as an argument. That script needs to be able to create a new HTML document from the file's contents when created or modified, and remove the HTML document upon deletion.

A working implementation of this system is available at *http://github.com/timClicks/baddata/*.

Reflections

In many regards, this chapter has been about conservatism. I encountered a negative experience with new and unfamiliar tools. That sent me back to rely on what I know. Files are approachable and everywhere.

It is very difficult to know if a particular technology will be suited for your use case. If you discover that it's not, you can be too far down a particular track to escape and move to something else. The project can feel futile. You will develop workarounds for problems with the original stack, which may themselves need to be worked around at some later date. And so on.

Files are not a perfect solution, but they can be a good enough solution. Hopefully, if you start your data analysis project using files, you'll buy yourself some time to evaluate several other tools. Sometimes waiting until you have collected the data you are interested in analyzing can be the best way to discover how to store it. Until then, you may be able to make do with files.

I hope that you have been able to extract lessons from my experience. My aim with this chapter has been to provide a different view on how to structure a data mining project. You have not found the universal truth here. Hopefully, what you have found is a series of useful tools that you can apply bits of in future projects. Thank you for reading. *Kia kaha.*

CHAPTER 13

Crouching Table, Hidden Network

Bobby Norton

"You were enlightened?"
"No. I didn't feel the bliss of enlightenment.
Instead… I was surrounded by an endless sorrow."
卧库表
隱网络
—Yu Shu Lien describing the effects of bad data
(…or something similar…) to Li Mu Bai in
Crouching Tiger, Hidden Dragon

Data is good to the extent that it can be quickly analyzed to reveal valuable information. With good data, we're able to learn something about the world, to increase revenue, to reduce cost, or to reduce risk. When data is locked up in the wrong representation, however, the value it holds can be hidden away by accidental complexity.

Let's start our journey in the context of a seemingly simple problem faced by many IT-intensive enterprises: keeping track of who is paying for costs incurred by the business. To name but a few types of costs, consider servers, software licenses, support contracts, rent in the data center, Amazon EC2 costs, the cost of teams building software and providing support; the list goes on. As a business grows and the costs associated to run the business increase, sooner or later, someone will ask, "What are we paying for?"

From a business owner or executive perspective, not every department within a business uses shared assets to the same degree. As such, it doesn't make sense to evenly split costs across the entire business. Doing so may hide the fact that a particular line of business isn't profitable and should be shut down. For large enterprises, this could be a multi-million dollar problem.

A Relational Cost Allocations Model

An accountant would likely think of cost allocation in terms of accounts called cost centers. Cost centers have both revenue and costs, and subtracting cost from revenue yields either profit or loss. We need a way to track the incoming costs from cost drivers (the expenses we incur to run the business) to all cost centers. Let's euphemistically call our cost drivers *assets*, in the hope that they'll contribute to our business being profitable. To start modeling the cost allocation domain, we can use a technique from the relational database world called Entity-Relationship (ER) modeling. Entities correspond to the concepts in our domain that will be uniquely identified. Relationships capture attributes describing connections among entities. Please see Figure 13-1 for an ER diagram of this scenario. The cost allocation relationships we'll consider are:

- One Asset can be allocated to many Cost Centers; one Cost Center can have many Assets.
- One Department can have many Cost Centers; one Cost Center can belong to many Departments.
- One Asset can be allocated to many Departments; one Department can have many Assets.
- One Asset can be allocated to many Services; one Service can have many Assets.
- One Service can be allocated to many Products; one Product uses many Services.
- One Product can be allocated to many Departments; one Department can have many Products.
- One Product can be allocated to many Cost Centers; one Cost Center can have many Products.

We've identified five entities and seven associative entities that capture our many-to-many relationships. Our entities need a name and description, and our allocation relationships need an effective date and percentage to capture the allocation ratio. This gives us the basis of a conceptual schema that we can use to attack the problem.

The longest path that an asset a1 could take through the entities in this schema would involve the asset being allocated to a number of services (s_n), products (p_n), departments (d_n), and cost centers (c_n). The resulting number of rows for a_1 is r_1 and is given by:

$$r_1 = s_n \times p_n \times d_n \times c_n$$

If $s_n = 5$, $p_n = 2$, $d_n = 2$, and $c_n = 3$, we have 60 rows for a_1. If this represents an average number of allocation rows per asset, then we would have 600,000 rows in a final report covering 10,000 assets.

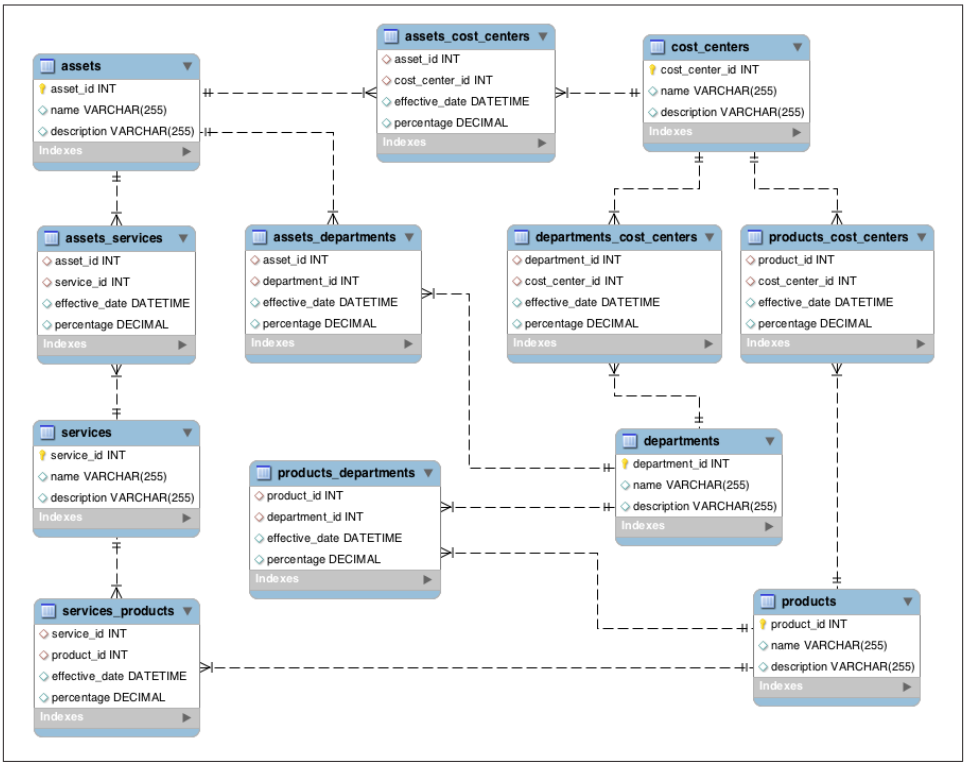


Figure 13-1. Entity-relationship diagram

Assume we've inserted some sample representative data into the above schema. What data would the allocation report rows contain? Let's take a simple case, an asset allocated to two cost centers. An SQL query to retrieve our allocations is:

```
select * from assets
inner join assets_cost_centers on assets.asset_id =
assets_cost_centers.asset_id
inner join cost_centers on assets_cost_centers.cost_center_id =
cost_centers.cost_center_id;
```

Two sample rows might look like the following:

asset_id	name	description	asset_id	cost_center_id	effective_date	percentage	cost_center_id	name	description
1	web server	the box…	1	1	2012-01-01…	0.20000	1	IT 101	web services
1	web server	the box…	1	2	2012-01-01…	0.80000	2	SAAS 202	software services

The percentage attribute in each row gives us the ratio of the asset's cost that the cost center will be charged. In general, the total asset allocation to the cost center in each row is the product of the allocation percentages on the relationships in between the asset and the cost center. For an asset with n such relationships, the total allocation percentage, a_{total}, is given by:

$$a_{total} = cost \times \prod_{k=1}^{n} p_k$$

If the server costs us $1,000/month, the IT 101 cost center will pay $200 and the IS 202 cost center will pay $800.

This is fairly trivial for a direct allocation, but let's look at the query for asset to service, service to product, product to department, and department to cost center:

```
select * from assets
inner join assets_services on assets.asset_id = assets_services.asset_id
inner join services on assets_services.service_id = services.service_id
inner join services_products on services.service_id =
           services_products.service_id
inner join products on services_products.product_id = products.product_id
inner join products_departments on products.product_id =
           products_departments.product_id
inner join departments on products_departments.department_id =
           departments.department_id
inner join departments_cost_centers on departments.department_id =
           departments_cost_centers.department_id
inner join cost_centers on departments_cost_centers.cost_center_id =
           cost_centers.cost_center_id
```

One of the resulting rows would look like:

asset	percentage	service	percentage	product	percentage	department	percentage	cost_center
file server	0.10000	Data Science S…	0.40000	Search Engine	0.7000	R&D	0.25000	SAAS 201

Our allocation percentage here is now the product of the allocation percentages in between the asset and the cost center. For this row, we have

$$\prod_{k=1}^{n} p_k = 0.1 \times 0.4 \times 0.7 \times 0.25 = 0.007$$

A server with a $3,000 monthly charge would, therefore, cost the SAAS 202 cost center $21 per month.

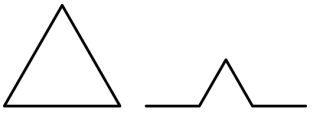
These are only the two corner cases, representing the shortest and the longest queries. We'll also need queries for all the cases in between if we continue with this design to cover assets allocated to departments and products allocated to cost centers. Each new level of the query adds to a combinatorial explosion and begs many questions about the

design. What happens if we change the allocation rules? What if a product can be allocated directly to a cost center instead of passing through a department? Are the queries efficient as the amount of data in the system increases? Is the system testable? To make matters worse, a real-world allocation model would also contain many more entities and associative entities.

The Delicate Sound of a Combinatorial Explosion…

We've introduced the problem and sketched out a rudimentary solution in just a few pages, but imagine how a real system like this might evolve over an extended period of months or even years with a team of people involved. It's easy to see how the complexity could be overlooked or taken for granted as the nature of the problem we set out to solve.

A system can start out simple, but very quickly become complex. This fact has been deeply explored in the study of complex systems and cellular automata. To see this idea in action, consider a classic technique for defining a complex graphical object by starting with two simple objects:



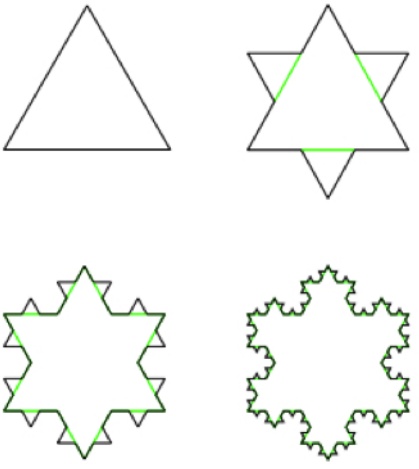
"One begins with two shapes, an initiator and a generator…each stage of the construction begins with a broken line and consists in replacing each straight interval with a copy of the generator, reduced and displaced so as to have the same end points as those of the interval being replaced."

Benoît Mandelbrot[1]

In just three iterations of this algorithm, we can create a famous shape known as the Koch snowflake.[2]

1. Przemyslaw Prusinkiewicz and Aristid Lindenmayer. 1996. *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., New York, NY, USA.

2. *http://en.wikipedia.org/wiki/File:KochFlake.svg*



Not so different than what just happened with our relational schema, is it? Our entities play the role of the "straight interval," and the associative many-to-many entities act as the complexity generators.

The Hidden Network Emerges

Let's step back. If we were to just step up to a whiteboard and draw out what we were trying to accomplish with the asset allocations for our servers, our sketch might look something like Figure 13-2.

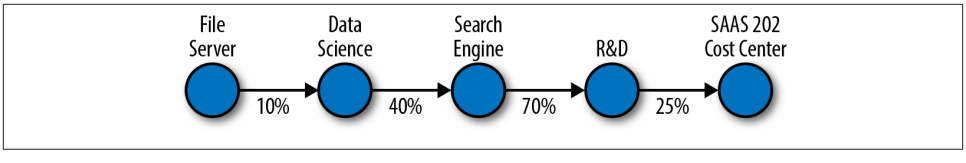


Figure 13-2. Visual model of what's being accomplished

The visual model makes it easier to see that, for purposes of calculating allocated cost, there really isn't a great deal of difference among these "types." At the most fundamental level, there are dots, lines, and text describing both. We can read the first dot and line in this drawing as, "The File Server asset is allocated 10% to the Data Science service."

We can simplify our ER data model dramatically if we take a cue from this visual model and define our domain in terms of these dots, lines, and text descriptions. Because our lines are directed, and we don't have any cycles, our allocation model is actually a directed acyclic graph. The dots are vertices, the lines are edges, and the properties are key-value pairs that optionally add additional information to each vertex and edge.

We don't need typed entities to represent our vertices and edges. We can simply include a `type` property on our vertices to capture whether the vertex represents, for example, an asset or a service. We can use labels on the edges to represent that each edge represents an allocation. We can capture the percentage and effective date of the allocation as a property on the edge, along with any other properties we might need.

Given that we now have only vertices that can be allocated to one another, our schema reduces from the model in the ER diagram to Figure 13-3.

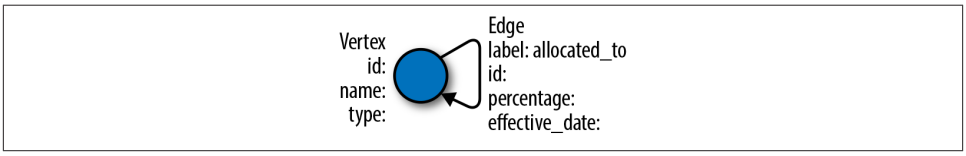


Figure 13-3. Reduced from the model in the ER diagram

Storing the Graph

Once we have decided on a graph data structure to model our domain, we have a choice to make about how we will persist the data. We could continue to use a relational database to represent our graph, perhaps storing the list of allocations as an adjacency list of edges. SQL is a set-oriented language, however, and expressing graph concepts is not its strength. Whatever our schema, using SQL and RDBMS technology to solve graph problems courts an impedance mismatch often seen in object-oriented systems backed by relational databases.

Impedance matching originates in the field of electrical engineering and is the process of matching source and load impedance to maximize power transfer and minimize signal reflection in the transmission line. The object-relational impedance mismatch is a metaphor applying when a relational database schema (source) is at odds with the target object model in object-oriented languages such as Java or Ruby (load), necessitating a mapping between the two (signal reflection/power loss). Object-relational mapping (ORM) frameworks such as Hibernate, MyBatis, and ActiveRecord attempt to abstract away this complexity. The downside is that each tool introduces a potentially long learning curve, possible performance concerns, and of course, bugs.

Ideally, we want to minimize the complexity and distraction that these extra layers can introduce and keep our solution close to our data model. This expressivity is a key driver behind a host of persistence solutions that operate outside of the relational model. The term NoSQL (Not Only SQL) has arisen over the last few years to cover systems oriented around alternative data models such as documents, key-value pairs, and graphs. Rather than discarding relational models, NoSQL solutions espouse polyglot persistence: Using multiple data models and persistence engines within an application in order to use the right tool at the right time.

Graph databases allow us to natively reason about our data in terms of vertices, edges, properties, and traversals across a network. Several popular graph databases on the market today include Titan, Neo4j, and OrientDB. Each of these solutions implements a Java interface known as Blueprints,[3] an Application Programming Interface (API) that provides a common abstraction layer on top of many popular graph databases and allows graph frameworks to interoperate across vendor implementations. Like most offerings in the NoSQL space, graph databases don't require a fixed schema, thus accommodating rapid prototyping and iterative development. Many graph database solutions are also open source, with commercial licensing and support available from the vendors.

Because graph databases don't support SQL, querying must be done through client frameworks and vendor-specific API's. Blueprints-compliant graph databases can use Gremlin,[4] a graph traversal framework built on the JVM that includes implementations in Groovy, Java, Scala, and Clojure. Let's use Gremlin to create our allocations graph and compare it to our relational solution.

Navigating the Graph with Gremlin

Using Gremlin, we can define the allocations for the three asset allocations we've explored. We can start exploring the graph using the Gremlin Groovy console and a TinkerGraph, an in-memory graph useful for experimental sessions:

```
g = new TinkerGraph()

==>tinkergraph[vertices:0 edges:0]
```

We then add vertices to the graph, passing in a Map of properties:

```
fs = g.addVertex([name:'File Server', type:'Asset'])
s  = g.addVertex([name:'Data Science', type:'Service'])
```

We then create a relationship between our file server and data science service:

```
allocation = g.addEdge(fs, s, 'allocated_to',
[percentage:0.1, effective_date:'2012-01-01'])
```

After adding our allocations, we can query the graph for paths. The equivalent of the first inner join from our relation schema from assets to services becomes a path to the head vertices (inV) of the outgoing edges (outE) from an asset vertex (g.v(0)):

```
gremlin> g.v(0).outE.inV.path

==>[v[1], e[6][1-allocated_to->3], v[3]]
```

Inspecting the properties on this path gives us:

3. *http://blueprints.tinkerpop.com*
4. *http://gremlin.tinkerpop.com*

```
==>{name=File Server, type=Asset}
==>{percentage=0.10, effective_date=2012-01-01}
==>{name=Data Science, type=Service}
```

To find all paths to cost centers, we can start the traversal with the collection of asset nodes and loop until we encounter vertices of type `cost center`:

```
g.V('type', 'Asset').as('allocated').outE.inV.loop('allocated')
    {it.loops < 6}{it.object.type == 'Cost Center'}.path

==>[v[0], e[2][0-allocated_to->1], v[1], e[6][1-allocated_to->3],
    v[3], e[7][3-allocated_to->4], v[4], e[8][4-allocated_to->5], v[5]]
```

This concise expression has the benefit of returning all allocation paths shorter than six edges away from the start of the traversal. If we follow the pattern above and add all of the allocation edges from our relational design, the query will return our three paths:

```
==>[v[0], e[2][0-allocated_to->1], v[1], e[6][1-allocated_to->3],
    v[3], e[7][3-allocated_to->4], v[4], e[8][4-allocated_to->5], v[5]]
==>[v[9], e[11][9-allocated_to->10], v[10]]
==>[v[9], e[12][9-allocated_to->5], v[5]]
```

Given these paths, it would be a simple matter to extract the allocation percentage property from each edge to obtain the final allocation from asset to cost center by applying a post-processing closure:[5]

```
g.V('type', 'Asset').as('allocated').outE.inV.loop('allocated'){it.loops < 6}
    {it.object.type == 'Cost Center'}.path{it.name}{it.percentage}

==>[File Server, 0.10, Data Science, 0.40, Search Engine, 0.70, R&D, 0.25,
    SAAS 202]
==>[Web Server, 0.20, IT 101]
==>[Web Server, 0.80, SAAS 202]
```

Finding Value in Network Properties

The graph-based solution exploits the ease of finding paths through a directed graph. In addition to the reporting concern we explored in which we sought out all paths between two points, pathfinding has useful applications in risk management, impact analysis, and optimization. But paths are really just the beginning of what networks can do. Graph databases and query languages like Gremlin allow for networks, "graphs that represent something real,"[6] to be included in modern application architectures.

5. *http://gremlin.tinkerpop.com/Path-Pattern*

6. Ted G. Lewis. 2009. *Network Science: Theory and Applications*. Wiley Publishing.

The field of network science is devoted to exploring more advanced structural properties and dynamics of networks. Applications in sociology include measuring authority and influence and predicting diffusion of information and ideas through crowds. In economics, network properties can be used to model systemic risk and predict how well the network can survive disruptions.

Going back to our allocations example, we might be interested in evaluating the relative importance of a service based on the profitability of the cost centers to which it eventually allocates out. Put another way, services that allocate out to the most profitable cost centers could be considered the most important in the business from a risk management perspective, since interruptions in one of these services could lead to disruption of the firm's most profitable lines of business. This perspective allows the allocation model to become data useful for operations, resource planning, and budgeting.

There are also many examples of businesses capitalizing on network properties. Facebook is powered by its Open Graph, the "people and the connections they have to everything they care about."[7] Facebook provides an API to access this social network and make it available for integration into other networked datasets.

On Twitter, the network structure resulting from friends and followers leads to recommendations of "Who to follow." On LinkedIn, network-based recommendations include "Jobs you may be interested in" and "Groups you may like." The recommendation engine hunch.com is built on a "Taste Graph" that "uses signals from around the Web to map members with their predicted affinity for products, services, other people, websites, or just about anything, and customizes recommended topics for them."[8]

A search on Google can be considered a type of recommendation about which of possibly millions of search hits are most relevant for a particular query. Google is a $100B company primarily because it was able to substantially improve the relevance of hits in a web search through the PageRank algorithm.[9] PageRank views the Web as a graph in which the vertices are web pages and the edges are the hyperlinks among them. A link implicitly conveys relevance. A linked document therefore has a higher chance of being relevant in a search if the incoming links are themselves highly linked and very relevant.

It's interesting to consider that PageRank is based on topological network properties. That is, the structure of the Web alone provides a predictive metric for how relevant a

7. *http://developers.facebook.com/docs/opengraph*

8. "eBay Acquires Recommendation Engine Hunch.com," *http://www.businesswire.com/news/home/20111121005831/en*

9. Brin, S.; Page, L. 1998. "The anatomy of a large-scale hypertextual Web search engine." Computer Networks and ISDN Systems 30: 107–117

given page will be, even before considering a web page's content. This is a perfect example of network structure providing value above and beyond the data in the source of the vertices. If you have related data that you don't explore as a network, you may be missing opportunities to learn something valuable.

Think in Terms of Multiple Data Models and Use the Right Tool for the Job

We've seen two models of how to solve a cost allocation problem. The relational model closely matched the domain, but added accidental complexity to an already inherently complicated problem. An alternative graphical model allowed us to dramatically simplify the representation of the data and easily extract the allocation paths we needed using open-source tools.

A big part of our jobs as data scientists and engineers is to manage complexity. Having multiple data models to work with will allow you to use the right tool for the job at hand. Graphs are a surprisingly useful abstraction for managing the inherent complexity in networks while introducing minimal accidental complexity.

Hopefully, the next time you encounter highly connected data, you'll recognize it as a network and you'll be better equipped to wring out all the valuable information it's hiding. Good luck in finding the value in your data…may your models serve you well and bring you more enlightenment than sorrow!

Acknowledgments

Thanks to my colleagues at Aurelius for critical feedback: Dr. Marko Rodriguez, Dr. Matthias Bröcheler, and Stephen Mallette. Thanks also to the entire TinkerPop community for supporting the graph database landscape with Blueprints, Gremlin, and the rest of the TinkerPop software stack. Thanks to my wife Sarah Aslanifar for early reviews and for AsciiDoc translation of the original manuscript. Last, but certainly not least, thanks to Q and the O'Reilly team for making *Bad Data Handbook* happen!

CHAPTER 14

Myths of Cloud Computing

Steve Francia

Myths are an important and natural part of the emergence of any new technology, product, or idea as identified by the hype cycle. Like any myth, technology myths originate in a variety of ways, each revealing intriguing aspects of the human psyche. Some myths come from early adopters, whose naive excitement and need to defend their higher risk decision introduce hopeful, yet mistaken myths. Others come from vendors who, with eagerness, over-promise to their customers. By picking apart some of the more prominent myths surrounding the cloud, we gain better understanding of not only this technology, but hopefully the broader ability to discern truth from hype.

Introduction to the Cloud

In some ways, cloud computing myths are easily among the most pervasive of all technology myths. Myths about the cloud are quickly perpetuated through a blend of ambiguity of what "the cloud" actually means and the excitement surrounding the hype of a new technology promising to be the solution to all our problems. As the hype around "the cloud" grows, each new vendor adopts that term while simultaneously redefining it to fit their product offerings.

What Is "The Cloud"?

For the purposes of this text, we will be using the term "the cloud" to refer to virtualized nodes on elastic demand as provided by vendors like Amazon's EC2, Rackspace, Microsoft Azurel, Joyent, and more. Even with this somewhat restricting definition, there are significant differences between the different vendors.

The Cloud and Big Data

You may be wondering what cloud computing has to do with big data. A significant percentage of companies today are using cloud computing and that number is increasing daily. While some positions exist where a data scientist can leave things completely to an infrastructure team, in many jobs they may be responsible for the infrastructure. In a startup, it's quite likely, at least to some degree. In all jobs, some knowledge and awareness of infrastructure strengths and best practices would benefit the diligent data scientist. It's natural for someone to think that the infrastructure isn't her problem; but in a smaller firm, a data scientist may have to make decisions about storage and when the data disappears, it's everyone's problem. The hope is that through understanding these myths and through them the strength of cloud computing, the astute data practitioner will be able to leverage the cloud to be more productive while avoiding disasters along the way.

I'm going to take a slightly different approach from the sections you have already read. Rather than sharing a single experience, I'll be sharing with you many experiences to which I've been privy courtesy of working for 10gen. 10gen develops and supports MongoDB and as a result, we benefit from sharing experiences with our customers, many of whom are on the cloud. In order to protect their privacy, I have taken some isolated experiences and woven them into a cohesive story about a fictional startup. To quote Dragnet, "The story you are about to read is true; only the names have been changed to protect the innocent…." I've also done my best to abstract the specific technologies and vendors used to their root principles, as the experiences included could have easily been had across any of the cloud vendors.

Introducing Fred

The central character in our story is Fred. Fred is the CTO of a six-month old data driven fictitious social startup called ProdigiousData. He and his team have finished their initial prototype and are about to launch the product. They recognize that while their immediate needs are small, with a small degree of success they will have big data needs very soon. Fred decides that they will launch their product on the cloud due to its easy ability to scale to handle their big data needs. Fred, and more importantly his CFO, are excited about the low cost that the cloud provides, especially without needing to purchase anything up front.

At First Everything Is Great

After months of preparation, sweat, and lots of coffee, the launch happens and it's an immediate success. All the important things are happening just right. User growth is steadily increasing and more importantly people seem to really like the product. The system they have designed is quite capable of handling the load. Fred and his team couldn't be happier.

They Put 100% of Their Infrastructure in the Cloud

Under the time and pressure constraints surrounding a startup that hadn't yet launched, they decided to go with a fairly simple and straightforward infrastructure. They are using all cloud-based machines and services, from the load balancer and firewall to the database and data processing nodes. The current makeup is two smaller machines running software load balancers in an active-passive configuration. The load balancers distribute requests to three application nodes. At the back, they have a pair of database nodes configured in a master-slave setup. They feel they have eliminated any single points of failure and their virtual cluster is optimally utilized.

As Things Grow, They Scale Easily at First

As the load increases, they are able to stand up another app node with ease. They simply clone an existing node that is running and within minutes they have additional capacity. This is the horizontal scalability that they were expecting.

Then Things Start Having Trouble

A couple of weeks go by before they have their first blip. It's manifesting in some users getting timeouts. It's pretty irregular, but it definitely has Fred worried. The fact that they haven't yet set up a sophisticated monitoring system keeps him up at night, but with only a handful of machines, it never seemed like much of a priority. Upon inspecting the application logs, they discover the problem is the application is hanging on database operations. The CPU on the database machine is working around the clock. Load is in the double digits. They reboot the database and boost the specs on the virtual node on which it's running. They jump up to the largest size available, increasing the number of cores and RAM on the virtual machine. While the team thinks the problem is solved, Fred knows better. He knows that they haven't solved anything, but simply delayed the inevitable. At some point, that larger database node will also reach a point of saturation and at the rate their load is increasing, it's going to happen soon.

They Need to Improve Performance

In an effort to delay this even further, they begin to optimize their database. While their dataset is growing in size, it's growing in use far more. They are doing much more read and writes than they expected at this stage. They need to find some way to increase the database performance. Running `iostat` on the database nodes is very telling. Their IO performance is poor and seek times are worse. They've gone with a popular cloud provider that has ephemeral local storage. Data persistence is achieved via networked storage. As a result, durable block stores in the cloud will have slower performance and less predictable throughput when compared to a local disk.

Higher IO Becomes Critical

Fred's no rookie. He knows that to increase IO performance you either need faster drives or more of them. Since his provider only has one tier of drives, they go with attaching 4 volumes configured in RAID 10. RAID 10 gives them the best of both worlds, providing double read and write performance and full redundancy. After the change is made to both database nodes things stabilize for the most part. Now that the fire is out they set up a more sophisticated monitoring system, one that not only provides better diagnostics into what is happening by tracking stats and graphing them over time, but also alerts when certain conditions and thresholds are met. It's a lot of work, but they've gotten a wake up call from this initial scare that they have been flying blind and and they won't likely be this lucky next time.

A Major Regional Outage Causes Massive Downtime

Seemingly out of nowhere, disaster strikes. A regional outage occurs with their cloud provider. They are completely offline and it provides them little comfort to know that it's not just them, but also some other fairly notable websites. Hours go by without any information other than the occasional status update from their vendor. Ten hours later the provider is back online. This is a very short-lived victory, for it is only when they try to bring their machines back online do they realize their disaster has just begun. They haven't yet automated the build of each machine and now isn't the time to do it. Because each machine was ephemeral, with this full power outage they lost all setups and are more or less starting from scratch. They manually configure each machine. First the app servers and then the database. Luckily the data is there, but the database won't start. It's complaining that it shut down uncleanly (duh) and the data files need to be repaired. After a lengthy repair, it looks like all their data is there, and 21 coffee-filled hours later they are back online. They have learned that managing nodes in the cloud requires a lot of work and that automation is essential. While an outage could have just as easily happened at a data center, there is no question that if they had an account at a data center

they would have had more feedback from their account manager. A dedicated data center would be working with them to bring their machines back online and of course every host provides persistent storage, so the restoration of their infrastructure would be trivial. They certainly wouldn't have needed to rebuild all their nodes.

Higher IO Comes with a Cost

In the weeks that follow, no real issues occur. With all that has happened, they aren't taking chances. They are keeping a close watch on their infrastructure, especially the database. With RAID 10 and monitoring in place, they know that for the most part they are in good shape. Over time they begin to notice some strange behaviors and they struggle to explain it. It seems that overall performance has increased dramatically but one particular operation has actually degraded. The nightly bulk import is actually taking longer than it did before, even though the data imported is relatively the same. After crunching a bunch of data in an external system, they load the data in a large batch. This behavior is contradictory to all their expectations and they struggle to make sense of what's happening. Google searches produce some forum answers, but no clear explanation emerges. They logically think that due to the somewhat random high latency present on these multitenancy network drives that by doubling the output and bandwidth they would be hedging against these issues. After spending a lot of time trying to diagnose, including trying to launch their cluster on a different region, they eventually abandon their pursuit, assuming it's either an issue with their monitoring or a problem without a solution. They accept this degradation of performance because overall performance has increased. Most operations have improved in measurable and expected ways.

Data Sizes Increase

As user growth increases, the data increases even faster. They proactively realize that they will need to partition their data across multiple machines as they can not sustain growth on one pair of servers. They knew this all along, in fact they planned on it, but with the recent outage they have adjusted their approach.

Geo Redundancy Becomes a Priority

One of the nice features of the cloud is that cloud providers seamlessly provide hardware/services in many regions or availability zones. ProdigiousData realizes that to achieve their desired uptime, they need to be in at least two zones. They now have leveraged *chef* to be able to quickly create nodes.[1] They can easily create load balancers and app

1. *www.opscode.com/chef*

servers in the new region as they are predominantly stateless, but what about the database servers? How do they replicate or partition the data effectively? They do a quick test and realize that there is about a 0.250 ms latency between the two different regions and at times it's considerably higher.

Horizontal Scale Isn't as Easy as They Hoped

They come to the realization that this is going to be a lot of work. While the stateless application nodes scale effortlessly, the stateful database nodes are far less portable. Even though their underlying database technology makes it quite easy to add more nodes to the database cluster, each new node begins empty. Data needs to be migrated from existing nodes to the new one. Beyond that, they need to worry about where to place the nodes for maximum performance and minimum downtime. Fred concedes that for their business needs, they can survive with slightly stale data as caused from replicating over the wan from nodes in one region to nodes in another. They place some app nodes into each of the two locations and database nodes in each, but in a creative way. The data is partitioned into geographical regions and then evenly distributed across the different nodes in each region. Each primary node (the one accepting writes) then replicates to two different nodes, one local to that region and one in the other region. The application writes and reads to the locally writing database and reads from the stale local data that was replicated from the other region. Setting all of this up in a automated fashion was a big task that took weeks. Unfortunately, software doesn't currently exist to both set up and coordinate efforts across many different nodes playing different roles in a cluster.

Costs Increase Dramatically

It wasn't easy, but they got there. They now have a pretty scalable application running in the cloud. It has multiple location redundancy and is even optimized to route users to two different availability zones depending on their location. Everything is going well…well, until Fred gets the bill for the month. Something must be wrong. He never paid this much in a month with his dedicated colocation hosting company. Fred began to think of all the things they had added. Multiple locations, three copies of each node, six copies of all pieces of data (2 per RAID 10 configuration x 3 replicated database nodes). They also maxed out the configuration on each of those nodes. He began to do a cost analysis against an old statement. He discovered that when running on the cloud, it often required more nodes and resources to achieve similar performance. While the cost per node was often cheaper, cloud nodes and a server running on hardware customized for a task were not the same.

Fred's Follies

Does Fred's story sound familiar? Perhaps it reminds you of your own. What myths did Fred fall prey to and how can you best learn from Fred's follies to avoid these pitfalls yourself?

Myth 1: Cloud Is a Great Solution for All Infrastructure Components

For a few years now, the cloud has been billed as the future of infrastructure. Marketers use such language as "Enterprise data centers will be largely replaced by cloud computing within 20 years" and "Public cloud computing offers many incredible possibilities, like the prospect of doing supercomputer-level processing on demand and at an incredibly low cost."[2] Consumers have largely been taken in. They view the cloud as an easy solution for all their infrastructure needs. In a world where software is able to emulate nearly any hardware, people often are using the cloud for everything because they can, often with blissful ignorance. The lure of being able to stand up a load balancer, firewall, or RAID controller without any expensive hardware entices many.

How This Myth Relates to Fred's Story

Another place where virtualized nodes can't come close to the performance and functionality of dedicated hardware are load balancers. While expensive up front, an F5 or Stingray will produce amazing results and are much easier to configure than any purely software tool. Both have fully redundant options and work well when distributing load across multiple locations. It is true that some vendors provide some load balancing offerings, but none offer the flexibility or performance that one can obtain through hardware. Additionally, they are all designed to load balance between the Internet and your cluster, but not for uses within the cluster.

Myth 2: Cloud Will Save Us Money

Let me begin this section by stating clearly that using the cloud effectively can result in cost savings. Additionally, from a purely financial perspective, when using the cloud instead of your own equipment, the expense switches from a capital expenditure (CapEx) into a more flexible operational expenditure (OpEx). This appeals to many CFOs, which makes the CTO/CIO look like a financial genius. While this doesn't make sense in every situation as there are still some times when years of CapEx tax depreciation are preferred, for a majority of companies OpEx is preferred over CapEx.

2. *http://www.informationweek.com/news/galleries/cloud-computing/infrastructure/232901167*

To better illustrate this point I'd like to use a simple analogy. There are three common ways to obtain a car. You can rent, lease, or buy. Each has its own place and to many people, it's pretty straightforward which one makes the most sense for their situation. If you are in a place for a few days, renting a car makes the most sense. If you want to reduce your upfront cost and intend to use a car longer term, then leasing makes a lot of sense. If you are using a car longer term and want to completely customize it for your situation, purchasing makes the most sense. The world of computing presents these same three options. You can rent (cloud computing), lease (managed hosting), or purchase (colocation/data center). Similar logic applies here. If you intend a node for long-term use, purchasing will save you significantly over renting.

One place where this analogy fails is that in cloud computing, you aren't given a lot of choices. You can increase the number of CPUs and amount of RAM independently of one another. While in time, it's quite possible that more levers will be made available by vendors, historically, options have been quite limited. Choices have been boiled down to the most simple terms such as small, medium, and large. In our analogy, it's more like you can rent any car as long as it's a sedan. If you need to carry more than five people, you can rent two. You can rent a fast sedan or an economic one, but they are all sedans.

Often, running application servers that can easily adjust to needs by scaling up or down the number of nodes can produce real savings. But savings require you to manage and adjust the number of nodes appropriately.

For many uses, you could drastically reduce the cost by optimizing the hardware for your needs. One such case is with databases. On the cloud, the value (and performance) is often just not there. Databases benefit heavily from high IO with low random seek times. SSDs in a RAID 10 configuration powered by a reliable high performance hardware RAID controller will produce amazing results untouchable with any cloud configuration. What's worse here is that to achieve acceptable performance on the cloud, you'll end up spending a lot more on faster drives and more nodes, and consequently more replicated nodes for high availability.

This principle applies to all databases, relational and nonrelational alike. Some of the newer databases like MongoDB have been designed with the cloud in mind. In the case of MongoDB, the database utilizes memory mapped files as a cache to alleviate many read and write operations from the slower IO present on cloud. This is an extremely beneficial improvement, but eventually all data needs to be read from and written to disk. As a consequence of this memory management style, money spent for the purchase of hardware to be used with MongoDB is better spent on additional memory than on more powerful CPUs. Unfortunately, with the cloud you can't adjust RAM independently from CPU, and most software and services don't have a linear relationship between their processing and memory needs.

How This Myth Relates to Fred's Story

While the cost savings up front were significant, over time they found themselves spending more and more. When you factor in the overhead of managing more nodes than needed it's impossible to make the claim that they saved money by running on the cloud. Fred, like many CTOs, flew under the radar on this one. The reality is that very few people are watching this closely enough and doing a cost comparison. Like boiling a frog, the temperature slowly rises and before you know it, you're cooked. Sometimes the switch from a CapEx to an OpEx itself is enough of a selling point to justify the cost.

Myth 3: Cloud IO Performance Can Be Improved to Acceptable Levels Through Software RAID

Perhaps this isn't a myth as much as a misunderstood behavior. As mentioned in the previous myths, people put a lot of trust in software being capable of replacing hardware. And to some degree it can, from the perspective of meeting the minimum feature requirements. In some areas of computing, we have gone over completely to software-based solutions. It used to be that sound processing was done completely by add-in sound cards. I remember bragging about my SoundBlaster AWE64, which meant I could play 64 simultaneous MIDI instruments, but more importantly to me, any game on the market. While a few sound cards are still on the market, they have nearly all been replaced by software-based solutions with negligible impact on performance. It would seem like RAID would be in the same boat. It's a fairly low-level, but simple function. Linux's `md` feature provides virtually all the RAID functionality present from the various hardware vendors.

How This Myth Relates to Fred's Story

In our story, Fred and the ProdigiousData team fell prey to this myth. They put all of their infrastructure on virtualized nodes, using software-based solutions for many things that would have previously been done using hardware.

Unfortunately in many circumstances, hardware simply trumps software. Recall in the story that odd decrease in behavior when they increased their IO by switching from one drive to four in a RAID 10 configuration. They simply gave up trying to solve it, assuming it was just a fluke. It wasn't a fluke; it's easily isolatable and repeatable. It occurs as a result of Linux `md` (multi disk) functionality falling far short of the performance achievable by a good RAID controller. One area where this is quite noticeable is in random-write performance, which is quite poor. Similar things happen throughout computing. Whether discussing hardware or software, we often make the trade-off between the

convenience of virtualized or interpreted over the performance of native. For example, compare dynamic languages to complied languages, or iPhone apps (native code) to Android apps (virtual machine). In cases where performance matters, it's often an expensive mistake to make this compromise.

See *http://engineering.foursquare.com/2012/06/20/stability-in-the-midst-of-chaos/*

Myth 4: Cloud Computing Makes Horizontal Scaling Easy

Dr. Werner Vogel, Amazon's CTO, explained that building horizontally scaling, reliable geo-redundant systems on cloud platforms "becomes relatively easy." Of all the myths presented here, this is the most pervasive one about the cloud.

It is a common misconception that you can simply deploy your application to the cloud and it "just works." It is commonly believed that the cloud eliminates the need for (careful) planning on how to scale an application. Geographic redundancy and 24/7 global access is easy because they are able to fire up nodes in multiple data centers. This may be the vision of the future, but it's certainly not the present. Without careful planning and the appropriate infrastructure, there are common factors among cloud providers that make horizontal scaling even more difficult.

How This Myth Relates to Fred's Story

The ProdigiousData group learned this through many hard experiences. When all was said and done, they had a pretty robust (and complicated) infrastructure, and they earned it. They invested heavily into each solution and experienced their share of downtimes and sleepless nights getting there. After all their experiences, they didn't regret what they had done, but wondered if there was a better way.

The fact that there is a cottage industry around making the cloud manageable should itself dispel this very persistent myth, and yet to no avail. Perhaps the reason this myth is so prevalent is that some myths drive us. Great accomplishments are often driven by great visions. These optimistic goals define the future. Perhaps this myth isn't a false tale as much as a vision we are all hoping comes to pass. It's conceivable that in the near future, advancements in virtualization, operating systems, data storage, data processing, and the glue tying it all together will make this myth true. Of all of the above myths, this one may be a vision that may actually come to fruition.

Conclusion and Recommendations

The idea to treat computing as a utility, commodified and available at the turn of a switch, has revolutionized the industry. The gamble Amazon made years ago has been the most significant advancement to computer infrastructure in the last decade. As wonderful as this advancement is, it's not the end of all computing. While cloud com-

puting has its obvious benefits, it's still relatively early in its development, with rapid advancements from various vendors all trying to one-up each other complicating the market. Not only is it a rapidly emerging technology, but every technology has its sweet spot. There is great power that comes from using the right tool for the job. Cloud computing is fantastic for stateless, process heavy jobs, such as most application servers. The cloud has historically been weaker at jobs where state matters. Data processing typically falls in the middle of these two. For me, the ideal infrastructure would include the best of both worlds: easy management of stateful machines running on optimized hardware connected via LAN to commoditized cloud nodes for application processing. It's important to recognize that these cloud offerings are still infants in their life cycles. In time, as offerings develop and improve, it's likely that the cloud's current weaknesses will be erased and it will become an increasingly viable solution.

CHAPTER 15

The Dark Side of Data Science

Marck Vaisman

More often than not, data scientists hit roadblocks that do not necessarily arise from problems with data itself, but from organizational and technical issues. This chapter focuses on some of these issues and provides practical advice on dealing with them, both from human and technical perspectives. The anecdotes and examples in this chapter are drawn from real-world experiences working with many clients over the last five years and helping them overcome many of these challenges.

Although the ideas that are presented in this chapter are not new, the main purpose is to highlight common pitfalls that can derail analytical efforts. When put into context, these guidelines will help both data scientists and organizations be successful.

Avoid These Pitfalls

The subject of running a successful analytics organization has been explored in the past. There are many books, articles, and opinions written about it and this will not be addressed here. However, if you would like to be successful in executing and/or managing analytical efforts within your organization, you should *not* heed the "commandments" listed below.

I. Know nothing about thy data
II. Thou shalt provide your data scientists with a single tool for all tasks
III. Thou shalt analyze for analysis' sake only
IV. Thou shalt compartmentalize learnings
V. Thou shalt expect omnipotence from data scientists

These commandments attempt to cluster-related ideas, which I will explore in the following sections. If you do choose to obey one or more of these commandments—which we've explicitly warned you *not* to—you will most likely head down the path of not achieving your goals.

Know Nothing About Thy Data

You have to know your data, period. This cannot be stressed enough. Real world data is messy and dirty; that is a fact. Regardless of how messy or dirty your data is, you need to understand all of its nuances. You need to understand the metadata about the data. If your data is dirty, know that. If there are missing values, know that, and know why they are missing. If you have multiple sources with different formatting, know that.

Knowing thy data is a crucial step in a successful analysis effort. Time spent up-front understanding all of the nuances and intricacies of the data is time well spent. The rule of thumb says that 80% of time spent in analytics projects is cleaning, munging, transforming, and so on; so the more you know about the data, the less time you'll spend in these tasks.

In the following sections, we'll highlight some examples of times when organizations knew about their data, but did not now enough. (In our experience, knowing nothing about the data is the exception. Usually people do have some level of knowledge.)

Be Inconsistent in Cleaning and Organizing the Data

The first step in an analysis effort is to establish consistent processes to clean the data. This way, other people know what to expect when they work with it.

Case study: A client had defined several processes to move, clean, and archive the data into tab-delimited flat files, which in turn served as the source data for many other analytics efforts. Most analyses of this data seemed to work well, but one particular case yielded unexpected results. A visual inspection didn't reveal any obvious flaws in the data, but a closer investigation revealed that the troublesome files used a mix of spaces and tabs as delimiters. The culprit? The process that had generated these files had inserted some unexpected spaces instead of tabs.

Assume Data Is Correct and Complete

A common pitfall is to assume that you're working with correct and complete data. Usually, a round of simple checks—counting records, aggregating totals, plotting, and comparing to known quantities—will reveal any problems. If your data is incorrect or incomplete, you need to know that, so your decisions can take that fact into account.

Case study: A client had created a data product that calculated certain metrics on a daily basis, over a period of time. All production processes were running without issue and

the consumers of the data were using the data provided without question. The size of the processed daily datasets was in the order of hundreds of millions of data points. Certain decisions of large economic impact were made based on the results of these data products.

The client later launched a separate research investigation, using the same datasets, to try to understand if and how the daily distributions changed over time. This project yielded some unexpected results, raising the question of whether the data was correct and complete during certain time periods. Further analysis revealed that, during those time periods, about twenty percent of the data was missing. This means that the client had made (wrong) decisions, simply because no one knew that data was incomplete! Simple summaries, made early on, would have indicated missing data.

Spillover of Time-Bound Data

It's common that organizations partition their data by some time interval—such as day, hour, or minute—and then organize the files in directories named accordingly. For example, given a directory called `data_20120706`, one could reasonably expect that every file therein would hold data somehow related to July 6, 2012.

In my experience, though, this doesn't always hold true. Instead, many projects exhibit "spillover," which is a nice way of saying that a path for one time interval contains data from other intervals. In this example it would mean that the directory `data_20120706` would also contain data from July 5, 2012, or July 7, 2012.

Spillover can happen for any number of reasons, including inadequate accuracy in the partitioning scheme. While you may not be able to completely eliminate spillover, you can at least be aware of it. Don't expect that the data is partitioned perfectly.

Thou Shalt Provide Your Data Scientists with a Single Tool for All Tasks

There is no single tool that allows you to perform all of your data science tasks. Many different tools exist, and each tool has a specific purpose. In order to be successful, data scientists should have access to the tools they need and also the ability to configure these tools as needed—at least in a research and development (R&D) environment—without having to jump through hoops to do their work. Providing one fixed tool (or set of tools) to perform all tasks is unrealistic and unreasonable.

Using a Production Environment for Ad-Hoc Analysis

The use cases of performing exploratory analysis or any other data R&D effort are very different than the use cases for running production analytics processes. Generally, the design of production systems specify that they have to meet certain service level agree-

ments (SLAs), such as for uptime (availability) and speed. These systems are maintained by an operations or devops teams, and are usually locked down, have very tight user space quotas, and may be located in self-contained environments for protection. The production processes that run on these systems are clearly defined, consistent, repeatable, and reliable.

In contrast, the process of performing ad-hoc analytical tasks is nonlinear, error-prone, and usually requires tools that are in varying states of development, especially when using open source software. Using a production environment for ad-hoc and exploratory data science work is inefficient because of the limitations described above.

Case study: A client had two Hadoop clusters, one for R&D and the other for production. All R&D work was performed on the R&D cluster, which included an ample set of tools (R, Python, Pig, and Hive, among others). However, the R&D cluster was managed as a production system: R&D users did not have administrative privileges. This meant that even though users could run jobs, the R or Python streaming scripts were limited to only using the core libraries. Therefore, it took more time to develop analysis jobs because users had to implement "creative" solutions to work around these limitations.

One such workaround, to use special R or Python libraries, involved moving data out of the Hadoop cluster to a separate machine where analysts had administrative access. The entire process was cumbersome and added unnecessary time and headaches to a project.

You need to carefully plan out the architecture, configuration, and administration of your tools and environments. The use cases are different, and therefore the management and operation of the system should be different as well.

I certainly don't advocate that all users or data scientists have administrative privileges and do as they please; but they should have enough privileges to set up the environment to suit their analytics needs. If this is not possible, it would make sense to have the operations teams and the analytics users work in partnership and devise workable solutions.

The Ideal Data Science Environment

At the time of this writing, it is relatively easy and inexpensive to set up an ideal data science environment. With low hardware costs (processors, storage, and memory) and the ease of access to cloud computing resources (Amazon EC2, Rackspace, or other cloud services), organizations should get the tools their data scientists need and let them manage the tools as they need to as mentioned before.

At a minimum, I recommend you set up one or more multi-core analytics machines running Linux, with lots of RAM and ample storage space. I recommend Linux as an operating system because most analytics tools and programming languages are designed with Linux in mind (e.g., Hadoop), and many external libraries run on Linux only.

If you have a cluster, you should try to have your analytics machine within the same environment as your cluster, especially if you store data in a distributed file system such as Hadoop's HDFS. Data scientists should have some level of administrative rights so they can compile and install libraries as needed.

Setting up the right environment is not only a matter of using the right tools, but also of having the right organizational mindset. Ideally, a data science environment is not used for other development purposes so data scientists can take advantage of all available resources for their needs, especially when running large scale analysis in a parallel fashion.

Thou Shalt Analyze for Analysis' Sake Only

There are many kind of analytical exercises you can do. Some begin as an exploration without a specific question in mind; but it could be argued that even when exploring, there are some questions in mind that are not formulated. Other exercises begin with a specific question in mind, and end up answering another question. Regardless, before you embark on a research investigation, you should have some idea of where you are going. You need to be practical and know when to stop and move onto something else. But again, start with some end in mind. Just because you have a lot of data does not mean you have to do analysis just for analysis' sake. Very often, this kind of approach ends with wasted time and no results.

Additionally, before you embark on an data science project, you should assess your analytics readiness level. This assessment will help set an end goal and avoid digging into a rabbit hole. Understanding where you fall in this readiness spectrum will help set priorities and define an end goal. Some of the possible readiness levels are:

- We don't even know where to begin.
- We don't know what we have, and we've never done any analysis before.
- We have an idea of what we have, but we've never done any analysis before.
- We know what we have, and we've tried answering specific questions, but we're stuck.

Case study: A Fortune 500 technology company had a process that generated data based on day-to-day operations. Historically, their data warehousing team performed most analytics tasks, such as traditional Business Intelligence (BI). This team's primary responsibility was to develop relational database-driven tools, though they had also been dabbling in less-conventional analytics projects.

They decided to bring in a data scientist to help with the latter endeavor. The belief was that the data scientist would magically find a golden nugget hidden in their data, which

they could easily translate into some results. Management's directive was, quite simply: "Go and find me the value in the data!" They did not follow up with any context or direction; they simply set an unrealistic expectation that a wizard was off to do some magic and return with all the answers.

The proper response in this case—and the one the data scientist provided, mind you—was to object to this directive and ask for direction, get context, and set the parameters for engaging in an analytics exercise. Part of a data scientist's role and value is to help the organization ask the right questions, and steer clear of unnecessary work.

Thou Shalt Compartmentalize Learnings

This commandment is pretty straightforward. The idea here is that, as an organization, you should share your knowledge. This is especially important when analytical efforts are performed by different areas throughout the company.

Share your findings. If you are doing analysis and you find something related to any of the pitfalls mentioned previously—whether you find missing data, formatting errors, shortcuts to get things done—share them. If you've already processed data into aggregates for some reason and you think that could be useful for other analyses, share it. When you finish a body of work, share the findings (code, results, charts, or other documentation). Document your assumptions. Document your code. Have informal gatherings to share and discuss.

It is amazing, especially in large organizations, when you spend time working on something and you consult with colleagues in other areas, how often you hear: "oh, yes, we looked at that a while ago and we have some results in a file somewhere." The amount of time (which usually translates into economic value) saved by sharing can be quite large. By sharing your knowledge, you are also contributing to the learning across the organization.

Thou Shalt Expect Omnipotence from Data Scientists

Data scientists come in all shapes, sizes, and colors, and hail from traditional and unusual career paths. They blend skills in programming, mathematics, statistics, computer science, business, and machine learning, among others. Above all, great data scientists are very curious about everything and have broad knowledge across many different domains.

The current availability of computing power, analytically focused programming languages (such as R and Python), and parallel computing frameworks (such as Hadoop)

allows data scientists to be very effective. Data scientists are able to perform many tasks across the analytics spectrum, from spinning up a cluster in the cloud, to understanding the subtleties and trade-offs of different technologies and system quirks, to running a clustering algorithm and building predictive models.

Omnipotence is defined as having unlimited powers and there seems to be an expectation that data scientists can and should do it all. While they are willing and able to work on many tasks across the data science process, from munging and modeling to visualizing and presenting, it's quite rare to find talent with extensive experience in *all* aspects of data science.

Organizations and managers would do well to adjust their expectations accordingly. A successful data science function is made up not by one person, but at a minimum two or three individuals whose broad skills have much overlap while their unique expertise does not.

Where Do Data Scientists Live Within the Organization?

Finding a place for data scientists can be a bit tricky. Sometimes you'll find them living within an engineering organization, sometimes within a product organization, sometimes within a research organization, and other times they live under some other umbrella or on their own. However, wherever data scientists live in your organization, make sure there is unified guidance and management that understands using data science as an asset.

Final Thoughts

I hope that this chapter's case studies helped you think about the higher-level organizational issues that may arise in an analytics effort. If you want success in your data science endeavors, please heed my advice and do *not* follow the commandments I've outlined here.

It is hard to quantify the impact of the pitfalls outlined in this chapter. Suffice it to say that the impact is usually economic, and in some cases can be quite large.

CHAPTER 16

How to Feed and Care for Your Machine-Learning Experts

Pete Warden

Machine learning is a craft as well as a science, and for the best results you'll often need to turn to experienced specialists. Not every team has enough interesting problems to justify a full-time machine-learning position, though. As the value of the approach becomes better-known, the demand for part-time or project-based machine-learning work has grown, but it's often hard for a traditional engineering team to effectively work with outside experts in the field. I'm going to talk about some of the things I learned while running an outsourced project through Kaggle,[1] a community of thousands of researchers who participate in data competitions modeled on the Netflix Prize. This was an extreme example of outsourcing: we literally handed over a dataset, a short description, and a success metric to a large group of strangers. It had almost none of the traditional interactions you'd expect, but it did teach me valuable lessons that apply to any interactions with machine-learning specialists.

Define the Problem

My company Jetpac creates a travel magazine written by your friends, using vacation photos they've shared with you on Facebook and other social services. The average user has had over two hundred thousand pictures shared with them, so we have a lot to choose from. Unfortunately, many of them are not very good, at least for our purposes. When we showed people our prototype, most would be turned off by the poor quality of the images, so we knew we needed a solution that would help us pick out the most beautiful from the deep pool to which we had access.

1. "Kaggle: making data science a sport." (*http://www.kaggle.com/*)

This intention was too vague to build on, though. We had to decide exactly what we wanted our process to produce, and hence what question we wanted to answer about each photo. This is the crucial first step for any machine-learning application. Unless you know what your goal is, you'll never be able to build a successful solution. One of the key qualities you need is some degree of objectivity and repeatability. If, like me, you're trying to predict a human's reaction, it's no good if it varies unpredictably from person to person. Asking "Is this a good flavor of ice cream?" might be a good market research question, but the result says more about an individual's personal taste than an objective "goodness" quality of the dessert.

That drove us to ask specifically, "Does this photo inspire you to travel to the place shown?" We were able to confirm that the answers for individual photos were quite consistent across the handful of people we ran initial tests on, so it appeared that there was some rough agreement on criteria. That proved we could use one person's opinion as a good predictor of what other people would think. Therefore, if we could mimic a person with machine learning, we should be able to pick photos that many people would like.

As with most software engineering disciplines, the hardest part of a machine-learning expert's job is defining the requirements in enough detail to implement a successful solution. You're the domain expert, so in the end you're the only one who can define exactly what problem you want to solve. Spend a lot of time on this stage; everything else depends on it.

Fake It Before You Make It

Because the problem definition is so crucial, you should prototype what would happen if you could get good answers to the question you're asking. Does it actually achieve the goals you set when you try it within your application? In our case, we could determine that by using a human to rate a handful of users' photos, and then present the end result within our interface to see how it changed the experience. The results were extremely positive: users went from a lukewarm reaction to enthusiasm when the first photos they saw were striking and inspiring. From a technical standpoint, it helped us, too, because it ensured we had set up the right modular interface so that the quality algorithm could be a true black box within our system, with completely-defined inputs and outputs.

For almost any machine-learning problem, you can build a similar human-powered prototype at the cost of some sweat and time. It will teach you crucial lessons about what you need from the eventual algorithm, and help you to be much smarter about steering the rest of the process. A good analogy is the use of static "slideware" presentations of an application to get initial feedback from users. It gives people a chance to experience something close to the result for which you're aiming, at a stage when it's very easy to change.

In practice, the system we set up worked very simply. We'd ask a user to sign up on the site, and we'd send them an email telling them that their slideshow would be ready soon. We loaded information about all the photos in their social circle into our database, and then added them to a queue. We had a photo-rating page that team-members could access that displayed a single photo from the person at the head of the queue, and had two buttons for indicating that the photo was inspiring, or not so much. Choosing one would refresh the page and show the next unrated photo, and if multiple people were rating, the workload would be spread out among them. This allowed us to rapidly work through the tens of thousands of photos that were shared with an average user, and we often had their results back within an hour. It meant that when we sat down with advisors or potential investors for a meeting, they could sign up at the start and we could show them a complete experience by the end of the discussion. That helped convince people that the final product would be fun and interesting if we could solve the quality issue, and enabled them to give us meaningful advice about the rest of the application before that was done.

Create a Training Set

Prototyping naturally led to this next stage. Before we could build any algorithm, we needed a good sample of how people reacted to different photos, both to teach the machine-learning system and to check its results. We also wanted to continue testing other parts of our application while we worked on the machine-learning solution, so we ended up building a human-powered module that both produced a training set, and manually rated the photos for our early users. This might sound like a classic Mechanical Turk[2] problem, but we were unable to use Amazon's service, or other similar offerings, because the photos weren't public. To protect people's privacy, we needed to restrict access to a small group of vetted people who had signed nondisclosure agreements and would work as our employees. We did experiment with hiring a few people in the Philippines, but discovered that the cultural gap between them and our user population was too large. Situations like conferences and graduations were obviously inspiring to us and our target users in the US, but weren't recognizable as such by our overseas hires. We could no doubt have fixed this with enough training and more detailed guidelines, but it proved easier to tackle it internally instead.

We already had millions of photos available to analyze, and we wanted to build as big a training set as possible, because we knew that would increase the accuracy of any solution. To help us rate as many photos as possible, I built a web interface that displayed a grid of nine photos from a single album, and let us mark the set as good or bad and advance to the next group with one keystroke. Initially these were *y* and *n*, but we found that switching to , and . helped improve our speed. The whole team joined in, and we

2. Amazon Mechanical Turk: "Artificial Artificial Intelligence." (*https://www.mturk.com/*)

spent hours each day on the task. We had all rated tens of thousands of photos by the time we stopped, but our community manager Cathrine Lindblom had a real gift for it, notching up multiple days of more than fifteen thousand votes, which works out to a sustained rate of one every two seconds!

One of the decisions that was implicit in our interface was that we'd be rating entire albums, rather than individual photos. This was an optimization that seemed acceptable because we'd noticed anecdotally that the quality of photos seemed pretty consistent within a single album, and that a selection of nine pictures at once was enough to get a good idea of that overall quality. It did constrain any algorithm that we produced to only rating entire albums, though.

To ensure that we were being consistent, we kept an eye on the accepted percentage rate. On average, about 30% of all photos were rated as good, so if anyone's personal rating average was more than a few percentage points above or below that, they knew they were being too strict or lenient. I also considered collecting multiple ratings on some photos, to make sure we were really being consistent, but when I did a manual inspection of our results, it seemed like we were doing a good enough job without that check.

In the end, we had over two hundred and fifty thousand votes, while at the time we started the competition we'd only accumulated about fifty thousand.

Pick the Features

We were starting to develop a good reference library of which photos were considered good and bad, but we hadn't decided which attributes we'd try to base the predictions on. The pixel data seemed like the obvious choice, but because the pictures were externally hosted on services like Facebook, that would potentially mean fetching the data for hundreds of thousands of images for every user, and the processing and bandwidth requirements would be far too much for our budget. We were already relying heavily on the text captions for identifying which places and activities were in the photos, so I decided to test a hunch that particular words might be good indicators of quality. I didn't know ahead of time which words might be important, so I set out to create a league table of how highly the most common words correlated with our ratings.

I used some simple Pig scripts to join the votes we'd stored in one table in our Cassandra database with the photos to which they were linked, held in a different table. I then found all the words used in the album title, description, and individual captions, and created overall frequency counts for each. I threw away any words that appeared less than a thousand times, to exclude rare words that could skew the results. I then used Pig to calculate the ratio of good to bad photos associated with each word, and output a CSV file that listed them in order.

When I saw the results, I knew that my hunch was right. For certain words like *Tombs* and *Trails*, over ninety percent of the pictures were rated highly, and at the other end of

the scale, less than three percent of photos with *Mommy* in the caption made people want to travel there. These extremely high and low ranking words weren't numerous enough to offer a simple solution—they still only occurred in a few percent of the captions—but they did give me confidence that the words would be important features for any machine-learning process. One thing I didn't expect was that words like *beautiful* weren't significant indicators of good pictures; it seems like bad photographers are as likely to use that in their captions as good ones!

I also had a feeling that particular locations were going to be associated with clusters of good or bad photos, so I split the world up into one-degree sized boxes, and used another Pig script to place our rated photos into them based on their rounded latitude and longitude coordinates, and ordered the results by the ratio of good to bad photos in each, excluding buckets with less than a thousand photos. Again, there seemed to be some strong correlations with which to work, where places like Peru had over ninety percent of their pictures highly rated. That gave me a good enough reason to put the location coordinates into the inputs we'd be feeding to the machine learning.

Along with word occurrences and position, I also added a few other simple metrics like the number of photos in an album and the photo dimensions. I had no evidence that they were correlated with quality, but they were at hand and seemed likely suspects. At this point I had quite a rich set of features, and I was concerned that providing too many might result in confusion or overfitting. The exact process of picking features and deciding when to stop definitely felt like a bit of an art, but doing some sanity testing to uncover correlations is a good way to have some confidence that you're making reasonable choices.

Encode the Data

Machine-learning algorithms expect to have inputs and outputs that are essentially spreadsheets, tables of numbers with each row representing a single record, with the columns containing particular attributes, and a special column containing the value you're trying to predict. Some of my values were words, so I needed to convert those into numbers somehow. One of the other things I was concerned about was ensuring that the albums had no identifiable information, since essentially anyone could download the datasets from the Kaggle website. To tackle both problems, I randomly sorted the most common words and assigned to them numbers based on the order they appeared in the resulting list. The spreadsheet cells contained a space-separated list of the numbers of the words that appeared in the captions, description, and title.

In a similar way, I rounded down the latitude and longitude coordinates to the nearest degree, to prevent exact locations from being revealed. The other attributes were already integral numbers, so I had everything converted into convenient numerical values. The final step was converting the predicted good or bad values into numbers, in this case just 1 for good and 0 for bad.

Split Into Training, Test, and Solution Sets

Now I had the fifty thousand albums we'd voted on so far described in a single spreadsheet, but for the machine-learning process, I needed to split them up into three different files. The first was the training set, a list of albums' attributes together with the human rating of each one. As the name would suggest, this was used to build a predictive model, fed into the learning algorithm so that correlations between particular attributes and the results could be spotted and used. Next was the test set, which contained another list of different albums' attributes, but without including the results of a human's rating of its photos. This would be used to gauge how well the model that had been built using the training set was working, by feeding in the attributes and outputting predictions for each one.

These first two spreadsheets would be available to all competitors to download, but the third solution set was kept privately on the Kaggle servers. It contained just a single column with the actual human predictions for every album in the test set. To enter the contest, the competitors would upload their own model's predictions for the test set, and behind the scenes Kaggle would automatically compare those against the true solution, and assign each entry a score based on how close all the predicted values were to the human ratings.

One wrinkle was that to assign a score, we had to decide how to measure the overall error between the predictions and the true values. Initially I picked a measure I understood, root-mean square, but I was persuaded by Jeremy Howard to use capped binomial deviance instead. I still don't understand the metric, despite having stared at the code implementing it, but it produced great results in the end, so he obviously knew what he was talking about!

We had to pick how to split the albums we had between the training and the test sets, and a rule of thumb seemed to be to divide them with roughly a three to one ratio. We had just over fifty thousand albums rated at that point, so I put forty thousand in the training, and twelve thousand in the test. I used standard Unix tools like tail to split up the full CSV file, after doing a random sort to make sure that there weren't any selection biases caused by ordering creeping into the data. After that, I was able to upload the three files to Kaggle's servers, and the technical side of the competition was ready.

These same files were also perfect inputs to try with an off-the-shelf machine learning framework. I spent a few hours building an example using the scikit-learn[3] Python package and its default support vector machine package, but it didn't produce very

3. scikit-learn: machine learning in Python (*http://scikit-learn.org/*)

effective results. I obviously needed the expertise of a specialist, so for lack of an in-house genius, I went ahead with the Kaggle competition. It's a good idea to do something similar before you call in outside expertise though, if only to sanity check the data you've prepared and get an understanding of what results a naive approach will achieve.

Describe the Problem

Beyond just providing files, I needed to explain what the competition was for, and why it was worth entering. As a starving startup, we could only afford five thousand dollars for prize money, and we needed the results in just three weeks, so I wanted to motivate contestants as much as I could with the description! As it turned out, people seemed to enjoy the unusually short length of the contest. The description was also useful in explaining what the data meant. In theory, machine learning doesn't need to know anything about what the tables of numbers represent; but in practice, there's an art to choosing which techniques to apply and which patterns to focus on, and knowing what the data is actually about can be helpful for that. In the end, I wrote a few paragraphs discussing the problem we were encountering, and that seemed quite effective. The inclusion of a small picture of a tropical beach didn't seem to hurt, either!

In theory, a machine-learning expert can deal with your problem as a pure numerical exercise. In practice, building good algorithms requires a lot of judgment and intuition, so it's well worth giving them a brief education on the domain in which you're working. Most datasets have a lot of different features, and it helps to know which attributes they represent in the real world so you can tune your algorithms to focus on those with the greatest predictive power.

Respond to Questions

Once we'd launched, I kept an eye on the Kaggle forums and tried to answer questions as they came up. This was actually fairly tough, mostly because other community members or Kaggle staff would often get to them before me! It was very heartening to see how enthused everybody was about the problem. In the end, the three-week contest involved very little work for me, though I know the contestants were extremely busy.

This was one of the areas where our data contest approach led to quite a different experience than you'd find with a more traditionally outsourced project. All of the previous preparation steps had created an extremely well-defined problem for the contestants to tackle, and on our end we couldn't update any data or change the rules part-way through. Working with a consultant or part-time employee is a much more iterative process, because you can revise your requirements and inputs as you go. Because those changes are often costly in terms of time and resources, up-front preparation is still extremely effective.

Several teams apparently tried to use external data sources to help improve their results, without much success. Their hope was that by adding in extra information about things like the geographic location where a photo was taken, they could produce better guesses about its quality. In fact, it appeared that the signals in the training set were better than any that could be gleaned from outside information. We would have been equally happy to use a data source from elsewhere as part of our rating algorithm if it had proven useful, but we were also a little relieved to confirm our own intuition that there weren't preexisting datasets holding this type of information!

Initially, we were quite concerned about the risk of leaking private information. Even though our competition required entrants to agree not to redistribute or otherwise misuse the dataset we were providing, other contests have been won by teams who managed to de-anonymize the datasets. That is, those teams had used the attributes in the test set to connect the entities with external information about them, and in turn to *discover* the right answers directly rather than *predict* them. In our case, we had to make sure that no contestant could find out which albums were being referred to in our test set, lest they look at the photos they contained directly to make a direct human evaluation of their quality. The one-way nature of our encoding process, Facebook's privacy controls, and lack of search capabilities for photos on the site kept the original images safe from the teams.

Integrate the Solutions

We had an amazing number of teams enter—over 400—and the top 10 entries were so close as to be almost identical, but we'd promised to split the prize money among the top 3. That also gave us a nonexclusive license to the code they'd used, and this was the payoff of the whole process for us. My colleague Chris Raynor evaluated the entries, and chose the code from the second place entrant, Jason Tigg. The top three entries were very close, but Tigg's Java code was the easiest to add to our pipeline. We were running the evaluation as a batch process, so we ended up compiling the Java into a command-line app that communicated with Ruby, via sockets. It wasn't the most elegant solution, but it proved robust and effective for our needs.

One unexpected benefit of the machine-learning algorithm was that it produced a real number expressing the probability that an album was good or bad, rather than the original binary up or down vote that the humans produced. That let us move high-probability photos to the front of slideshows, rather than just having two categories.

Conclusion

We're still using the results of the competition very successfully in our product today. I hope this walk-through gave you an idea of how to work effectively with outside machine-learning experts, whether through a contest like Kaggle or through a more traditional arrangement.

CHAPTER 17

Data Traceability

Reid Draper

Your software consistently provides impressive music recommendations by combining cultural and audio data. Customers are happy. However, things aren't always perfect. Sometimes that Beyoncé track is attributed to Beyonce. The artist for the Béla Fleck solo album shows up as Béla Fleck and the Flecktones. Worse, the ボリス biography has the artist name listed as ???. Where did things go wrong? Did one of your customers provide you with data in an incorrect character encoding? Did one of the web-crawlers have a bug? Perhaps the name resolution code was incorrectly combining a solo artist with his band?

How do we solve this problem? We'd like to be able to trace data back to its origin, following each transformation. This is reified as *data provenenace*. In this chapter, we'll explore ways of keeping track of the source of our data, techniques for backing out bad data, and the business value of adopting this ability.

Why?

The ability to trace a datum back to its origin is important for several reasons. It helps us to back-out or reprocess bad data, and conversely, it allows us to reward and boost good data sources and processing techniques. Furthermore, local privacy laws can mandate things like auditability, data transfer restrictions, and more. For example, California's Shine the Light Law requires businesses disclose the personal information that has been shared with third-parties, should a resident request. Europe's Data Protection Directive provides even more stringent regulation to businesses collecting data about residents.

We'll also later see how data traceability can provide further business value by allowing us to provide stronger measurements on the worth of a particular source, realize where to focus our development effort, and even manage blame.

Personal Experience

I previously worked in the data ingestion team at a music data company. We provided artist and song recommendations, artist biographies, news, and detailed audio analysis of digital music. We exposed those data feeds via web services and raw dumps. Behind the scenes, these feeds were composed of many sources of data, which were in turn cleaned, transformed, and put through machine-learning algorithms.

One of the first issues we ran into was learning how to trace a particular result back to its constituent parts. If a given artist recommendation was poor, was it because of our machine-learning algorithm? Did we simply not have enough data for that artist? Was there some obviously wrong data from one of our sources? Being able to debug our product became a business necessity.

We developed several mechanisms for being able to debug our data woes, some of which I'll explore here.

Snapshotting

Many of the data sources were updated frequently. At the same time, the web pages we crawled for news, reviews, biography information, and similarity were updated inconsistently. This meant that even if we were able to trace a particular datum back to its source, that source may have been drastically different than at the time we had previously crawled or processed the data. In turn, we needed to not only capture the source of our data, but the time and an exact copy of the source. Our database columns or keys would then have an extra field for a timestamp.

Keeping track of the time and the original data also allows you to track changes from that source. You get closer to answering the question, "why were my recommendations for The Sea and Cake great last week, but terrible today?"

This process of writing data once and never changing it is called *immutability,* and it plays a key role in data traceability. I'll return to it later, when I walk through an example.

Saving the Source

Our data was stored in several different types of databases, including relational and key-value stores. However, nearly every schema had a *source* field. This field would contain one or more values. For original sources, a single source was listed. As data was processed and transformed into roll-ups or learned-data, we would preserve the list of sources that went into creating that new piece of data. This allowed us to trace the final data product back to its constituent parts.

Weighting Sources

One of the most important reasons we collected data was to learn about new artists, albums, and songs. That said, we didn't always want to create a new entity that would end up in our final data product. Certain data sources were more likely to have errors, misspellings, and other inaccuracies, so we wanted them to be vetted before they would progress through our system.

Furthermore, we wanted to be able to give priority processing to certain sources that either had higher information value or were for a particular customer. For applications like learning about new artists, we'd assign a trust-score to each source that would, among other things, determine whether a new artist was created.

If the artist wasn't created based solely on this source, it would add weight to that artist being created if we ever heard of them again. In this way, the combined strength of several lower-weighted sources could lead to the artist being created in our application.

Backing Out Data

Sometimes we identified data that was simply incorrect or otherwise bad. In such cases, we had to remove the data from our production offering.

Recall that our data would pass through several stages of transformation on its way to the production offering. A backout, then, required that we first identify potential sources of the bad data, remove it, then reprocess the product without that source. (Sometimes the data transformations were so complex that it was easier to generate all permutations of source data, to spot the offender.) This is only possible because we had kept track of the sources that went into the final product.

Because of this observation, we had to make it easy to redo any stage of the data transformation with an altered source list. We designed our data processing pipeline to use parameterized source lists, so that it was easy to exclude a particular source, or explicitly declare the sources that were allowed to affect this particular processing stage.

Separating Phases (and Keeping them Pure)

Often we would divide our data processing into several stages. It's important to identify the state barriers in your application, as doing this allowed us to both write better code, and create more efficient infrastructure.

From a code perspective, keeping each of our stages separate allowed us to reduce side effects (such as I/O). In turn, this made code easier to test, because we didn't have to set up mocks for half of our side-effecting infrastructure.

From an infrastructure perspective, keeping things separate allowed us to make isolated decisions about each stage of the process, ranging from compute power, to parallelism, to memory constraints.

Identifying the Root Cause

Identifying the root cause of data issues is important to being able to fix them and control customer relationships. For instance, if a particular customer is having a data quality issue, it is helpful to know whether the origin of the issue was from data they gave you, or from your processing of the data they gave you. In the former case, there is real business value in being able to show the customer the exact source of the issue, as well as your solution.

Finding Areas for Improvement

Related to blame is the ability to find sources of improvement in your own processing pipeline and infrastructure. This means that the steps in your processing pipeline become data sources in their own right.

It's useful to know, for instance, when and how you derived a certain piece of data. Should an issue arise, you can immediately focus on the place it was created. Conversely, if a particular processing stage tends to produce excellent results, it is helpful to be able to understand why that is so. Ideally, you can then replicate this into other parts of your system.

Organizationally, this type of knowledge also allows you to determine where to focus your teams' effort, and even to reorganize your team structure. For example, you might want to place a new member of the team on one of the infrastructure pieces that is doing well, and should be a model for other pieces, as to give them a good starting place for learning the system. A more senior team member may be more effective on pieces of the infrastructure that are struggling.

Immutability: Borrowing an Idea from Functional Programming

Considering the examples above, a core element of our strategy was *immutability*: even though our processing pipeline transformed our data several times over, we never changed (overwrote) the original data.

This is an idea we borrowed from functional programming. Consider imperative languages like C, Java, and Python, in which data tends to be mutable. For example, if we want to sort a list, we might call `myList.sort()`. This will sort the list in place. Consequently, all references to `myList` will be changed. If we now want review `myList`'s original state, we're out of luck: we should have made a copy before calling `sort()`.

By comparison, functional languages like Haskell, Clojure, and Erlang tend to treat data as immutable. Our list sorting example becomes something closer to `myNewSortedList = sort(myList)`. This retains the unsorted list `myList`. One of the advantages of this immutability is that many functions become simply the result of processing the values passed in. Given a stack trace, we can often reproduce bugs immediately.

With mutable data, there is no guarantee that the value of a particular variable remains the same throughout the execution of the function. Because of this, we can't necessarily rely on a stack trace to reproduce bugs.

Concerning our data processing pipeline, we could save each step of transformation and debug it later. For example, consider this workflow:

```
rawData = downloadFrom(someSite)
cleanData = cleanup(rawData)
newArtistData = extractNewArtists(cleanData)
```

Let's say we've uncovered a problem in the `cleanup()` function. We would only have to correct the code and rerun that stage of the pipeline. We never replaced `rawData` and hence it would be available for any such debugging later.

To take further advantage of immutability, we persisted our data under a compound key of identifier and timestamp. This helped us find the exact inputs to any of our data processing steps, which saved time when we had to debug an issue.

An Example

As an example, let me walk you through creating a news aggregation site. Along the way, I'll apply the lessons I describe above to demonstrate how data traceability affects the various aspects of the application.

Let's say that our plan is to display the top stories of the day, with the ability to drill down by topic. Each story will also have a link to display coverage of the same event from other sources.

We'll need to be able to do several things:

1. Crawl the web for news stories.
2. Determine a story's popularity and timeliness based on social media activity and perhaps its source. (For example, we assume a story on the New York Times home page is important and/or popular.)
3. Cluster stories about the same event together.
4. Determine event popularity. (Maybe this will be aggregate popularity of the individual stories?)

Crawlers

We'll seed our crawlers with a number of known news sites. Every so often, we'll download the contents of the page and store it under a composite key with URL, source, and timestamp, or a relational database row with these attributes. (Let's say we crawl frequently updated pages several times a day, and just once a day for other pages.)

From each of these home pages we crawl, we'll download the individual linked stories. The stories will also be saved with URL, source, and timestamp attributes. Additionally, we'll store the composite ID of the home page where we were linked to this story. That way if, for example, later we suspect we have a bug with the way we assign story popularity based on home page placement, we can review the home page as it was retrieved at a particular point in time. Ideally, we should be able to trace data from our own home page all the way back to the original HTML that our crawler downloaded.

In order to help determine popularity and to further feed our news crawlers, we'll also crawl social media sites. Just like with the news crawlers, we'll want to keep a timestamped record of the HTML and other assets we crawl. Again, this will let us go back later and debug our code. One example of why this would be useful is if we suspect we are incorrectly counting links from shares of a particular article.

Change

Keeping previous versions of the sites we crawl allows for some interesting analytics. Historically, how many articles does the Boston Globe usually link to on their home page? Is there a larger variety of news articles in the summer? Another useful by-product of this is that we can run new analytics on past data. Because immutability can give us a basis from the past, we're not confined only to the data we've collected since we turned on our new analytics.

Clustering

Clustering data is a difficult problem. Outlying or mislabeled data can completely change our clusters. For this reason, it is important to be able to cheaply (in human and compute time) be able to experiment with rerunning our clustering with altered inputs. The inputs we alter may remove data from a particular source, or add a new topic modeling stage between crawling and clustering. In order to achieve this, our infrastructure must be loosely coupled such that we can just as easily provide inputs to our clustering system for testing as we do in production.

Popularity

Calculating story popularity shares many of the same issues as clustering stories. As we experiment, or debug an issue, we want to quickly test our changes and see the result.

We also want to see the most popular story on our own page and trace all the way through our own processing steps, back to the origin site we crawled. If we find out that we've ranked a story as more popular that we would've liked, we can trace it back to our origin crawl to see if, perhaps, we had put too much weight in its position on its source site.

Conclusion

You will need to debug data processing code and infrastructure just like normal code. By taking advantage of techniques like immutability, you can dramatically improve your ability to design and improve your system in a logical manner. Furthermore, we can draw from decades of experience in software design to influence our data processing and infrastructure decisions.

CHAPTER 18
Social Media: Erasable Ink?

Jud Valeski

The commercial use of publicly broadcast social experiences is exploding as businesses work to understand how to engage with our collectively digitized consciousness. This process is forcing every actor in the ecosystem to make decisions around how to behave when they engage with this data. From creation, to consumption, and all of the layers in-between, everyone's expectations are being challenged. This chapter explores these expectations and how the various players are managing them.

The transition from commercially created content (such as news stories and product pages) being the only kind of data available to work with on the network, to personal end user generated content rapidly taking over, is adding completely new behavioral characteristics to the software we're all writing. Many of these characteristics impose significant implementation challenges that "get in the way" of how we're used to working with data. Traditionally, business requirements could be adjusted with negotiation and money, whereas with the common population so intimately involved in today's data production, negotiation and money don't have the same leverage they once did.

What follows is an inspection of our end user expectations and the technical ramifications around what happens to public content that is generated during those moments when we want to "take it back" or "undo" it. What we expect to be happening in those instances is often not. Our public content is resyndicated (propagated from its original source to downstream systems, individuals, and commercial consumers) at phenomenal rates, and it can often be lost or stolen, which breaks a chain of trust along the way. What happens next is left in the balance between end user wants and needs, and technical and policy implementation.

These content "recall" behaviors are inherently challenging to the broader public social data ecosystem. Consider a traditional media outlet broadcasting our expressions on television. What happens if a news outlet puts your social network post on screen for

60 seconds, yet you realize it's being displayed 15 seconds in and delete it. What should happen to the remaining 45 seconds of display? In this case, the data is bad because it poses significant technical challenges when it's leveraged at various points in the ecosystem. From there, it can violate end user expectations.

Social Media: Whose Data Is This Anyway?

As the promise of everything having an IP address comes to fruition (we'll need IPv6 proliferation of course), the amount of data we exhaust into the network has exploded. Much of the information we "share" is implicit, but as social media has taken off, much of that activity has become explicitly conveyed. Entire social media services with multi-billion dollar valuations exist almost exclusively to broadcast end users' public thoughts online for the world to see. The question of ownership is inevitably raised. As an end user who posts on a given service, do I own that content or does the service own it? In attempting to answer that question, subsequent questions around control arise.

To better understand the framework for evaluating the answers to some of these challenging questions, it's useful to consider the two most crucial components in the system: the creator of the content (the end user) and the platform used to make it available. The evolution of these two components is important.

We, as end users, have been able to publicly broadcast our thoughts from an IP address for nearly 17 years now (since the broader network widely came online). It has been technically possible to set up an HTTP server, routable via public DNS, and publish content to it for public consumption for a long time. Over the years, that process has radically shifted, however, and barriers to entry have changed.

1995: Very hard and expensive. Purchase a server (several thousand dollars), an operating system (hundreds of dollars), and connect it to the network via a hosting provider (several thousands of dollars per month). Purchase an HTTP server (hundreds of dollars). Figure out how to write/publish HTML. Purchase a domain name and tie it to DNS (hundreds of dollars). Publicize your "site" (thousands of dollars in marketing campaigns).

2001: Still difficult, but more affordable. Web logging/journaling services started popping up standalone and as an add-on functionality to existing services. For around twenty bucks per month, you could get your thoughts out there for the world to see. You explicitly paid for this "right," however, with an exchange of money for the service.

2006: Private, easy, and free. Closed social networks started coming online (Facebook).

Today: Easy and free (in a monetary sense). On today's network, all you have to do is provide a service with an arbitrary name, an email address, and a password in order to

have the ability to digitally publish your thoughts (from textual expression/posts, to more subtle expressions like re-tweeting or "liking") for the world to see. Social media service providers have absorbed all the infrastructure costs and barriers to entry in order to solve this use case today.

Control

This evolution has resulted in tradeoffs around control and ownership. The burgeoning use of public social media has also put the spotlight on our expectations around control of the content that was created. In 1995, because creation and platforms were managed by the same person, answers to many of the hard questions were easy. However, the ubiquity of all of this public social interaction today raises challenging questions.

Whenever major capital outlays are made in underlying communication infrastructure, control ultimately becomes highly contentious. In the early days, everyone is just having fun using the new service. Railroads, telegraph wires, hardline telephone networks, cable infrastructure, satellite systems, and fiber optic networks have all exhibited control struggles between end users and the firms that spent the billions of dollars to build them. As you would expect, terms of use surrounding these pieces of infrastructure often read something like "you can use my infrastructure for free, or for a fee, but I get to own whatever you do in/on it, and you have to play by my rules. If the government asks me for information regarding your use, I'm going to give it to them." As end users, we generally agree to these terms because the benefits often outweigh the potential downsides.

If the platform/infrastructure is ultimately widely adopted, then the questions around its use get harder. If the entity that originally outlaid the capital starts to impose control, and/or radically change the way the system works or is used, they'll have an end user uprising on their hands. This moment becomes very real, very fast; consider Net Neutrality.

The Net Neutrality effort is a large-scale conflict between the end users (us) of billions of dollars of capital outlay on the part of some pioneering businesses that want to capitalize on their investment. While this is a seemingly reasonable thing to desire, a challenge to the fairness of it arises when end user expectations are developed for years using a set of rules that radically shift due to motivations exclusive to the platform owner.

I believe there's a tipping point for most platforms that theoretically changes a private service over to a public service. Note that I said "theoretically." There's a point wherein we, as end users, certainly start thinking of the service as effectively "public" insofar as we believe the service should be bowing to the broad general desires of a mass user base. Whether or not a private sector entity/service could, or should, be technically moved into the public domain is a can of worms I'm not going to touch on here.

I'll never forget a Voice Over Internet Protocol (VOIP) conference I attended a the Berkley Business School a decade or so ago. The room was filled with people building products, technologies, and services on top of the backbone that a large northern California telco had spent billions of dollars building over several decades. Presentation after presentation of awesome stuff filled the day, and then the single representative from the phone company had his turn on stage. The poor guy had to try and explain to this room of innovative entrepreneurs how his company actually controlled whether or not they could do what they wanted to do. He made no bones about the fact that he believed his investment in this infrastructure gave him the right to control it however he wanted. Regardless of who was right in this scenario, it personified this interesting challenge around who controls what and when.

Back to the question of whose data is this anyway? For the time being, the answer lies in the commercial agreement we as end users agree to when we sign up for, and engage with, a given social media service. I encourage you to read the terms of service for those services.

Commercial Resyndication

As you'd expect, online commerce moves to the interaction ball, wherever the greatest amount of expression/interaction is occurring. Hundreds of millions of humans digitally engaging with each other, and businesses, online, in real-time, is the most significant shift in human behavior since the network widely came online nearly twenty years ago. This has resulted in business and government services (e.g., disaster relief) desiring commercial access to this conversation in order to help humanity and further business/commerce in general. General "developer programs" and lightweight APIs that the publishing services openly provide, due to relative ease of support, do not tackle the challenges inherent in the system. In fact, they can often exacerbate them.

The technical response to this resyndication desire has varied. Some services disallow any resyndication of any kind, while others have leveraged developer APIs and platforms to disseminate the public activity of its end users. Further, some services enter into proprietary resyndication agreements with third-parties in order to ensure resyndication beyond what typical rate-limited, REST-ful, interfaces can provide.

From researchers wanting to understand the human condition, to commercial platforms seeking more effective advertising models on behalf of the companies and brands they represent, the commercial demand for this digitized public conversation is insatiable. A post that an end user makes can be resyndicated thousands of times in the blink of an eye to dozens of other platforms or services. Those platforms and services seek to extract value, sometimes commercial value and sometimes not, from what was said.

Expectations Around Communication and Expression

Whenever new forms of communication arise, it's important to consider previous mechanisms and ways of doing. It gives us a sense of where we've been, and how we should consider, use, and respond to, the new way of doing. Walking through a variety of traditional forms of communication, our expectations around them, the "rights" we've gained (and lost) as they've developed, should help form a basis around how the new medium should behave. I inevitably see gross conflicts in how I want something new to behave when compared to how similar communication means have behaved in the past. Let's consider a handful of ways we communicate.

Verbal Communication

If I vocalize a statement with others around, I can't take back the fact that I said it. I can adjust it and clarify what I meant (even go back on it), but I can't take the words back as if they were never spoken. Humans are accustomed to this dynamic. Politicians understand it down to a scientific level.

Letter Writing

If I hand-write you a letter, I'm in complete control over what I'm communicating all the way down to the moment you open the letter and read it. There are moments when it is out of my control, of course (during postal service delivery), but I'm in complete control while writing it and putting it in the mailbox. I can also be in control on the other end if I want to be physically present when you receive the letter mail. The government has even intervened in this communication stream to impose felony penalties if you interfere with the delivery with letters being sent through the Postal Service.

SMS/Texting

When I text someone, I'm in control of the message up until the point it is "sent" from my phone. Thereafter, however, I have no means of engaging with it.

IM

Similar to SMS, I'm in control until I hit Send.

Email

While there are some features/systems/plugins/extensions in place that allow me to "unsend" or "retract" an email after I've hit "send," they're generally infrequently used, and rarely work in real-world use cases. Generally, email is considered a "fire and forget" communication medium. Once it's sent, that's it!

There are various encryption and certification mechanisms that have also existed for eons around email authenticity and robustness from a trust standpoint; however, unfortunately, mainstream email clients and servers rarely effectively support them in a manner that end users could easily take advantage of.

Social Media Services

Enter social media services such as Facebook and Twitter. An amalgam of clearly private and often not-so-clear public communication comprises many of today's social media services. End users want to express themselves to a variety of types of people using these services; however, a key component to them is that their content is often readily available to other computers and people they didn't necessarily intend to share with at the onset. The underlying Internet framework is based on URLs wherein the name itself implies availability. These services are generally built on top of this model.

The broader URL/network framework has been in place for nearly twenty years, and thousands of products and companies have built up around it. One of the Internet's most successful firms, Google, has tied their existence to the notion of identifying domains, crawling URLs (guessing at what URLs exist out there, as well as walking from link-to-link), scraping the content on the other end of them, storing that content, indexing it, and making it available to individuals and commercial enterprises in a variety of ways.

When you consider how public social data is accessed and leveraged by commerce today, it effectively mirrors how Google has been accessing and leveraging content on the network for over a decade now. However, there is suddenly an intense focus on end user expectations around data handling. What changed?

A new generation largely unaware of how the framework worked underneath started using social media services heavily. MySpace, Facebook, and Twitter garnered massive user bases with hundreds of millions of people using them to communicate. With user bases that large you, by definition, include users that don't fully understand how the system works. The majority of them are simply "users" of a system that others created. They have little or no interest in understanding the underlying framework and the associated benefits and limitations. All they care about is that the higher-level behavioral characteristics they expect around communication on the service are enforced. The social media service must bend to end user desires in order to remain relevant and keep their user base.

The blend of blog-like functionality (specifically the ability to delete a post after you had published it) and high-velocity/real-time hosted social media services like Facebook and Twitter naturally led to end user demand for the ability to delete a previous social network post. Unfortunately, delete functionality being exposed to the end users came along well after these services began making the end user timeline data/posts available through the traditional web frameworks (crawling/scraping), as well as more honed developer APIs. The end result was that there were plenty of data resyndication mechanisms in play among the social media service providers, yet none provided recall or delete capabilities.

The social media services adjusted their developer terms of service and end user terms of service to try and build rules around when and where the underlying data could be used, and how it should be handled should a modification event, such as delete, occur. Some of them go so far as to provide an event notification that a previously created item had been deleted.

From a data standpoint, the cart unfortunately wound up in front of the horse, and this has led to some data handling scenarios that have tremendous infrastructure impact on the overall system. From traditional media outlets to innovative startups, leveraging all of this public social data in a "compliant" manner is a challenge.

Technical Implications of New End User Expectations

One of the services outlined above has the end user expectation of being able to retract a statement that has previously been made. It is notably the most recent and prolific platform of all the social media services. It is also the one that commerce is trying to engage with the most directly.

We're at a point in time where a widely used communication medium has been front-ended by the general web infrastructure. We've always had user generated content (email, paper letters, conversations, SMS), but what is relatively new is that all of this content being created on web-based social media services.

The net result of combining these new communication behavioral expectations (the desire to retract previously made public statements) with the commercial demands is a new layer in the system that must constantly validate and control the dissemination of the content. This is expensive, and can lead to "bad data" proliferating through the network.

Consider the lifespan of a typical public social event (such as a tweet). First it is created and posted using a Twitter client. It is then consumed by Twitter's infrastructure and disseminated to Twitter clients and other third-parties through resyndication means.

At this point, sometime after the message was originally posted, whether one second, or a month afterward, the creator of the Tweet may decide to delete it. If that occurs, the delete request is fielded by Twitter, and that event is disseminated throughout Twitter's infrastructure, including its controlled clients, and through to its resyndication partners (who then, in turn, resyndicate the notification out to their network of customers).

Twitter is often chastised for the control it wants to impose on its product suite. However, if you consider the above flow, without the right levels of control over the user experience,

if for no other reason than to handle deleted Tweets, the end user experience can indeed severely suffer. Twitter has set the expectation with me (the end user), that I can delete my posts. If they can't keep that promise, social contracts can break, and I become an unhappy user and leave the service.

That covers the social media service's expectations and experience, but what about the third parties that consume public posts that the social media service has decided to disseminate downstream?

Well, it's complicated. Consider a general food-stuff product recall. At the point the recall order gets issued, potentially tons of the food-stuff has been distributed, and potentially tons of it has already been consumed. The systems in place to try and recall food-stuff can only be so effective. In a sense, the cat is already out of the bag, and the potential for damage being done prior to knowing whether or not it could have been prevented will always exist.

Now, move this scenario down into the software stack. Just like a blog post going viral on the network, digitized forms of communication are difficult to recall. Often within seconds, a blog post can be crawled, indexed, and cached by search engines such as Google. Social media services coalesce billions of social activities (e.g., posts) everyday, and disseminate them through official and unofficial channels, and end users expect to have control over their activity. It is also worth noting that other social media service users (e.g., your "friends") can share, quote, and resyndicate the content you create, just as rapidly.

In a closed system, this is relatively straightforward to handle. On a simple level, a single database that holds all of the activities and posts made by users using the application on top of the database can be responsible for all the reads and writes for the underlying data. That database is the source of record for everything. If a user deletes some content, it's gone (or suppressed) from the only place it existed to begin with. Nice and clean; good data.

If you start to expand the edge of the system however, the technical challenges get more interesting. Even in our single-db example, for services that gain popularity and need to horizontally scale in order to handle overall access load, it is common to move reads out to caches beyond the database in order to reduce contention. Memcached is a prolific example of this style of scaling. The memcached pattern moves readable data outside of the database and fans it out for more rapid access to the application layer. This means the single source of record is now disconnected from data access requests, and the potential for a disconnect between a data read and a data write exists. In a closed system, this disconnect is relatively understood and underlying database/source of record validation functionality is generally built-in and tunable by the developer.

However, in an open environment, such as the one we're all living in from a network perspective, these cache hit/miss/update policies aren't as well understood or supported. They're certainly far from standardized. The distance between the source of record and a third party consumer of the user-generated content can be vast, and the systems in place to close that gap are new and immature.

If you think about the system as a single source of record database (the social media service responsible for the original inbound content creation posts), with layers upon layers of edge caches (often just other databases), you can see bad data emerge. The update notifications (e.g., deletes) ultimately get in the way of downstream attempts to leverage the data; they're bad data. The orphaned posts and activities that never get modified because the deletes never reach them become bad data as well.

Some systems know how to ingest update notices from the source of record social media services, but many do not. If a system in the web does not know how to handle this scenario, end user expectations can be violated, and the social media service itself can suffer the consequences. Remember that the social media services are ultimately responsible for how end users feel about their functionality.

What Does the Industry Do?

As someone spending their energy building software to support this notion of recall, it sure would be nice if all of us end users simply changed our minds about wanting this functionality. It would make developer's jobs much easier. But, that's not going to happen. The genie is out of the bottle. Instead, we need to be building in the notion of compliant data into our applications. In a sense, social media services acquire third party applications built on top of their developer APIs for precisely this purpose. Doing so allows them to control the complete experience and ensure that the terms of service between themselves and the end user are upheld. Not only are they able to ensure that the branding experience is consistent, but they are also able to ensure that the behavioral expectations are consistent across the suite of products.

That said, there are thousands of third party applications out there that leverage public social media service data constantly. While some social media services indeed require that those third party applications handle the recall use case all the way downstream, many do not, simply because it is not well understood from a policy standpoint, nor from a technical one.

At a minimum, social media services need to make one of two facilities available to ensure data is valid before it is consumed or used by a third party: a validation API or an update notification API.

Validation API

A validation API allows a downstream application or service to validate data before it is put to use. For example, before I display a public post from an end user, I can ask the source of record social media service whether or not it is still there. If it is, I can leverage the data within the confines of the terms of service I signed as a third party service or developer. If it is not, I need to treat the updated data accordingly (either don't display it, or display it according to the update policies the social media service outlines).

Update Notification API

An update notification API disseminates notifications of updates to downstream consuming third party services. This model indicates when an update to originally produced content has occurred. This model has severe technical drawbacks unfortunately. The amount of data resyndicated to third party applications downstream measures in the tens of billions of activities per day. The resyndication service can either maintain a map of who received which activities to ensure only updates to activities a further downstream client received are conveyed, or they can produce a firehose of all update activities for the clients to digest and filter down to what is relevant to them. Either scenario is a large-scale data challenge with nontrivial, ever-growing costs. Keep in mind that end user expectations are that they can delete content at any point after having created it. That could be measured along the spectrum of a second, or year granularity. Maintaining these long-term, growing, databases for potential activity updates is cumbersome, error prone, and expensive.

What Should End Users Do?

End users need to understand the environment in which they're playing. We need to understand how simple it is for our generated content to proliferate across the network. The web was designed for rapid dissemination of information, and today's social media services take complete advantage of that.

We need to ensure that the services we use are providing the support for the behavioral characteristics we demand. Given the amount of resyndication that occurs everyday, there is an overall need to ensure that those expectations and requirements are being enforced. An often overlooked, or ignored, problem is how easily public content is scraped out of a public social media service. It is relatively easy to extract large amounts of content out of a social media service, often without the service even being aware of it occurring. Often when this happens the service's terms of service are being ignored. As end users, we should be demanding that our social media services are taking actions to prevent this sort of behavior. It's akin to your bank not having appropriate levels of security in place to ensure that your money is safe.

Scrapers often go to great lengths to acquire social data. Bot-net-like servers are spun-up in a dynamic manner across vast IP address blocks on the network, and the social media services are often unable to differentiate web-browser requests for information from these automated requests. If they can identify the often unwanted scraping activity, they have to go to great lengths to mitigate it. By the time this is done, any logical link between some content you may have publicly published, and your ongoing expected ability to recall that content is broken.

An often leveraged model for minimizing this kind of scraping is to provide official channels for the kind of data the scrapers are actually after. It's easier for a social media service to open up complete coverage access (such as firehoses) that third parties can leverage officially, rather than for them to continue any cat-and-mouse activity they may be playing with parties scraping data from them. It's easier to provide voluminous data access through non-multipurpose web interfaces. Bringing resyndication into the light where everyone can see what's happening provides transparency to the system. Tolerance of this kind of resyndication on the part of end users is something we should seriously consider.

Overall, we need to be privy to the underlying foundation of communicating on today's social media services. Unless you're participating in a completely proprietary closed network, your online activities can rapidly be distributed across many other services (some good, some bad). We need to be fully aware of these dynamics, and only communicate things online that we are comfortable proliferating. Until recall dynamics are fully realized and implemented (from expectations to technical implementation), awareness and caution are in order around what we're sharing, and when.

How Do We Work Together?

Equal exchange of value is ultimately the answer. That exchange can be explicit in the form of money for service, or it can take a more circuitous route. A friend of mine refused to use the "express toll" lane on her commute for years because she didn't want the service to be tracking her whereabouts. Eventually, the value of a faster commute usurped her desire for privacy in that scenario, and she signed up for the service to speed her commute. The exchange ultimately became worth it to her.

As end users, we've evolved to expect that the services we use be free. In light of that, those services are generally compelled to derive financing through other means. We need to ensure that our behavioral expectations are met, while accepting that it is a system of give-and-take.

The public social data ecosystem will need to adapt to support the "recall" use case that we, as end users, clearly demand. This "bad data" will ultimately become good, but it is going to take some work to get it there. We're at an amazing moment in time when public

social data resyndication can lead to phenomenal new innovations as our digitized social consciousness is engaged. However, if we don't get the "recall" use case right, the data will be forever confined to the source-of-record social media service wherein it was originally posted.

CHAPTER 19

Data Quality Analysis Demystified: Knowing When Your Data Is Good Enough

Ken Gleason and Q. Ethan McCallum

Most of the decisions we make in our personal and professional lives begin with a query. That query might be for a presentation, a research project, a business forecast or simply finding the optimal combination of shipping time and price on tube socks. There are times when we are intuitively comfortable with our data source, and/or are not overly concerned about the breadth or depth of the answers we get, for instance, when we are looking at movie reviews. Other times, you might care a little more, for example, if you are estimating your requirements for food and water for the Badwater Ultramarathon. [1] Or even for mundane things like figuring out how much of a product to make, or where your production bottlenecks are on the assembly line.

But how do we know when to care and when not to care, and about what? Should you throw away the survey data because a couple of people failed to answer certain questions? Should you blindly accept that your daily sales of widgets in Des Moines seem to quintuple on alternate Fridays? Maybe, maybe not. Much of what you (think you) know about the quality of a given set of data relies on past experience that evolves to intuition. But there are three problems with relying solely on intuition. First, intuition is good at trapping obvious outliers (errors that stick out visibly) but likely won't do much to track more subtle issues. Second, intuition can be wrong. How do you validate what your gut tells you looks funny (or doesn't)? Third, as discussed above, intuition relies on evolved experience. What happens when you lack direct domain experience? This is hardly an academic question; in the data business, we are often thrown into situations with brand

1. Badwater is a 135-mile foot race that goes through California's Death Valley. And before you ask: no, neither of the authors has come remotely close to attempting it. We just think it's cool.

new problems, new datasets, and new data sources. A systematic approach to data quality analysis can guide you efficiently and consistently to a higher degree of awareness of the characteristics and quality of your data before you spend excessive time making personal or business decisions.

Operating from a data quality framework allows you to:

- Quit worrying about what you think you know or don't know about the data.
- Step outside conventional wisdom about what you need and don't need, and establish fresh conceptions about your data and its issues.
- Develop and re-use tools for data quality management across a wider variety of scenarios and applications.

This chapter outlines a conceptual framework and approach for data quality analysis that will hopefully serve as a guide for how you think about your data, given the nature of your objective. The ideas presented here are born from (often painful) experience and are likely not new to anyone who has spent any extended time looking at data; but we hope it will also be useful for those newer to the data analysis space, and anyone who is looking to create or reinforce good data habits.

Framework Introduction: The Four Cs of Data Quality Analysis

Just as there are many angles from which to view your data when searching for an answer, there are many viewing angles for assessing quality. Below we outline a conceptual framework that consists of four facets. We affectionately refer to them as The Four Cs of Data Quality Analysis[2]:

1. **Complete:** Is everything here that's supposed to be here?
2. **Coherent:** Does all of the data "add up?"
3. **Correct:** Are these, in fact, the right values?
4. **aCcountable:** Can we trace the data?

Granted, these categories are fairly broad and they overlap in places. Sometimes a C or two won't even apply to your situation, depending on your requirements and your place in the data processing chain. Someone interested in *gathering* and storing the data may

2. One may argue that this is more like 3.5 Cs…. Fair enough.

have a different view of "complete" than someone who is trying to use the data to build an *analysis* and drive a decision. That's fine. We don't intend the Four Cs to be a universal, airtight methodology. Instead, consider them a starting point for discussion and a structure around which to build your own data quality policies and procedures.

Complete

The notion of a *complete* dataset is paradoxically difficult to nail down. It's too easy to say, "it's a dataset that has everything," and then move on. But how would we define "everything" here? Is it simply the number of records? Expected field count? Perhaps we should include whether all fields have values—assuming that "lack of value" is not, in and of itself, a value. (This doesn't yet cover whether the values are all valid; that's a matter of Correctness, which we describe below.) This leads us down the path of learning which fields are necessary, versus nice to have.

Most of these questions stem from the same root, and this is the very nature of completeness in data:

Do I have all of the data required to answer my question(s)?

Even this can be tricky: you often first have to use the data to try to answer your question, and then verify any results, before you know whether the data was sufficient. You may very well perform several iterations of this ask-then-verify dance. The point is that you should approach your initial rounds of analysis as checks for completeness, and treat any findings as preliminary at best. This spares you the drama of actions taken on premature conclusions, from what is later determined to be incomplete data.

Let's start with the simplest, but frequently overlooked sort of completeness: do you actually have *all* of the records for a given query? This is a basic yet essential question to ask if you are running any kind of analysis on data that consists of a finite and well-defined number of records, and it's important that the totals (say, number of orders or sum of commission dollars or total students registered) match some external system. Imagine presenting a detailed performance analysis on the stock trading activity for your biggest client, only to find that you've missed fully half their activity by forgetting to load some data. Completeness can be that simple, and that important. Without getting into aCcountability just yet, it's simply not enough to *assume* that the data you received is all you need.

Thinking more about stock trading and modern electronic trading (where computer programs make the bulk of decisions about how orders get executed) offers a wealth of examples to consider. Take a simple one: many modern methods of evaluating whether a given trade was done "well" or not involve measuring the average price achieved versus some "benchmark," such as the price at which the stock was trading at the time the order arrived (the idea being that a well traded "buy" ("sell") order should not move the stock up (down) excessively). While it's not necessary to delve into the gory details, it is

sufficient for our purposes to observe that most modern benchmarks require one or more inputs, including the time that an order started. If this sort of performance is a requirement, the order data information would not be considered complete unless the start time for each order was present.

Another example from the world of electronic trading: consider the relationships between orders in a stock trading system. A trader may place an order to buy 100,000 shares of a given stock, but the underlying system may find it optimal to split that one logical order into several smaller "child orders" (say, 100 child orders of 1000 shares each). The trading firm's data warehouse could collect the individual child order times and amounts but omit other information, such as where the order actually traded or identifiers that tie the child orders to the original order. Such a dataset could indeed be considered complete if the question were, "what was our total child order volume (count) last month?" On the other hand, it is woefully incomplete to answer the question, "what is our breakdown of orders, based on where they were traded?" Unless the firm has recorded this information elsewhere, this question simply cannot be answered. The remedy would be to extend and amend the data warehouse to collect these additional data points, in anticipation of future such queries. There is clear overlap here with the process of initial database design, but it bears review at query analysis time, given how expensive replaying / backfilling the data could be.

Evaluating your dataset for completeness is straightforward:

Understand the question you wish to answer. This will determine which fields are necessary, and what percentage of complete records you'll need. Granted, this is not always easy: one aspect of data analysis is asking questions that haven't been asked before. Still, you can borrow a page from game theory's playbook: *look ahead and reason back*.[3] Knowing your business, you can *look ahead* to the general range of questions you'd expect to answer, and then *reason back* to figure out what sort of data you'd need to collect, and start collecting it. It also helps to note what data you access or create but don't currently collect, and start collecting that as well. A common refrain in technology is that *disk is cheap*[4] and that data is the new gold. Passing up data collection to avoid buying storage, then, is like passing up money to avoid finding a place to put it.

Confirm that you have all of the records needed to answer your question. The mechanics here are straightforward, but sometimes onerous. Straightforward, in that you can check for appropriate record count and presence of fields' values. Onerous, in that you typically have to write your own tools to scan the data, and often you'll go through several

3. Dixit and Nalebuff's *Thinking Strategically: The Competitive Edge in Business, Politics, and Everyday Life* is a text on game theory. It covers the "look ahead and reason back" concept in greater detail.

4. At least, it's common to say that these days. We began our careers when disk (and CPU power, and memory) was still relatively expensive.

iterations of developing those tools before they're truly road-worthy.[5] That we live in an age of Big Data™ adds another dimension of hassle: checking a terabyte dataset for completeness is our modern day needle-and-haystack problem. Sometimes, it is simply not possible. In these cases, we have to settle for some statistical sampling. (A thorough discussion of sampling methodology is well beyond the scope of this book.)

Take action. What we've talked about so far is almost entirely evaluation; what action(s) should you take when you have assessed completeness? In the case of record-level completeness, it's easy; you either have all the records or you don't, and you then find them and backfill your dataset. But what should you do if you find 10% (or whatever your meaningful threshold is) of your records having missing values that prevent some queries from being answered? The choices generally fall into one of the following categories:

- Fix the missing data. Great, if you can, though it's not always possible or practical.
- Delete the offending records. This is a good choice if your set of queries need to be internally consistent with each other.
- Flag the offending records and note their incompleteness with query results.
- Do nothing. Depending on your queries, you may just be able to happily crack on with your analysis, though, in our experience, this tends to be a bad idea more often than not…

These are only four options. There are certainly more, but making (and documenting) the evaluation and subsequent actions can save considerable pain later on.

Coherent

Assuming your data is now complete, or at least as complete as you need it to be, what's next? Are you ready to uncover the gold mine hidden in your data? Not so fast—we have a couple of things left to think about. After completeness, the next dimension is *Coherence.* Simply put: *does your data make sense relative to itself?*

In greater detail, you want to determine whether records relate to each other in ways that are consistent, and follow the internal logic of the dataset. This is a concept that may, at first glance, feel more than a little redundant in the context of data analysis; after all, relational databases are designed to enforce coherence through devices like referential integrity, right? Well, yes, and no.

5. Keep in mind that any time you write your own tools, you have to confirm that any problems they detect are in fact data problems and not code bugs. That is one advantage to working with the same dataset over a long period of time: the effort you invest in creating these tools will have a greater payoff and a lot more testing than their short-lived cousins.

We can't always trust (or even expect) referential integrity. Such checks can cause a noticeable performance hit when bulk-loading data, so it's common to disable them during raw data loads. The tradeoff here is the risk of orphaned or even duplicate records, which cause a particular brand of headache when referential integrity is reenabled. Also, consider data that is too "dirty" for automatic integrity checks, yet is still valid for certain applications. Last, but not least, your data may be stored in a document database or other NoSQL-style form. In this case, referential integrity at the database level is quite intentionally off the menu. (A discussion about whether or not the data is structured in an appropriate fashion may certainly be warranted in this case, but is well beyond the scope of this book.)

Referential integrity is only one sort of coherence. Another form is *value integrity:* are the values themselves internally consistent where they need to be? Let's revisit our stock trading order and execution database for an example of value integrity. Consider a typical structure, in which we have two tables:

Table 19-1. Columns in sample database table 1: orders

Value
order_id
side
size
price
original_quantity
quantity_filled
start_time
end_time

Table 19-2. Columns in sample database table 2: fills

Value
original_order_id
fill_quantity_shares
fill_price
fill_time

An *order* is an order to buy or sell a stock. A *fill* is a record of one particular execution on that stock. For example, I may place an order to buy 1,000 shares of AAPL at a certain limit price, say $350.[6] In the simplest case, my order is completely executed ("filled") and thus I would expect to have a corresponding record in the `fills` table with

6. We plan to do this with our publication royalties from this chapter.

`fill_quantity_shares = 1000`. What if, however, I buy my 1,000 shares in more than one piece, or fill? I could conceivably get one fill of 400 shares, and another of 600, resulting in *two* records in the fills table instead of one. So the basic relationship is one order to many fills. How, then, does value integrity fit here? In our model of order and fill data, the sum of fill quantities should never be greater than the original quantity of the order. Compare:

```
select sum(fill_quantity_shares) where fills.order_id = x
```

to:

```
select original_quantity from orders where orders.order_id = x
```

Again, it's attractive to trust the data and assume that this sort of value level integrity is maintained at record insert time, but in the real world of fragmented systems, multiple data sources and (gasp!) buggy code, an ounce of value analysis is often worth a pound (or ton!) of cure.

Another, more subtle example of value integrity concerns timestamps: Similar to the sum of share quantities above, by definition, all fills on a given order should occur in time no earlier than the order's start time, and no later than the order's end time. [7]

In the case of our two-table dataset:

```
if fills.original_order_id = orders.order_id,
```

then it should be true that:

```
orders.start_time <= fills.fill_time <= orders.end_time
```

As with Completeness, evaluation of Coherence can take many forms.

Determine what level and form of coherence you really need. Is it validation on referential integrity? Simple value integrity validation on one or more field values? Or does it require a more complex integrity validation?

Determine how complete your validation needs to be, and what your performance and time constraints are. Do you need to validate every single record or relationship? Or can you apply statistical sampling to pick a meaningful but manageable subset of records to evaluate?

As always, it's critical that your evaluation fits your needs and properly balances the time/quality tradeoff.

7. However, if you do happen to figure out a way to get your fills to actually happen before you send your order, please tell us how you did it!

Once you have validated your data, you have to decide what to do with the problems you've found. The decisions to fix, omit, or flag the offending records are similar to those for Completeness and will depend as always on your requirements, though the balance may be different. Fixing errors involving referential or value integrity tend to be a mixture of finding orphaned records, deleting duplicates, and so forth.

Correct

Having confirmed that your data is both complete and coherent, you're still not quite ready to crunch numbers. You now have to ask yourself whether your data is *correct* enough for what you're trying to do. It may seem strange to consider this a precursor to analysis, as analysis often serves to somehow validate the dataset; but keep in mind, there may also be "sub-dimensions" of correctness that bear validation before you move on to the main event. Similar to testing for coherence, correctness requires some degree of domain knowledge.

One thing to remember is that correctness itself can be relative. Imagine you've gathered data from a distributed system, composed of hundreds of servers, and you wish to measure latency between the component services as messages flow through the system. Can you just assume that clocks on all the machines are synchronized, or that the timestamps on your log records are in sync? Maybe. But even if you configure this system yourself, things change (and break).

One simple check would be to confirm that the timestamps are moving in the proper direction. Say that messages flow through systems `s1`, `s2`, and `s3`, in that order. You could check that the timestamps are related as follows:

```
message_timestamp(s1) <= message_timestamp(s2) <= message_timestamp(s3)
```

You may determine that timestamps of messages passing through `s2` are consistently less than (newer than) the timestamps when those same messages passed through `s1`. Once you rule out time travel, you reason that you have a systematic error in the data (brought about by, say, a misconfigured clock on either `s1` or `s2`). You can choose between correcting the systems' clocks and rebuilding the dataset, or adjusting the stored timestamps, or some other corrective action. It may be tempting, then, to include such logic inline in your queries, but that would add additional complexity and runtime overhead. A more robust approach would be to uncover this problem ahead of time, before making the data available for general analysis. This keeps the query code simple, and also ensures that any corrections apply to all queries against the dataset.

As a second example, imagine you're in charge of analytics for a system that tracks statistics—times, routes, distances, and so on—for thousands of runners. Runners who train or compete at different distances will typically run longer distances at a slower

pace. A person whose best one-mile time is six minutes will, more than likely, run a marathon (26.2 miles) at a speed *slower* than six minutes per mile. Similarly, older age groups of runners will tend to have slower times (except in some cases for longer distances).

Simple, straightforward validation checks can uncover possible issues of internal correctness (distinct from coherence). How you handle them depends, as always, on your objective and domain knowledge. If you're simply gathering site statistics, it might not matter. On the other hand, it could also be interesting to learn that certain members or groups consistently under-report race times. In this way, the border between correctness and simply "interesting" data becomes a gray area, but useful to think about nevertheless.

The general process for evaluation of Correctness should start to look familiar by now:

Itemize the elements of your data that can be easily verified out of band. Given a data dump from a point-of-sale system, are all transactions expected to have occurred in the years 2010 and 2011? Perhaps they are all from your friend's dollar store, where a single line-item sale of $100 could reasonably be considered erroneous?

Determine which of these are important to validate, in the context of what you care about. As data-minded people, it's common to want to categorize every dataset and grade it on every dimension. To do so would be highly inefficient, because we are limited by time constraints, and requirements tend to change over time. We suggest you review every dataset and attach a log of what was checked at the time.

Understand how much of your data must be correct. As with Coherence, can (must?) you check *every* record, or is some sort of sampling sufficient?

Decide what you will do with incorrect data. Is it possible to "fix" the records somehow, or must you work without them? Depending on the questions you're trying to answer, you may be able to weight known-incorrect records and reflect that in the analysis. Another option would be to segregate these records and analyze them separately. It could make for an interesting find if, say, your model were known to perform equally well on correct and incorrect data. (That could indicate that the field in question has no predictive power, and may be safely removed from your feature set.)

aCcountable

Who is responsible for your data? This may seem an odd question when discussing data quality, but it does indeed matter. To explain, let's first consider how data moves.

Data flow typically follows a pattern of: acquire, modify, use. *Acquire* means to get the data from some source. To *modify* the data is to clean it up, enrich it, or otherwise tweak

it for some particular purpose. You can *use* the data to guide internal decisions, and also distribute it externally to clients or collaborators. The pattern is repeated as data passes from source to recipient, who in turn becomes the source for the next recipient, and so on.

This is quite similar to supply chain management for tangible goods: one can trace the flow of raw material through perhaps several intermediate firms. Some goods, such as food and drink, have a final destination in that they are consumed. Others, through recycling and repurposing, may continue through the chain for perpetuity.

One critical element of the supply chain concept is *accountability:* one should be able to trace a good's origins and any intermediate states along the way. An outbreak of food-borne illness can be traced back to the source farm and even the particular group of diseased livestock; an automobile malfunction can be traced back to the assembly line of a particular component; and so on.

When assessing your data for quality, can you make similar claims of traceability? That leads us to answer our original question, then. Who's responsible for your data? *You are responsible,* as are your sources, and their sources, and so on.

Data's ultimate purpose is to drive decisions, hence its ultimate risk is being incorrect. Wise leaders therefore confirm any numbers they see in their reports. (The more costly the impact of the decision, the more thoroughly they check the supporting evidence.) They check with their analysts, who check their work and then check with the data warehouse team, who in turn check their work and *their* sources, and so on. This continues until they either reach the ultimate source, or they at least reach the party ultimately responsible.[8]

Equally pressing are external audits, driven by potential suitors in search of a sale, or even government bodies in the aftermath of a scandal. These have the unpleasant impact of adding time pressure to confirm your sources, and they risk publicly exposing how your firm or institution is (quite unintentionally, of course) headed in the wrong direction because it has been misusing or acting on bad information.

Say, for example, that you've been caught red-handed: someone's learned that your application has been surreptitiously grabbing end users' personal data. You're legally covered due to a grey area in your privacy policy, but to quell the media scandal you declare that you'll immediately stop this practice and delete the data you've already collected.

That's a good start, but does it go far enough? Chances are, the data will have moved throughout your organization and made its way into reports, mixed with other data, or

8. Inside a company, this can mean "someone we can fire." For external data vendors, this is perhaps "someone we can sue for having provided bad data." Keep in mind, the party ultimately responsible could be *you*, if the next person in the chain has managed to absolve themselves of all responsibility (such as by supplying an appropriate disclaimer).

even left your shop as it's sold off to someone else. Can you *honestly* tell your end users that you've deleted *all* traces of their data? Here's a tip: when the angry hordes appear at your doorstep, pitchforks and torches in hand, that is not the time to say, "um, maybe?" You need an unqualified "yes" and you want to back it with evidence of your data traceability processes.

It's unlikely that a violent mob will visit your office in response to a data-privacy scandal.[9] But there are still real-world concerns of data traceability. Ask anyone who does business with California residents. Such firms are subject to the state's "Shine the Light" legislation, which requires firms to keep track of where customers' personal data goes when shared for marketing purposes.[10] In other words, if a customer asks you, "with whom have you shared my personal information?", you have to provide an answer.

As a final example, consider the highly-regulated financial industry. While the rules vary by industry and region, many firms are subject to record retention laws which require them to maintain detailed audit trails of trading activity. In some cases, the firms must keep records down to the detail of the individual electronic order messages that pass between client and broker. The recording process is simple: capture and store the data as it arrives. The storage, however, is the tough part: you have to make sure those records reflect reality, and that they don't disappear. When the financial authorities make a surprise visit, you certainly don't want to try to explain why the data wasn't backed up, and how a chance encounter between the disk array and a technician's coffee destroyed the original copy. Possibly the most iconic example of the requirement for traceability and accountability is the Sarbanes-Oxley Act[11] in the United States that requires certification of various financial statements with criminal consequences.[12]

Humor aside, surprise audits do indeed occur in this industry and it is best to be prepared. To establish a documented chain of accountability should yield, as a side-effect, a structure that makes it easy to locate and access the data as needed. (The people responsible will, inevitably, create such a structure such that they do not become human bottlenecks for any sort of data request.) In turn, this means being able to quickly provide the auditors the information they need, so you can get back to work.[13]

9. So to all those who make their money swiping personal data, you may breathe a sigh of relief … for now.
10. California Civil Code section 1798.83: *http://www.leginfo.ca.gov/cgi-bin/displaycode?section=civ&group=01001-02000&file=1798.80-1798.84*. The Electronic Privacy Information Center provides a breakdown at *http://epic.org/privacy/profiling/sb27.html*
11. *http://en.wikipedia.org/wiki/Sarbanes%E2%80%93Oxley_Act*
12. Think of all those times when you felt that someone's incompetence was criminal…
13. …or even, to go home. Some audits have been known to occur at quitting time, and you don't want to be part of the crew stuck in the office till the wee hours of the morning hunting down and verifying data.

Once again, the steps required to achieve proper data accountability depend on your situation. Consider the following ideas to evaluate (or, if need be, define) your supporting infrastructure and policies:

Keep records of your data sources. Note how you access the data: push or pull, web service call or FTP drop, and so on. Determine which people (for small companies), teams (larger companies), and companies (for external sources) own or manage each dataset. Determine whom to contact when you have questions, and understand their responsibilities or service-level agreements (SLAs) for the data: what are the guarantees (or lack thereof) concerning the quality, accuracy, and format of the data, and the timeliness of its arrival?

Store everything. Hold on to the original source data *in addition to* any modifications you make thereof. This lets you check your data conversion tools, to confirm they still operate as expected. As an added bonus, it dovetails well with the next point.

Audit yourself. Occasionally spot-check your data, to confirm that your local copies match the source's version. Confirm that any modification or enrichment processes do not introduce any new errors. When possible, check your records *against themselves* to confirm the data has not been modified, either by subtle storage failure (bit rot), by accident, or by conscious choice (tampering).[14]

As an example, consider the Duracloud Bit Integrity Checker. Duracloud is a cloud-based storage service, and the Bit Integrity Checker lets admins upload MD5 checksums along with the data. In turn, the service periodically confirms that the stored data matches the supplied checksums.[15]

Most importantly, if local regulations subject you to audits, make sure you know the format and content the auditors require. Provide sample records to the auditors, when possible, to make sure you collect (and can access) all of the details they expect.

Watch the flow. Keep track of how the data flows through your organization, from source to modification, from enrichment to report. Follow the supply-chain concept to track any research results to their origin. If data is updated, any derivative data should follow suit.

Track access. Understand who can access the data, and how. In an ideal world, any read-write access to the data would occur through approved applications that could track usage, perform record-level auditing, and employ an entitlements system to limit what each end-user sees. (Such a scenario has the added benefit that the applications can, in turn, themselves be audited.)

14. Whether it's shady companies removing evidence of fraud, or a rogue sect hiding a planet where they're developing a clone army, people who remove archive data are typically up to no good.

15. *https://wiki.duraspace.org/display/DURACLOUD/Bit+Integrity+Checker*

If you have the unfortunate scenario that certain "power users" have direct read-write access to the raw data (say, through desktop SQL tools), at least provide everyone their own login credentials.[16]

At the end of the day, whether the audit is yours or the CEO's, for informal or criminal proceedings, for tube socks or Tube schedules, if you are unable to describe, audit, and maintain the aCcountability chain of your data, the maintenance of proper Completeness, Coherence, and Correctness not only becomes that much more difficult, it becomes largely irrelevant.

Conclusion

The Four Cs framework presented here is only one of many possible ways of looking at data quality, and as we've said, your mileage may vary (as will the relevance of any given *C*). That said, there are three principles inherent to this discussion that bear repeating and that should be relevant to any data quality analysis framework:

Think about quality separately (and first, and iteratively) from the main task at hand. Getting in the habit of thinking about data quality before the real work begins not only saves time, but gives you a better understanding of your capabilities and limitations with respect to analyzing that set.

Separate the actual execution of data quality checks from the main task at hand whenever it is practical to do so. Validation logic tends to be separated from main flow in most programming paradigms; why not in data handling?

Make conscious (and documented) decisions about the disposition of data that doesn't live up to your (also documented!) Four Cs checks. If you aren't making and documenting clear decisions and actions that result from your determination of data quality, you might as well skip the entire process.

We hope you found this chapter reasonably complete, coherent, and correct. If not, the authors are the only ones accountable.

Data quality analysis, and any related data munging, is a necessary first step in getting any meaningful insight out of our data. This is a dirty and thankless job, and is often more time-consuming than the acutal analysis. You may at least take comfort in the words of Witten, et. al.:

> Time looking at your data is always well-spent.[17]

16. … and then, lobby to build some of the aforementioned applications. It's quite rare that end users *need* that kind of raw data access, and yet all too common that they have it.

17. Besides providing a witty quote with which to close the chapter, Witten, et. al.'s *Data Mining: Practical Machine Learning Tools and Techniques* (third edition) includes some useful information on data cleaning and getting to know your data.

Indeed.

Index

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.

D

S

T

U

V

W

X

Y

About the Author

Q. Ethan McCallum is a consultant, writer, and technology enthusiast, though perhaps not in that order. His work has appeared online on The O'Reilly Network and Java.net, and also in print publications such as *C/C++ Users Journal*, *Doctor Dobb's Journal*, and *Linux Magazine*. In his professional roles, he helps companies to make smart decisions about data and technology.

Colophon

The animal on the cover of *Bad Data Handbook* is Ross's goose (*Chen rossii or Anser rossii*), a North American species that gets its name from Bernard R. Ross, a Hudson's Bay Company factor at Fort Resolution in Canada's Northwest Territories. Other names coined for this species are "galoot" and "scabby-nosed wavey," by Northmen. There is debate about whether these geese belong in the "white" geese genus of *Chen* or the traditional "gray" goose genus of *Anser*. Their plumage is primarily white with black wing tips, reminiscent of the white-phase Snow Goose, but about 40% smaller in size.

No matter their technical genus, these birds breed in northern Canada and the central Arctic (primarily in the Queen Maud Gulf Migratory Bird Sanctuary), wintering far south in the southern United States, central California, and sometimes northern Mexico. In Western Europe, these birds are kept mostly in wildfowl collections, but escaped or feral birds are often encountered with other feral geese (Cananda Goose, Greylag Goose, Barnacle Goose).

The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSansMonoCondensed.