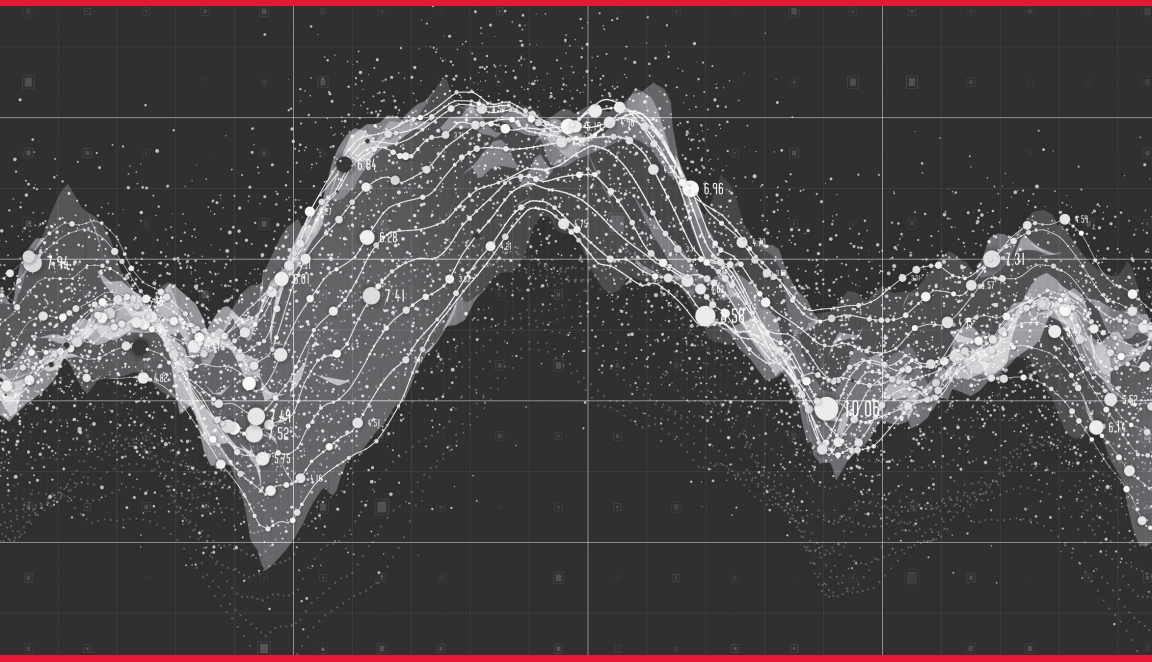
O'REILLY®

# Machine Learning Logistics

## Model Management in the Real World



Ted Dunning & Ellen Friedman






# Strata
## DATA CONFERENCE

## Make Data Work
**strataconf.com**

Data is driving business transformation. Presented by O'Reilly and Cloudera, Strata puts cutting-edge data science and new business fundamentals to work.

- Learn new business applications of data technologies
- Get the latest skills through trainings and in-depth tutorials
- Connect with an international community of data scientists, engineers, analysts, and business managers

Job # D4211

# Machine Learning Logistics

## *Model Management in the Real World*

*Ted Dunning and Ellen Friedman*

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

**Machine Learning Logistics**
by Ted Dunning and Ellen Friedman



Printed in the United States of America.

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



**Editor:** Shannon Cutt
**Production Editor:** Kristen Brown
**Copyeditor:** Octal Publishing, Inc.
**Interior Designer:** David Futato
**Cover Designer:** Karen Montgomery
**Illustrator:** Ted Dunning and Ellen Friedman

September 2017: First Edition

**Revision History for the First Edition**
2017-08-23: First Release
2017-10-30: Second Release



978-1-491-99761-1

[LSI]

# Table of Contents

# Preface

Machine learning offers a rich potential for expanding the way we work with data and the value we can mine from it. To do this well in serious production settings, it's essential to be able to manage the overall flow of data and work, not only in a single project, but also across organizations.

This book is for anyone who wants to know more about getting machine learning model management right in the real world, including data scientists, architects, developers, operations teams, and project managers. Topics we discuss and the solutions we propose should be helpful for readers who are highly experienced with machine learning or deep learning as well as for novices. You don't need a background in statistics or mathematics to take advantage of most of the content, with the exception of evaluation and metrics analysis.

## How This Book is Organized

Chapters 1 and 2 provide a fundamental view of why model management matters, what is involved in the logistics and what issues should be considered in designing and implementing an effective project.

Chapters 3 through 7 provide a solution for the challenges of data and model management. We describe in detail a preferred architecture, the *rendezvous architecture*, that addresses the needs for working with multiple models, for evaluating and comparing models effectively, and for being able to deploy to production with a seamless hand-off into a predictable environment.

Chapter 8 draws final lessons. In Appendix A, we offer a list of additional resources.

Finally, we hope that you come away with a better appreciation of the challenges of real-world machine learning and discover options that help you deal with managing data and models.

## Acknowledgments

We offer a special thank you to data engineer Ian Downard and data scientist Joe Blue, both from MapR, for their valuable input and feedback, and our thanks to our editor, Shannon Cutt (O'Reilly) for all of her help.

# CHAPTER 1
# Why Model Management?

*90% of the effort in successful machine learning is not about the algorithm or the model or the learning. It's about logistics.*

Why is model management an issue for machine learning, and what do you need to know in order to do it successfully?

In this book, we explore the logistics of machine learning, lumping various aspects of successful logistics under the topic "model management." This process must deal with data flow and handle multiple models as well as collect and analyze metrics throughout the life cycle of models. Model management is not the exciting part of machine learning—the cool new algorithms and machine learning tools—but it is the part that unless it is done well is most likely to cause you to fail. Model management is an essential, ubiquitous and critical need across all types of machine learning and deep learning projects. We describe what's involved, what can make a difference to your success, and propose a design—the *rendezvous architecture*—that makes it much easier for you to handle logistics for a whole range of machine learning use cases.

The increasing need to deal with machine learning logistics is a natural outgrowth of the big data movement, especially as machine learning provides a powerful way to meet the huge and, until recently, largely unmet demand for ways to extract value from data at scale. Machine learning is becoming a mainstream activity for a large and growing number of businesses and research organizations. Because of the growth rate in the field, in five years' time, the majority of people doing machine learning will likely have less than five years of experience. The many newcomers to the field need practical, real-world advice.

# The Best Tool for Machine Learning

One of the first questions that often arises with newcomers is, "What's the best tool for machine learning?" It makes sense to ask, but we recently found that the answer is somewhat surprising. Organizations that successfully put machine learning to work generally don't limit themselves to just one "best" tool. Among a sample group of large customers that we asked, 5 was the smallest number of machine learning packages in their toolbox, and some had as many as 12.

Why use so many machine learning tools? Many organizations have more than one machine learning project in play at any given time. Different projects have different goals, settings, types of data, or are expected to work at different scale or with a wide range of Service-Level Agreements (SLAs). The tool that is optimal in one situation might not be the best in another, even similar, project. You can't always predict which technology will give you the best results in a new situation. Plus, the world changes over time: even if a model is successful in production today, you must continue to evaluate it against new options.

A strong approach is to try out more than one tool as you build and evaluate models for any particular goal. Not all tools are of equal quality; you will find some to be generally much more effective than others, but among those you find to be good choices, likely you'll keep several around.

## Tools for Deep Learning

Take deep learning, for example. Deep learning, a specialized subarea of machine learning, is getting a lot of attention lately, and for good reason. This is an over simplified description, but deep learning is a method that does learning in a hierarchy of layers—the output of decisions from one layer feeds the decisions of the next. The most commonly used style of machine learning used in deep learning is patterned on the connections within the human brain, known as neural networks. Although the number of connections in a human-designed deep learning system is enormously smaller than the staggering number of connections in the neural networks of a human brain, the power of this style of decision-making can be similar for particular tasks.

Deep learning is useful in a variety of settings, but it's especially good for image or speech recognition. The very sophisticated math behind deep learning approaches and tools can, in many cases, result in a surprisingly simple and accessible experience for the practitioner using these new tools. That's part of the reason for their exploding popularity. New tools specialized for deep learning include TensorFlow (originally developed by Google), MXNet (a newly incubating Apache Software Foundation project with strong support from Amazon), and Caffe (which originated with work of a PhD student and others at the UC Berkeley Vision and Learning Center). Another widely used machine learning technology with broader applications, H2O, also has effective deep learning algorithms (it was developed by data scientist Arno Candel and others).

Although there is no single "best" specialized machine learning tool, it is important to have an overall technology that effectively handles data flow and model management for your project. In some ways, the best tool for machine learning is the data platform you use to deal with the logistics.

## Fundamental Needs Cut Across Different Projects

Just because it's common to work with multiple machine learning tools doesn't mean you need to change the underlying technology you use to handle logistics with each different situation. There are some fundamental requirements that cut across projects; regardless of the tool or tools you use for machine learning or even what types of models you build, the problems of logistics are going to be nearly the same.

Many aspects of the logistics of data flow and model management can best be handled at the data-platform level rather than the application level, thus freeing up data scientists and data engineers to focus more on the goals of machine learning itself.

**NOTE**

With the right capabilities, the underlying data platform can handle the logistics across a variety of machine learning systems in a unified way.

Machine learning model management is a serious business, but before we delve into the challenges and discover some practical solutions, first let's have some fun.

# Tensors in the Henhouse

Internet of Things (IoT) sensor data, deep learning image detection, and chickens—these are not three things you'd expect to find together. But a recent machine learning project designed and built by our friend and colleague, Ian Downard, put them together into what he described as "an over-engineered attempt" to detect blue jays in his hens' nesting box and chase the jays away before they break any eggs. Here's what happened.

The excitement and lure of deep learning using TensorFlow took hold for Ian when he heard a presentation at Strata San Jose by Google developer evangelists. In a recent blog, Ian reported that this presentation was, to a machine learning novice such as himself, "... nothing less than jaw dropping." He got the itch to try out TensorFlow himself. Ian is a skilled data engineer but relatively new to machine learning. Even so, he plunged in to build a predator detection system for his henhouse—a fun project, and a good way to do a proof-of-concept and get a little experience with tensor computation. It's also a simple example that we can use to highlight some of the concerns you will face in more serious real-world projects.

The fact that Ian could do this himself shows the surprising accessibility of working with tensors and TensorFlow, despite the sophistication of how they work. This instance is, of course, a sort of toy project, but it does show the promise of these methods.

## Defining the Problem and the Project Goal

The goal is to protect eggs against attack by blue jays. The specific goal for the machine learning step is to detect motion that activates the system and then differentiate between chickens and jays, as shown in Figure 1-1. This project had a limited initial goal: just to be able to detect jays. How to act on that knowledge in order to protect eggs is yet to come.


*Figure 1-1. Image recognition using TensorFlow is at the heart of this henhouse-intruder detection project. Results are displayed via Twitter feed @TensorChicken (Tweets seem appropriate for a bird-based project.)*

### Lesson

It's important to recognize what data is available to be collected, how decisions can be structured, and to define a sufficiently narrow goal so that it is practical to carry out. Note that domain knowledge—such as, the predator is a blue jay—is critical to the effectiveness of this project.

### Planning and design

Machine learning uses an image classification system that reacts to motion detection. The deployed prototype works this way: movement is detected via a camera connected to a Raspberry Pi using an application called Motion. This triggers classification of the captured image by a TensorFlow model that has been deployed to the Pi. A Twitter feed (@TensorChicken) displays the top three scores; in the example shown in Figure 1-1, a Rhode Island Red chicken has been correctly identified.

For training during development, several thousand images captured from the webcam were manually saved as files in directories labeled according to categories to be used by the classification model. For the model, Ian took advantage of a pre-built TensorFlow called Inception v3 that he customized using the henhouse training images. Figure 1-2 shows the overall project design.

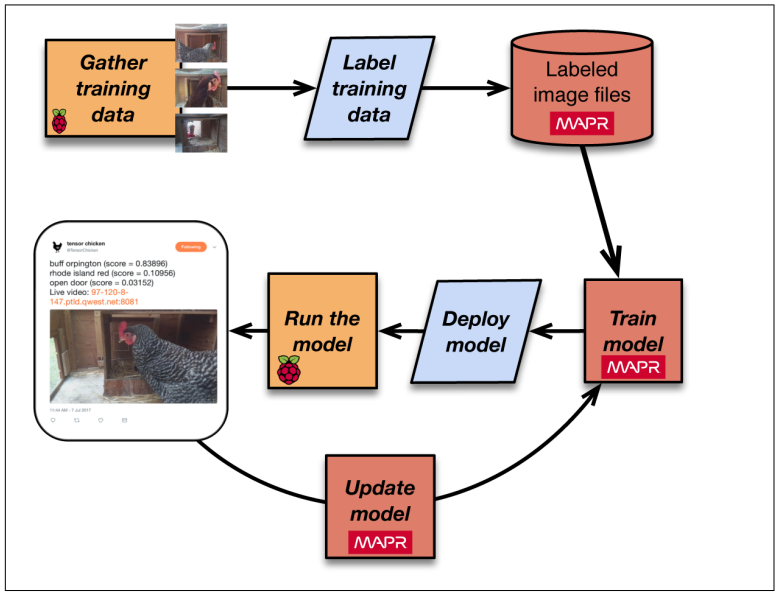



*Figure 1-2. Data flow for a prototype blue jay detection project using tensors in the henhouse. Details are available on the Big Endian Data blog (image courtesy of Ian Downard).*

### Lesson

The design provides a reasonable way to collect data for training, takes advantage of simplified model development by using Inception-v3 because it is sufficient for the goals of this project, and the model can be deployed to the IoT edge.

### SLAs

One issue with the design, however, is that the 30 seconds required for the classification step on the Pi are probably too slow to detect the blue jay in time to take an action to stop it from destroying eggs. That's an aspect of the design that Ian is already planning to address

by running the model on a MapR edge cluster (a small footprint cluster) that can classify images within 5 seconds.

### Retrain/update the model

A strength of the prototype design for this toy project is that it takes into account the need to retrain or update new models that will be in line to be deployed as time passes. See Figure 1-2. One potential way to do this is to make use of social responses to the Twitter feed @TensorChicken, although details remain to be determined.

### Lesson

Retraining or updating models as well as testing and rolling out entirely new models is an important aspect of successful machine learning. This is another reason that you will need to manage multiple models, even for a single project. Also note the importance of domain knowledge: After model deployment, Ian realized that some of his chickens were not of the type he thought. The model had been trained to erroneously identify some chickens as Buff Orpingtons. As it turns out, they are Plymouth Rocks. Ian retrained the model, and this shift in results is used as an example in Chapter 7.

### Expanding project goals

Originally Ian just planned to classify images for the type of bird (jay or type of chicken), but soon he wanted to expand the scope to know whether or not the door was open and when the nest is empty.

### Lesson

The power of machine learning often leads to mission creep. After you see what you can do, you may begin to notice new ways that machine learning can produce useful results.

## Real-World Considerations

This small tensor-in-the-henhouse project was useful as a way to get started with deep learning image detection and the requirements of building a machine learning project, but what would happen if you tried to scale this to a business-level chicken farm or a commercial enterprise that supplies eggs from a large group of farms to retail outlets? As Ian points out in his blog:

> Imagine a high-tech chicken farm where potentially hundreds of chickens are continuously monitored by smart cameras looking for predators, animal sickness, and other environmental threats. In scenarios like this, you'll quickly run into challenges...

Data scale, SLAs, a variety of IoT data sources and locations as well as the need to store and share both raw data and outputs with multiple applications or teams, likely in different locations, all complicate the matter. The same issues are true in other industries. Machine learning in the real world requires capable management of logistics, a challenge for any DataOps team. (If you're not familiar with the concept of DataOps, don't worry, we describe it in Chapter 2).

People new to machine learning may think of model management, for instance, as just a need to assign versions to models, but it turns out to be much more than that. Model management in the real world is a powerful process that deals with large-scale changing data and changing goals, and with ways to deal with models in isolation so that they can be evaluated in specifically customized, controlled environments. This is a fluid process.

## Myth of the Unitary Model

A persistent misperception in machine learning, particularly by software engineers, is that the project consists of building a single successful model, and after it is deployed, you're done. The real situation is quite different. Machine learning involves working with many models, even after you've deployed a model into production—it's common to have multiple models in production at any given time. In addition, you'll have new models being readied to replace production models as situations change. These replacements will have to be done smoothly, without interruptions to service if possible. In development, you'll work with more than one model as you experiment with multiple tools and compare models. That is what you have with a single project, and that's multiplied in other projects across the organization, maybe a hundred-fold.

One of the major causes for the need for so many models is mission creep. This is an unavoidable cost of fielding a successful model; once you have a one win, you will be expected to build on it and repeat it in new areas. Pretty soon, you have models depending on models in a much more complex system than you planned for initially.

The innovative architecture described in this book can be a key part of a solution that meets these challenges. This architecture must take into account the pragmatic business-driven concerns that motivate many aspects of model management.

## What Should You Ask about Model Management?

As you read this book, think about the machine learning projects you currently have underway or that you plan to build, and then ask how a model management system would effectively handle the logistics, given the solutions we propose. Here's a sample of questions you might ask:

- Is there a way to save data in raw-ish form to use in training later models? You don't always know what features will be valuable as you move forward. Saving raw data preserves data characteristics valuable for multiple projects.
- Does your system adequately and conveniently support multitenancy, including sharing the same data without interference?
- Do you have a way to efficiently deploy models and share data across data centers or edge processing in different locations, on premises, in cloud, or with a hybrid design?
- Is there a way to monitor and evaluate performance in development as well as to compare models?
- Can your system deploy models to production with ongoing validation of performance in this setting?
- Can you stage models into the production system for testing without disturbing system operation?
- Does your system easily handle hot hand-offs so new models can seamlessly replace a model in production?
- Do you have automated fall back? (for instance, if a model is not responding within a specified time, is there an automated step that will go to a secondary model instead?)
- Are your models functioning in a precisely specified and documented environment?

The recipe for meeting these requirements is the rendezvous architecture. Chapter 2 looks at some of the ingredients that go into that recipe.

CHAPTER 2

# What Matters in Model Management

The logistics required for successful machine learning go beyond what is needed for other types of software applications and services. This is a dynamic process that should be able to run multiple production models, across various locations and through many cycles of model development, retraining, and replacement. Management must be flexible and quickly responsive: you don't want to wait until changes in the outside world reduce performance of a production system before you begin to build better or alternative models, and you don't want delays when it's time to deploy new models into production.

All this needs to be done in a style that fits the goals of modern digital transformation. Logistics should not be a barrier to fast-moving, data-oriented systems, or a burden to the people who build machine learning models and make use of the insights drawn from them. To make it much easier to do these things, we introduce the *rendezvous architecture* for management of machine learning logistics.

## Ingredients of the Rendezvous Approach

The rendezvous architecture takes advantage of data streams and geo-distributed stream replication to maintain a responsive and flexible way to collect and save data, including raw data, and to make data and multiple models available when and where needed. A key feature of the rendezvous design is that it keeps new models

warmed up so that they can replace production models without significant lag time. The design strongly supports ongoing model evaluation and multi-model comparison. It's a new approach to managing models that reduces the burden of logistics while providing exceptional levels of monitoring so that you know what's happening.

Many of the ingredients of the rendezvous approach—use of streams, containers, a DataOps style of design—are also fundamental to the broader requirements of building a global data fabric, a key aspect of digital transformation in big data settings. Others, such as use of decoy and canary models, are specific elements for machine learning.

With that in mind, in this chapter we explore the fundamental aspects of this approach that you will need in order to take advantage of the detailed architecture presented in Chapter 3.

## DataOps Provides Flexibility and Focus

New technologies offer big benefits, not only to work with data at large scale, but also to be able to pivot and respond to real-world events as they happen. It's imperative, then, to not limit your ability to enjoy the full advantage of these emerging technologies just because your business hasn't also evolved its style of work. Traditionally siloed roles can prove too rigid and slow to be a good fit in big data organizations undergoing digital transformation. That's where a DataOps style of work can help.

The DataOps approach is an emerging trend to capture the efficiency and flexibility needed for data-driven business. DataOps style emphasizes better collaboration and communication between roles, cutting across skill guilds to enable teams to move quickly, without having to wait at each step for IT to give permissions. It expands the DevOps philosophy to include not only specialists in software development and operations, but also data-heavy roles such as data engineering and data science. As with DevOps, architecture and product management roles also are part of the DataOps team.

**NOTE**

A DataOps approach improves a project's ability to stay on time and on target.

Not all DataOps teams will include exactly the same roles, as shown in Figure 2-1; overall goals direct which functions are needed for that particular team. Organizing teams across traditional silos does not increase the total size of the teams, it just changes the most-used communication paths. Note that the DataOps approach is about organizing around data-related goals to achieve faster time to value. DataOps is does *not* require adding additional people. Instead, it's about improving collaboration between skill sets for efficiency and better use of people's time and expertise.

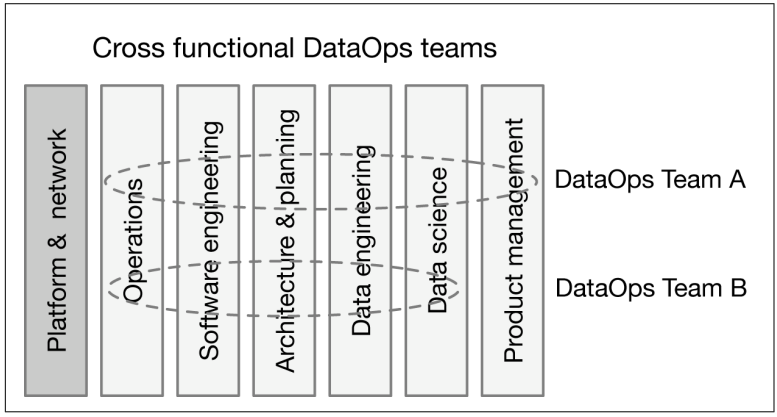



*Figure 2-1. DataOps team members fill a variety of roles notably including data engineering and data science. This is a cross-cutting organization that breaks down skill silos.*

Just as each DataOps team may include a different subset of the potential roles for working with data, teams also differ as to how many people fill the roles. In the tensor chicken example presented in Chapter 1, one person stretched beyond his usual strengths in software engineering to cover all required roles for this toy project—he was essentially a DataOps team of one. In contrast, in real-world business situations, it's usually best to draw on the specialties of multiple team members. In large-scale projects, a particular DataOps role may be filled by more than one person, but it's also common that some people will cover more than one role. Operations and software engineering skills may overlap; team members with software engineering experience also may be qualified as data engineers. Often, data scientists have data engineering skills. It's rare, however, to see overlap between data science and operations. These are not

meant as hard hard-edged definitions but; rather, they are suggestions for how to combine useful skills for data-oriented work.

What generally lies outside the DataOps roles? Infrastructural capabilities around data platform and network—needs that cut across all projects—tend to be supported separately from the DataOps teams by support organizations, as shown in Figure 2-1.

What all DataOps teams share is a common goal: the data-driven needs of the services they support. This combination of skills and shared goals enhance both the flexibility needed to adjust to changes as situations evolve and the focus needed to work efficiently, making it more feasible to meet essential SLAs.

DataOps is an approach that is well suited to the end-to-end needs of machine learning, For example, this style makes it more feasible for data scientists to have the support of software engineering to provide what is needed when models are handed over to operations during deployment.

The DataOps approach is not limited to machine learning. This style of organization is useful for any data-oriented work, making it easier to take advantage of the benefits offered by building a global data fabric, as described later in this chapter. DataOps also fits well with a widely popular architectural style known as *microservices*.

## Stream-Based Microservices

Microservices is a flexible style of building large systems whose value is broadly recognized across industries. Leading companies, including Google, Netflix, LinkedIn, and Amazon, demonstrate the advantages of adopting a microservices architecture. Microservices enables faster movement and better ability to respond in a more agile and appropriate way to changing business needs, even at the detailed level of applications and services.

What is required at the level of technical design to support a microservices approach? Independence between microservices is key. Services need to interact via lightweight connections. In the past, it has often been assumed that these connections would use RPC mechanisms such as REST that involve a call and almost immediate response. That works, but a more modern, and in many ways more advantageous, method to connect microservices is via a *message stream*.

Stream transport can support microservices if it can do the following:

- Support multiple data producers and consumers
- Provide message persistence with high performance
- Decouple producers and consumers

It's fairly obvious in a complex, large-scale system why the message transport needs to be able to handle data from multiple sources (data producers) and to have multiple applications running that consume that data, as shown in Figure 2-2. However, the other needs can be, at first glance, less obvious.

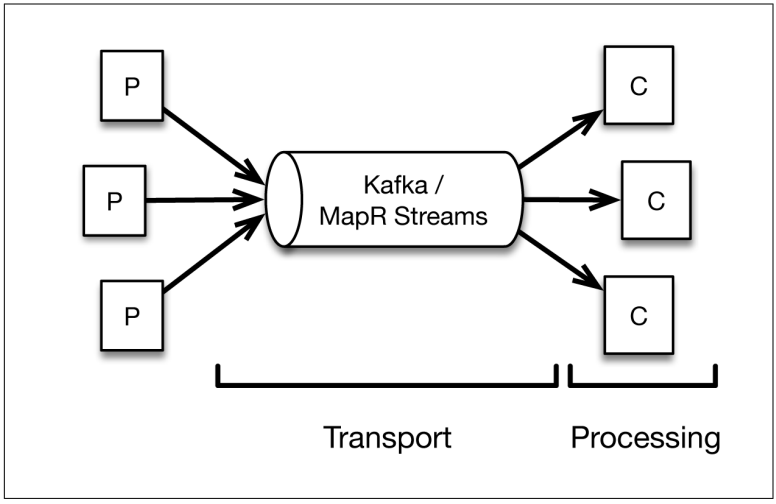



*Figure 2-2. A stream-based design with the right choice of stream transport supports a microservices-style approach.*

Clearly you want a high-performance system, but why do you need message persistence? Often when people think of using streaming data, they are concerned about some real-time or low-latency application, such as updating a real-time dashboard, and they may assume a "use it and lose it" attitude toward the streaming data involved. If so, they likely are throwing away some real value, because other groups or even themselves in future projects might need access to that discarded data. There are a number of reasons to want durable messages, but foremost in the context of microservices is that message persistence is required to decouple producers and consumers.

**NOTE**

A stream transport technology that decouples producers from consumers offers a key capability needed to take advantage of a flexible microservices-style design.

Why is having durable messages essential for this decoupling? Look again at Figure 2-2. The stream transport technologies of interest *do not broadcast* message data to consumers. Instead, consumers subscribe to messages on a topic-by-topic basis. Streaming data from the data sources is transported and made available to consumers immediately—a requirement for real-time or low-latency applications—*but the message does not need to be consumed right away*. Thanks to message persistence, consumers don't need to be running at the moment the message appears; they can come online later and still be able to use data from earlier events. Consumers added at a later time don't interfere with others. This independence of consumers from one another and from producers is crucial for flexibility.

Traditionally, stream transport systems have had a trade-off between performance and persistence, but that's not acceptable for modern stream-based architectures. Figure 2-2 lists two modern stream transport technologies that deliver excellent performance along with persistence of messages. These are Apache Kafka and MapR Streams, which uses the Kafka API but is engineered into the MapR converged data platform. Both are good choices for stream transport in a stream-first architecture.

## Streams Offer More

The advantages of a stream-first design go beyond just low-latency applications. In addition to support for microservices, having durable messages with high performance is helpful for a variety of use cases that need an event-by-event replayable history. Think of how useful that could be when an insurance company needs an auditable log, or someone doing anomaly detection as part of preventive maintenance in an industrial setting wants to replay data from IoT sensors for the weeks leading up to a malfunction.

Data streams are also excellent for machine learning logistics, as we describe in detail in Chapter 3. For now, one thing to keep in mind is that a stream works well as an immutable record, perhaps even better than a database.

**NOTE**

Databases were made for updates. Streams can be a safer way to persist data if you need an exact copy of input or output data for a model.

Streams are also a useful way to provide raw data to multiple consumers, including multiple machine learning models. Recording raw data is important for machine learning—don't discard data that might later prove useful.

We've written about the advantages of a stream-based approach in the book *Streaming Architecture: New Designs Using Apache Kafka and MapR Streams* (O'Reilly, 2016). One advantage is the role of streaming and stream replication in building a *global data fabric*.

## Building a Global Data Fabric

As organizations expand their use of big data across multiple lines of business, they need a highly efficient way to access a full range of data sources, types and structures, while avoiding hidden data silos. They need to have fine-grained control over access privileges and data locality without a big administrative burden. All this needs to happen in a seamless way across multiple data centers, whether on premises, in the cloud, or in a highly optimized hybrid architecture, as suggested in Figure 2-3. What is needed is something that goes beyond and works much better than a data lake. The solution is a global data fabric.

Preferably the data fabric you build is managed under uniform administration and security, with fine-grained control over access privileges, yet each approved user can easily locate data—each "thread" of the data fabric can be accessed and used, regardless of geo-location, on premises, or in cloud deployments.

Geo-distributed data is a key element in a global data fabric, not only for remote back-up copies of data as part of a disaster recovery plan, but also for day-to-day functioning of the organization and data projects including machine learning. It's important for different groups and applications in different places to be able to simultaneously use the same data.

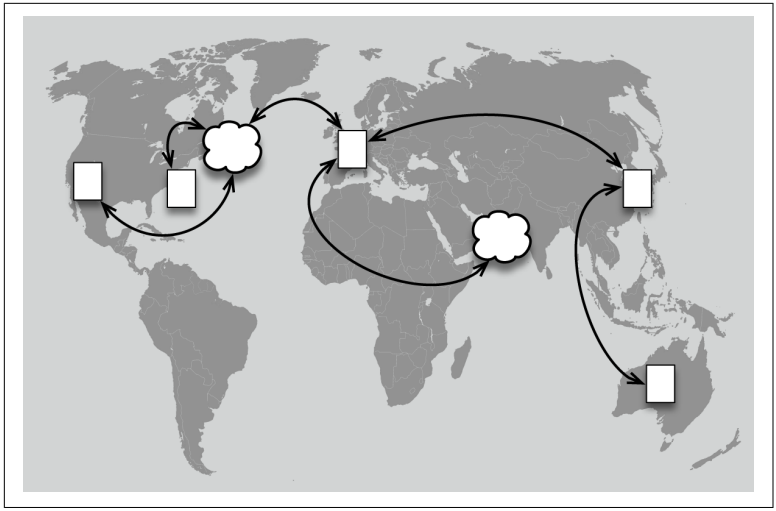


*Figure 2-3. A global data fabric provides an organization-wide view of data, applications and operations while making it easy to find exactly the data that is needed. (Reprinted from the O'Reilly data report "Data Where You Want It: Geo-Distribution of Big Data and Analytics," © 2017 Dunning and Friedman.)*

How can you do that? One example of a technology uniquely designed with the capabilities needed to build a large-scale global data fabric is the MapR converged data platform. MapR has a range of data structures (files, tables, message streams) that are all part of a single technology, all under the same global namespace, the same administration, and security. Streaming data can be shared through subscription by multiple consumers, and because MapR streams provide unique multi-master, omni-directional stream replication, streaming data is also shared across different locations, in cloud or on premises. In other words, you can focus on what your application is supposed to do, regardless of where the data source is located.

Administrators don't need to worry about what each developer or data scientist is building—they can focus on their own concerns of maintaining the system, controlling access and data location as needed, all under one security system. Similarly, MapR's direct table replication also contributes to this separation of concerns in building a global data fabric. Efficient mirroring of MapR data volumes with incremental updates goes even further to provide a way to

extend the data fabric through replication of files, tables, and streams at regular, configurable intervals.


A global data fabric suits the DataOps style of working and is a big advantage for the management of multiple applications, including machine learning models in development and production.

With a global data fabric, applications also need to run where you want them. The ability to deploy applications easily in predictable and repeatable environments is greatly assisted by the use of containers.

## Making Life Predictable: Containers

It's not surprising that you might encounter slight differences in conditions as you go from development to production or in running an application in a new location—but even small differences in the application version itself, or any of its dependencies, can result in big, unexpected, and generally unwanted differences in the behavior of your application. Here's where the convenience of launching applications in containers comes to the rescue.

A container behaves like just another process running on a system, but containerization is a much lighter-weight approach than virtualization. It's not necessary, for instance, to run a copy of the operating system in a container, as would be true for a virtual machine. With containers, you get environmental consistency that does away with surprises. You provide the same conditions for running your application in a variety of different situations, making its behavior predictable and repeatable. You can package, distribute, and run the exact bits that make up applications (including machine learning models) along with their dependencies in a carefully curated environment. Containers are a good fit to flexible approaches, useful for cloud deployments, and help you build a global data fabric. They're particularly important for model management.

To keep containers lightweight, however, there is a challenge regarding data that needs to live beyond the lifetime of the container. Storing data in the container, especially at scale, is generally not a good practice. You want to keep containers stateless, yet at times you need to run stateful applications, including machine learning models.

How, then, can you have stateless containers running stateful applications? A solution, shown in Figure 2-4, is for containers to access and persist data directly to a data platform. Note that applications running in the containers can communicate with each other directly or via the data platform.

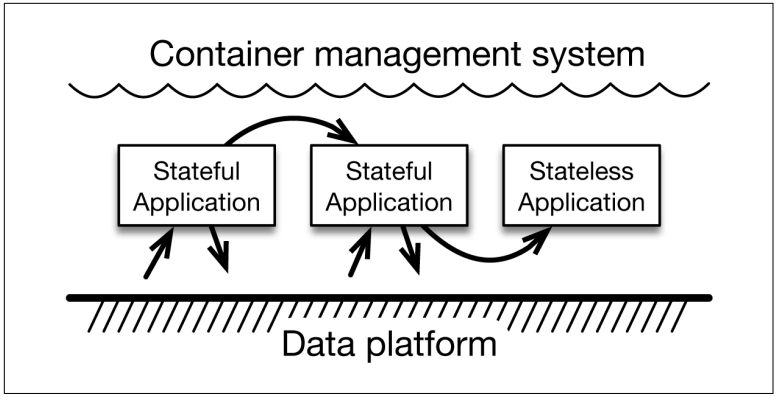



*Figure 2-4. Containers can remain stateless even when running stateful applications if there is data flow to and from a platform. (Based on "Data Where You Want It: Geo-Distribution of Big Data and Analytics.)*

Scalable datastores such as Apache Cassandra could serve the purpose of persistence, although it is limited to one data type (tables). Files could be persisted to a specialized distributed file system. Hadoop Distributed File System (HDFS), however, has some limitations on read performance for container-based applications. An alternative to either of these is the MapR Converged Data Platform that not only easily handles data persistence as tables or files at scale, it also offers the option of persistence to message streams. For more detail on running stateful container-based applications with persistence to an underlying platform, see the data report "Data Where You Want It: Geo-Distribution of Big Data and Analytics".

## Canaries and Decoys

So far, we've talked about issues that not only matter for model management but are also more broadly important in working with big data. Now, let's take a look at a couple of challenges that are specific for machine learning.

The first issue is to have a way to accurately record the input data for a model. It's important to have an exact and replayable copy of input data. One way to do this is to use a *decoy model*. The decoy is a service that appears to be a machine learning model but isn't. The decoy sits in exactly the same position in a data flow as the actual model or models being developed, but the decoy doesn't do anything except look at its input data and record it, preferably in a data stream. Chapter 4 describes in detail the use of decoy models as a key part of the rendezvous architecture.

Another challenge for machine learning is to have a baseline reference for the behavior of a model in production. If a model is working well, perhaps at 90 percent of whatever performance is possible, introducing a new model usually should produce only a relatively small change even if it is a desirable improvement. It would be useful, then, to have a way to alert you to larger changes that may signal something has been altered or gone wrong in your data chain for instance. Perhaps customers are behaving differently such that customer-based-data is very different.

The solution is to deploy a *canary model*. The idea behind a canary is simple and harkens back to earlier times when miners carried a canary (a live bird, not software) into a mine as a check for good air. Canaries are particularly sensitive to toxic gases, so as long as the canary was still alive, the air was assumed to be safe for humans. The good news is that the use of a canary model in machine learning is less cruel but just as effective. The canary model runs alongside the current production model, providing a reference for baseline performance against which multiple other models can be measured.

# Types of Machine Learning Applications

The logistical issues we discuss apply to essentially all types of machine learning applications, but the solutions we propose—in particular the rendezvous architecture—are a best fit for a *decisioning* type of machine learning. To help you recognize how this relates to your own projects, here's a brief survey of machine learning categories drawn with a broad brush.

## Decisioning

Machine learning applications that fall under the description of "decisioning" basically seek a "correct answer." Out of a short list of

answers, we hope number one is correct, and maybe partial credit for number two, and so on. Decisioning systems involve a query—response pattern with bounded input data and bounded output. They are typically built using supervised learning and usually require human judgments to build the training data.

You can think of decisioning applications as being further classified into two styles: discrete or flow. Discrete probably is more familiar. You have a single query followed by a single response, often in the moment, that goes back to the origin of the query. This is then repeated. With the flow style, there is a continuous stream of queries and the corresponding responses go back to a stream. The architecture we propose as a solution for managing models is a streaming system packaged to make it look discrete.

Use cases that fall in the decisioning category include transaction fraud detection projects such as those based on credit cards, credit risk analysis of home mortgage applications, and identifying potential fraud in medical claims. Decisioning projects also include marketing response prediction (predictive analytics), churn prediction, and detection of energy theft from a smart meter. Deep learning systems for image classification or speech recognition also can be viewed as decisioning systems.

## Search-Like

Another category of applications involves search or recommendations. These projects use bounded input and return a ranked list of results. Multiple answers in the list may be valid—in fact the goal of search is often to provide multiple desired results. Use cases involving search-like or recommendation-based applications include automated website organization, ad targeting or upsell, and product recommendation systems for retail applications. Recommendation is also used to customize web experience in order to encourage a user to spend more time on a website.

## Interactive

This last broad category contains systems that tend to be more complex and often require even higher level of sophistication than those we've already described. Answers are not absolute; the validity of the output generally depends on context, often in real-world and rapidly changing situations. These applications use continuous input and

involve autonomous worldly interactions. They may also use reinforcement learning. The actions of the system also determine what future actions are possible.

Examples include chat bots, interactive robots and autonomous cars. For the latter, a response such as "turn right" may or may not be correct, depending on the context and the exact moment. In the case of self-driving cars, the interactive nature of these applications involve a great deal of continuous input from the environment and the car's own internal systems in order respond in the moment. Other use cases involve sophisticated anomaly detection and alerting, such as web system anomalies, security intrusion detection or even predictive maintenance.

## Conclusion

All of these categories of machine learning applications could benefit from some aspects of the solutions we describe next, but for search-like projects or sophisticated interactive machine learning, the rendezvous architecture will likely need to be modified to work well. Solutions for model management for all these categories are beyond the scope of this short book. From here on, we focus on applications of the decisioning type.

CHAPTER 3

# The Rendezvous Architecture for Machine Learning

Rendezvous architecture is a design to handle the logistics of machine learning in a flexible, responsive, convenient, and realistic way. Specifically, rendezvous provides a way to do the following:

- Collect data at scale from a variety of sources and preserve raw data so that potentially valuable features are not lost
- Make input and output data available to many independent applications (consumers) even across geographically distant locations, on premises, or in the cloud
- Manage multiple models during development and easily roll into production
- Improve evaluation methods for comparing models during development and production, including use of a reference model for baseline successful performance
- Have new models poised for rapid deployment

The rendezvous architecture works in concert with your organization's global data fabric. It doesn't solve all of the challenges of logistics and model management, but it does provide a pragmatic and powerful design that greatly improves the likelihood that machine learning will deliver value from big data.

In this chapter, we present in detail an explanation of what motivates this design and how it delivers the advantages we've men-

tioned. We start with the shortcomings of previous designs and follow a design path to a more flexible approach.

## A Traditional Starting Point

When building a machine learning application, it is very common to want a *discrete response* system. In such a system, you pass all of the information needed to make some decision, and a machine learning model responds with a decision. The key characteristic is this synchronous response style.

For now, we can assume that there is nothing outside of the request needed by the model to make this decision. Figure 3-1 shows the basic architecture of a system like this.

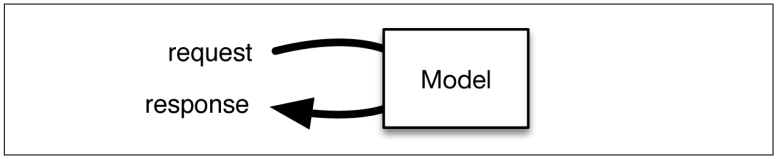



*Figure 3-1. A discrete response system, one in which a model responds to requests with decisions and poses problems that underline the need for the rendezvous architecture.*

This is very much like the first version for the henhouse monitoring system described in Chapter 1. The biggest virtue of such a system is its stunning simplicity, which is obviously desirable.

That is its biggest vice, as well.

Problems crop up when we begin to impose some of the other requirements that are inherent in deploying machine learning models to production. For instance, it is common in such a system to require that we can run multiple models at the same time on the exact same data in order to compare their speed and accuracy. Another common requirement is that we separate the concerns of decision accuracy from system reliability guarantees. We obviously can't completely separate these, but it would be nice if our data scientists who develop the model could focus on science-y things like accuracy, with only broad-brush requirements around topics like redundancy, running multiple models, speed and absolute stability.

Similarly, it would be nice if the ops part of our DataOps team could focus more on guaranteeing that the system behaves like a solid,

well-engineered piece of software that isolates models from one another, always returns results on time, restarts failed processes, transparently rolls to new versions, and so on. We also want a system that meets operational requirements like deadlines and makes it easy to decide, manage, and change which models are in play.

## Why a Load Balancer Doesn't Suffice

The first thought that lots of people have when challenged with getting better operational characteristics from a basic discrete decision system (as in Figure 3-1) is to simply replicate the basic decision engine and put a load balancer in front of these decision systems. This is the standard approach for microservices, as shown in Figure 3-2. Using a load balancer solves some problems, but definitely not all of them, especially not in the context of *learned models*. With a load balancer, you can start and stop new models pretty easily, but you don't easily get latency guarantees, the ability to compare models on identical inputs, nor do you get records of all of the requests with responses from all live models.

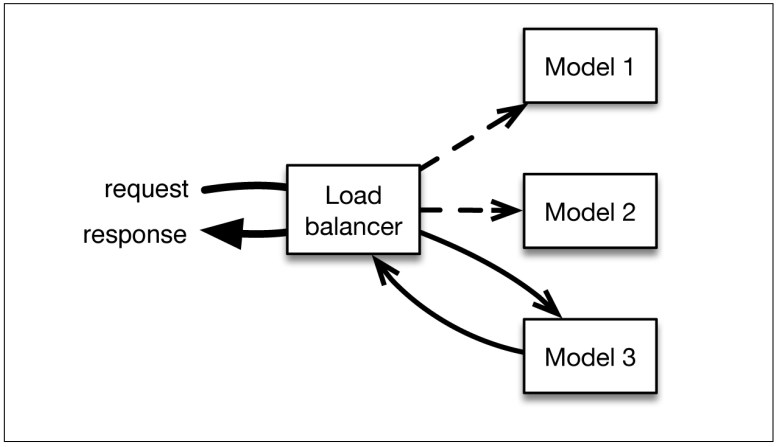



*Figure 3-2. A load balancer, in which each request is sent to one of the active models at a time, is an improvement but lacks key capabilities of the rendezvous style.*

Using a load balancer in front of a microservice works really well in many domains such as profile lookup, web servers, and content engines. So, why doesn't it work well for machine learning models?

The basic problem comes down to some pretty fundamental discrepancies between the nature and life cycle of conventional software services and services based on machine learning (or lots of other data-intensive services):

- The difference between revisions of machine learning models are often subtle, and we typically need to give the exact same input to multiple revisions and record identical results when making a comparison, which is inherently statistical in nature. Revisions of the software in a conventional microservice don't usually require that kind of extended parallel operation and statistical comparison.
- We often can't predict which of several new techniques will yield viable improvements within realistic operational settings. That means that we often want to run as many as a dozen versions of our model at a time. Running multiple versions of a conventional service at the same time is usually considered a mistake rather than a required feature.
- In a conventional DevOps team, we have a mix of people who have varying strengths in (software) development or operations. Typically, the software engineers in the team aren't all that bad at operations and the operations specialists understand software engineering pretty well. In a DataOps team, we have a broader mix of data scientists, software engineers, and operations specialists. Not only does a DataOps team cover more ground than a DevOps team, there is typically much less overlap in skills between data scientists and software engineers or operations engineers. That makes rolling new versions much more complex socially than with conventional software.
- Quite frankly, because of the wider gap in skills between data scientists and software or ops engineers, we need to allow for the fact that models will typically not be implemented with as much software engineering rigor as we might like. We also must allow for the fact that the framework is going to need to provide for a lot more data rigor than most software does in order to satisfy the data science part of the team.

These problems could be addressed by building a new kind of load balancer and depending heavily on the service discovery features of frameworks such as Kubernetes, but there is a much simpler path.

That simpler path is to use a stream-first architecture such as the rendezvous architecture.

## A Better Alternative: Input Data as a Stream

Now, we take the first step underlying the new design. As Chapter 2 explains, message streams in the style of Apache Kafka, including MapR Streams, are an ideal construct here because stream consumers control what and when they listen for data (pull style). That completely sidesteps the problem of service discovery and avoids the problem of making sure all request sources send all transactions to all models. Using a stream to receive the requests, as depicted in Figure 3-3, also gives us a persistent record of all incoming requests, which is really helpful for debugging and postmortem purposes. Remember that the models aren't necessarily single processes.

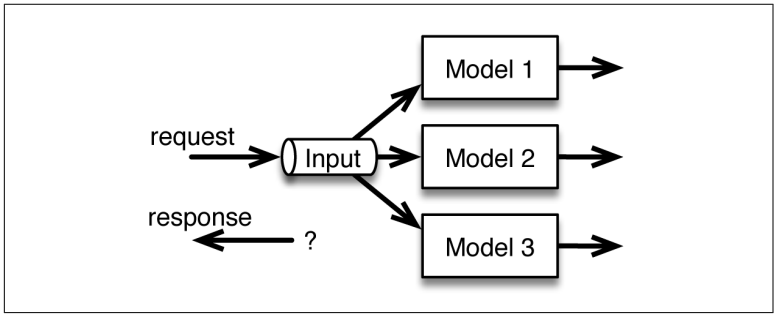



*Figure 3-3. Receiving requests via a stream makes it easy to distribute a request to all live models, but we need more machinery to get responses back to the source of requests.*

But we immediately run into the question that if we put the requests into a stream, how will the results come back? With the original discrete decision architecture in Figure 3-1, there is a response for every request and that response can naturally include the results from the model. On the other hand, if we send the requests into a stream and evaluate those requests with lots of models, the insertion into the input stream will complete before any model has even looked at the request. Even worse, with multiple models all producing results at different times, there isn't a natural way to pick which result we should return, nor is there any obvious way to return it. These additional challenges motivate the rendezvous design.

## Rendezvous Style

We can solve these problems with two simple actions. First, we can put a return address into each request. Second, we can add a process known as a rendezvous server that selects which result to return for each request. The return address specifies how a selected result can be returned to the source of the request. A return address could be an address of an HTTP address connected to a REST server. Even better, it can be the name of a message stream and a topic. Whatever works best for you is what it needs to be.

**NOTE** Using a rendezvous style works only if the streaming and processing elements you are using are compatible with your latency requirements.

For persistent message queues, such as Kafka and MapR Streams, and for processing frameworks, such as Apache Flink or even just raw Java, a rendezvous architecture will likely work well—down to around single millisecond latencies.

Conversely, as of this writing, microbatch frameworks such as Apache Spark Streaming will just barely be able to handle latencies as low as single digit seconds (not milliseconds). That might be acceptable, but often it will not be. At the other extreme, if you need to go faster than a few milliseconds, you might need to use nonpersistent, in-memory streaming technologies. The rendezvous architecture will still apply.

**NOTE** The key distinguishing feature in a rendezvous architecture is how the rendezvous server reads all of the requests as well as all of the results from all of the models and brings them back together.

Figure 3-4 illustrates how a rendezvous server works. The rendezvous server uses a policy to select which result to anoint as "official" and writes that official result to a stream. In the system shown, we assume that the return address consists of a topic and request identifier and that the rendezvous server should write the results to a well-known stream with the specified topic. The result should contain the request identifier to the process sending the request in the first place since it has the potential to send overlapping requests.

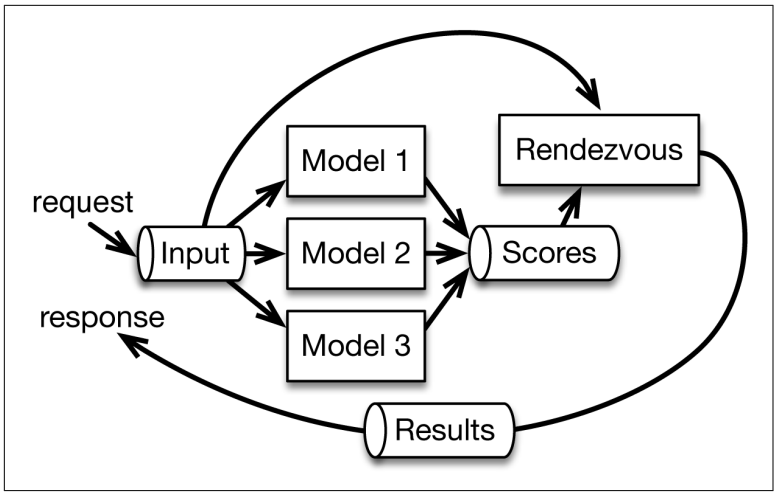



*Figure 3-4. The core rendezvous design. There are additional nuances, but this is the essential shape of the architecture.*

Internally, the rendezvous server works by maintaining a mailbox for each request it sees in the `input` stream. As each of the models report results into the `scores` stream, the rendezvous server reads these results and inserts them into the corresponding mailbox. Based on the amount of time that has passed, the priority of each model and possibly even a random number, the rendezvous server eventually chooses a result for each pending mailbox and packages that result to be sent as a response to the return address in the original request.

One strength of the rendezvous architecture is that a model can be "warmed up" before its outputs are actually used so that the stability of the model under production conditions and load can be verified. Another advantage is that models can be "deployed" or "undeployed" *simply by instructing the rendezvous server to stop (or start) ignoring their output.*

Related to this, the rendezvous can make guarantees about returning results that the individual models cannot make. You can, for instance, define a policy that specifies how long to wait for the output of a preferred model. If at least one of the models is very simple and reliable, albeit a bit less accurate, this simple model can be used as a backstop answer so that if more sophisticated models take too long or fail entirely, we can still produce some kind of answer before

a deadline. Sending the results back to a highly available message stream as shown in Figure 3-4 also helps with reliability by decoupling the sending of the result by the rendezvous server from the retrieving the result by the original requestor.

## Message Contents

The messages between the components in a rendezvous architecture are mostly what you would expect, with conventional elements like timestamp, request id, and request or response contents, but there are some message elements that might surprise you on first examination.

The messages in the system need to satisfy multiple kinds of goals that are focused around operations, good software engineering and data science. If you look at the messages from just one of these points of view, some elements of the messages may strike you as unnecessary.

All of the messages include a timestamp, message identifier, provenance, and diagnostics components. This makes the messages look roughly like the following if they are rendered in JSON form:

```
{
    timestamp: 1501020498314,
    messageId: "2a5f2b61fdd848d7954a51b49c2a9e2c",
    provenance: { ... },
    diagnostics: { ... },
    ... application specific data here ..
}
```

The first two common message fields are relatively self-explanatory. The timestamp should be in milliseconds, and the message identifier should be long enough to be confident that it is unique. The one shown here is 128 bits long.

The provenance section provides a history of the processing elements, including release version, that have touched this message. It also can contain information about the source characteristics of the request in case we want to drill down on aggregate metrics. This is particularly important when analyzing the performance and impact of different versions of components or different sources of requests. Including the provenance information also allows limited trace diagnostics to be returned to the originator of the request without having to look up any information in log files or tables.

The amount of information kept in the provenance section should be relatively limited by default to the information that you really need to return to the original caller. You can increase the level of detail by setting parameters in the diagnostics section. If tracing is enabled for a request, the provenance section will contain the trace parent identifier to allow latency traces to be knit back together when you want to analyze what happened during a particular query. Depending on your query rates, the fraction of queries that have latency tracing turned on will vary. It might be all queries or it might be a tiny fraction.

The diagnostics section contains flags that can override various environmental settings. These overrides can force more logging or change the fallback schedule that the rendezvous server uses to select different model outputs to be returned to the original requestor. If desired, you can even use the diagnostics element to do failure injection. The faults injected could include delaying a model result or simulating a fault in a system component or model. Fault injection is typically only allowed in QA systems for obvious reasons.

## Request Specific Fields

Beyond the common fields, every request messages includes a return address and the model inputs. These inputs are augmented with the external state information and are given identically to every model. Note that some model inputs such as images or videos can be too large or complex to carry in the request directly. In such cases, a reference to the input can be passed instead of the actual input data. The reference is often a filename if you have a distributed file system with a global namespace, or an object reference if you are using a system like S3.

The return address can be something as simple as topic name in a well known message stream. Using a stream to deliver results has lots of advantages, such as automatically logging the delivery of results so it is generally preferred over mechanisms such as REST endpoints.

## Output Specific Fields

The output from the models consists of score messages that include the original request identifier as well as a new message identifier and the model outputs themselves. The rendezvous server uses the origi-

nal request identifier to collect together results for a request in anticipation of returning a response. The return address doesn't need to be in the score messages, because the rendezvous server will get that from the original request.

The result message has whatever result is selected by the rendezvous server and very little else other than diagnostic and provenance data. The model outputs can have many forms depending on the details of how the model actually works.

## Data Format

The data format you use for the messages between components in a rendezvous architecture doesn't actually matter as much as you might think especially given the heat that is generated whenever you bring data format conventions up in a discussion. The cost of moving messages in inefficient formats including serializing and deserializing data is typically massively overshadowed by the computations involved in evaluating a model. We have shown the messages as if they were JSON, but that is just because JSON is easy to read in a book. Other formats such as Arrow, Avro, Protobuf, or OJAI are more common in production. There is a substantial advantage to self-describing formats that don't require a schema repository, but even that isn't a showstopper.

That being said, there is huge value in consensus about messaging formats. It is far better to use a single suboptimal format everywhere than to split your data teams into factions based on format. Pick a format that everybody likes, or go with a format that somebody else already picked. Either way, building consensus is the major consideration and dominates anything but massive technical considerations.

## Stateful Models

The basic rendezvous architecture allows for major improvements in the management of models that are pure functions, that is, functions that always give the same output if given the same input.

Some models are like that. For instance, the TensorChicken model described in Chapter 1 will recognize the same image exactly the same way no matter what else it has seen lately. Machine translation and speech recognition systems are similar. Only deploying a new model changes the results.

Other models are definitely not stateless. In general, we define a stateful model as any model whose output in response to a request cannot be computed just from that one request. In addition, it is useful to talk about "internal state," which can be computed from the past history of requests, and "external state," which uses information from the outside world or some other information system.

Card velocity is a great example of internal state. Many credit card fraud models look at where and when recent transactions happened. This allows them to determine how fast the card must have moved to get from one transaction to the next. This card velocity can help detect cloned cards. Card velocity is a stateful feature because it doesn't just depend on the current transaction; it also depends on the previous card-present transaction. Stateful data like this can depend on a single entity like a user, website visitor, or card holder, or it can be a collective value such as the number of transactions in the last five minutes or the expected number of requests estimated from recent rates. These are all examples of internal state because every model could compute its own version of internal state from the sequence of requests completely independently of any other model.

External state looks very different. For instance, the current temperature in the location where a user is located is an example of external state. A customer's current balance may also be an external state variable if a model doesn't get to see every change of balance. Outputs from other models are also commonly treated as external state.

In a rendezvous model, it is a best practice to add all external state to the requests that are sent to all models. This allows all external state to be recorded. Since external state can't be re-created from the input data alone, archiving all external state used by the models is critical for reproducibility.

For internal state, on the other hand, you have a choice. You can compute the internal state variables as if they were external state and add them to the requests for all models. This is good if multiple models are likely to use the same variables. Alternatively, you can have each model compute its own internal state. This is good if the computation of internal state is liable to change. Figure 3-5 shows how to arrange the computations.

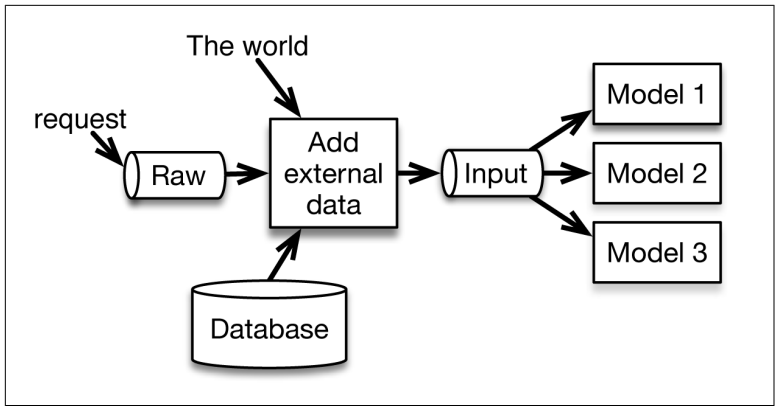



*Figure 3-5. With stateful models, all dependence on external state should be positioned in the main rendezvous flow so that all models get exactly the same state.*

The point here is that all external state computation should be external to all models. Forms of internal state that have stable and commonly used definitions can be computed and shared in the same way as external state or not, according to preference. As we have mentioned, the key rationale for dealing with these two kinds of state in this way is *reproducibility*. Dealing with state as described here means that we can reproduce the behavior of any model by using only the data that the decoy model has recorded and nothing else. The idea of having such a decoy model that does nothing but archive common inputs is described more fully in the next section.

## The Decoy Model

Nothing is ever quite as real as real data. As a result, recording live input data is extraordinarily helpful for developing and evaluating machine learning models. This doesn't seem at first like it would be much of an issue, and it is common for new data scientists to make the mistake of trusting that a database or log file is a faithful record of what data was or would have been given to a model. The fact is, however, that all kinds of distressingly common events can conspire to make reconstructed input data different from the real thing. The upshot is that you really can't expect data collected as a byproduct of another function to be both complete and correct. Just as unit tests and integration tests in software engineering are used to isolate dif-

ferent kinds of error to allow easier debugging, recording real data can isolate data errors from modeling errors.

The simplest way to ensure that you are seeing exactly what the models are seeing is to add what is called a *decoy model* into your system. A decoy model looks like a model and accepts data like any other model, but it doesn't emit any results (that is, it looks like a duck and floats like a duck, but it doesn't quack). Instead, it just archives the inputs that it sees. In a rendezvous architecture, this is really easy, and it is also really easy to be certain that the decoy records exactly what other models are seeing because the decoy reads from the same input stream.

Figure 3-6 shows how a decoy model is inserted into a rendezvous architecture.

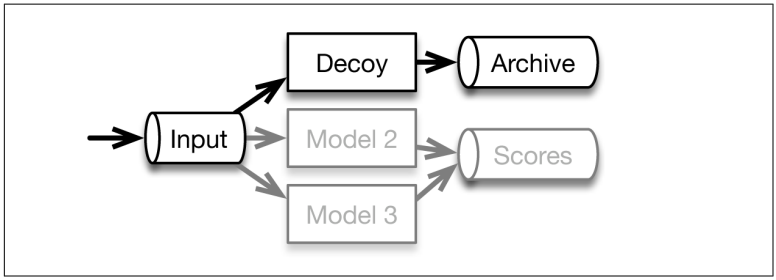



*Figure 3-6. A decoy model inserted into rendezvous architecture doesn't produce results. It just archives inputs that all models see, including external state.*

A decoy model is absolutely crucial when the model inputs contain external state information from data sources such as a user profile database. When this happens, this external data should be added directly into the model inputs using a preprocessing step common to all models, as described in the previous section. If you force all external state into all requests and use a decoy model, you can know *exactly* what external state the models saw. Handling external state like this can resolve otherwise intractable problems with race conditions between updates to external state and the evaluation of a model. Having a decoy that gets exactly the same input as every other model avoids that kind of question.

## The Canary Model

Another best practice is to always run a canary model even if newer models provide more accuracy or better performance. The point of the canary is not really to provide results (indeed, the rendezvous server will probably be configured to always ignore the canary). Instead, the canary is intended to provide a scoring baseline to detect shifts in the input data and as a comparison benchmark for other models. Figure 3-7 shows how the canary model can be used in the rendezvous architecture.

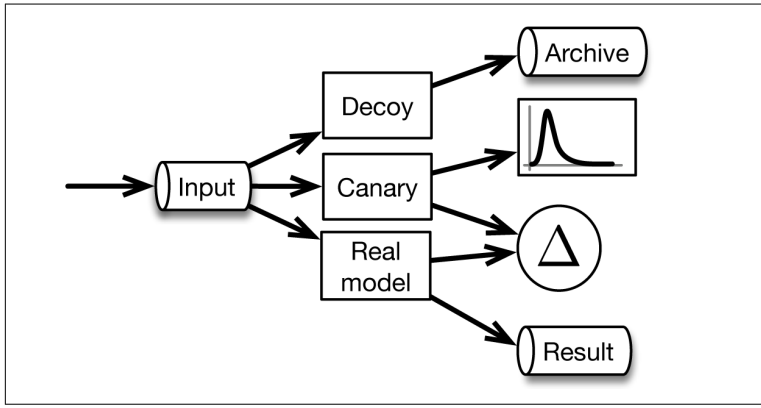



*Figure 3-7. Integrating a canary model into the rendezvous architecture provides a useful comparison for baseline behavior both for input data and other models.*

For detecting input shifts, the distribution of outputs for the canary can be recorded and recent distributions can be compared to older distributions. For simple scores, distribution of score can be summarized over short periods of time using a sketch like the *t*-digest. These can be aggregated to form sketches for any desired period of time, and differences can be measured (for more information on this, see Chapter 7). We then can monitor this difference over time, and if it jumps up in a surprising way, we can declare that the canary has detected a difference in the inputs.

We also can compare the canary directly to other models. As a bonus, we not only can compare aggregated distributions, we can use the request identifier to match up all the model results and compare each result against all others.

NOTE

For more ideas about how to compare models, see Chapter 5.

It might seem a bit surprising to compare new models against a very old model instead of each other. But the fact of the canary's age is exactly why it is so useful. Over time, every new model will have been compared to the canary during the preproduction checkout and warm-up period. This means that the DataOps teams will have developed substantial experience in comparing models to the canary and will be able to spot anomalies quickly. When new models are compared to new models, however, you aren't quite as sure what to expect precisely because neither of the models has much track record.

## Adding Metrics

As with any production system, reporting metrics on who is doing what to whom, and how often, is critical to figuring out what is really going on in the system. Metrics should not be (but often are) an afterthought to be added after building an entire system. Good metrics are, however, key to diagnosing all kinds of real-world issues that crop up, be they model stability issues, deployment problems, or problems with the data logistics in a system, and they should be built in from the beginning.

With ordinary microservices, the primary goal of collecting metrics is to verify that a system is operating properly and, if not, to diagnose the problem. Problems generally have mostly to do with whether or not a system meets service-level agreements.

With machine learning models, we don't just need to worry about operational metrics (did the model answer, was it quick enough?); we also need to worry about the accuracy of the model (when it did answer, did it give the right answer?). Moreover, we usually expect a model to have some error rate, and it is normal for accuracy to degrade over time, especially when the model has real-world adversaries, as in fraud detection or intrusion detection. In addition, we need to worry about whether the input data has changed in some way, possibly by data going missing or by a change in the distribution of incoming data. We go into more detail on how to look for data changes in Chapter 7.

To properly manage machine learning models, we must collect metrics that help us to understand our input data and how our models are performing on both operational and accuracy goals. Typically, this means that we need to record operational metrics to answer operational questions and need to record scores for multiple models to answer questions about accuracy. Overall, there are three kinds of questions that need to be answered:

- Across all queries, possibly broken down by tags on the requests, what are the aggregate values of certain quantities like number of requests and what is the distribution of quantities like latency? What are the properties of our inputs and outputs in terms of distribution? We answer these questions with aggregate metrics.
- On some subset of queries, what are the specific times taken for every step of evaluation? Note that the subset can be all queries or just a small fraction of all queries. We can answer these questions with latency traces.
- On a large number of queries, what are the exact model outputs broken down by model version for each query? This will help us compare accuracy of one model versus another. We answer these questions by archiving inputs with the decoy server and outputs in the `scores` stream.

The first kind of metrics helps us with the overall operation of the system. We can find out whether we are meeting our guarantees, how traffic volumes are changing, how to size the system going forward, and help diagnose system-level issues like bad hardware or noisy neighbors. We also can watch for unexpected model performance or input data changes. It might be important to be able to inject tags into this kind of metrics so that we can drill into these aggregates to measure performance for special customers, queries that came from particular sources, or where we might have some other hint that there is a class of requests to pay special attention to. We talk more about analyzing aggregated metrics in Chapter 7.

The second kind of metrics helps us drill into the specific timing details of the system. This can help us debug issues in rendezvous policies and find hot-spots in certain kinds of queries. These trace-based measurements are particularly powerful if we can trigger the monitoring on a request-by-request basis. That allows us to run low

volume tests on problematic requests in a production setting without incurring the data volume costs of recording traces on the entire production volume. For systems with low to moderate volumes of requests, the overhead of latency tracing doesn't matter, and it can be left on for all requests. Chapter 7 provides more information about latency tracing.

The third kind of metrics is good for measuring which models are more accurate than others. In this section, we talk about how the rendezvous helps gather the data for that kind of comparison, but check out Chapter 5 for more information on how models can actually be compared to each other.

In recording metrics, there are two main options. One is to insert data inline into messages as they traverse the system. The virtue here is that we can tell everything that has happened to every request. This is great for detecting problems with any particular message request.

The alternative of putting all of the metrics into a side-channel has almost exactly the opposite virtues and vices relative to inline metrics. Tracking down information on a single request requires a join or search, but aggregation is easier and the amount of additional data in the request themselves is very small. Metric storage and request archives can be managed independently. Security for metrics is separated from security for requests.

For machine learning applications, it is often a blend of both options that is best. There are a few metrics related to processing time, to which model's results were used, and so on, that can be very useful to the process that issued the request in the first place; those should be put inline in the response. Other metrics (probably larger by far) are of more interest to the operations specialists in the DataOps team, and thus separate storage is probably warranted for those metrics. It is still important to carry a key to join the different metrics together.

## Anomaly Detection

On seriously important production systems, you also should be running some form of automated analytics on your logs. It can help dramatically to do some anomaly detection, particularly on latency for each model step. We described how to automate much of the anomaly detection process in our book *Practical Machine Learning:*

*A New Look at Anomaly Detection* (O'Reilly, 2014). Those methods are very well suited to the components of a rendezvous architecture. The basic idea is that there are patterns in the metrics that you collect that can be automatically detected and give you an alert when something gets seriously out of whack. The model latencies, for instance, should be nearly constant. The number of requests handled per second can be predicted based on request rates over the last few weeks at a similar time of day.

## Rule-Based Models

Nothing says that we have to use machine learning to create models, even if the title of this book seems to imply that you do.

In fact, it can be quite useful to build some models by hand using a rule-based system. Rule-based models can be useful whenever there are hard-and-fast requirements (often of a regulatory nature) that require exact compliance. For example, if you have a requirement that a customer can't be called more often than once every 90 days, rules can be a good option. On the other hand, rules are typically a very bad way to detect subtle relationships in the data; thus, most fraud detection models are built using machine learning. It's fairly common to combine these types of systems in order to get some of the best out of both types. For instance, you can use rules to generate features for the machine learning system to use. This can make learning much easier. You could also use rules to post-process the output of a machine learning system into specific actions to be taken.

That said, all of the operational aspects of the rendezvous model apply just as well if you are using rule-based models or machine learning models, or even if you are using composites. The core ideas of injecting external state early, recording inputs using a decoy, comparing to a canary model and using streams to connect components to a rendezvous server still apply.

## Using Pre-Lined Containers

All models in a rendezvous server should be containerized to make management easier. It doesn't much matter which container or orchestration framework you pick, but containerization pays big benefits. The most important of these is that containers provide a

consistent runtime environment for models as well as a way to version control configuration. This means that data scientists can have a prototyping environment that is functionally identical to the production environment. Containers are also useful because the operations specialists in a DataOps team can provide scaffolding in a standardized container to requests from input streams and to put results into output streams. This scaffolding can also handle all metrics reporting, which, frankly, isn't likely to be a priority for the data science part of the team—at least not until they need to debug a nasty problem involving interactions between supposedly independent systems.

Having container specifications prebuilt can make it enormously easier to support different machine learning toolsets, as well. More and more, the trend by DataOps teams is to use "one of each" as a modeling technology strategy. With containers already set up for each class of model, having diversity in modeling technologies becomes an advantage rather than a challenge.

Containers also facilitate the use of source code version tracking to record changes in configuration and environment of models. Losing track of configuration and suffering from surprising environmental changes is a very common cause of mysterious problems. Assumptions about configuration and environments can make debugging massively more difficult. Falsifying such assumptions early by taking control of these issues is key to successfully replicating (or, better, avoiding) production problems.

CHAPTER 4

# Managing Model Development

Model development is the aspect that is most unchanged by the introduction of rendezvous systems, containers, and a DataOps-style of development, but the rendezvous style does bring some important changes.

One of the biggest differences is that in a DataOps team, model development goes on cheek-by-jowl with software development, and operations with much less separation between data scientists and others. What that means is that data scientists must take on some tasks relative to packaging and testing models that are a bit different from what they may be used to. The good news is that doing this makes deployment and management of the model smoother, so the data scientists are distracted less often by deployment problems.

## Investing in Improvements

Over time, systems that use machine learning heavily can build up large quantities of hidden technical debt. This debt takes many forms, including data coupling between models, dead features, redundant inputs, hidden dependencies, and more. Most important, this debt is different from the sort of technical debt you find in normal software, so the software and ops specialists in a DataOps team won't necessarily see it and data scientists, who are typically used to working in a cloistered and sterilized environment won't recognize it either because it is an emergent feature of real-world deployments.

A variety of straightforward things can help with this debt. For instance, you should schedule regular efforts to do leave-one-out analysis of all input variables in your model. Features that don't add performance are candidates for deletion. If you have highly collinear features, it is important to make time to decide if the variables really both need to be referenced by the model. Many times, one of the variables has a plausible causal link with the desired output, while the other may be a spurious and temporary correlation.

### Build One to Throw Away

When you build feature extraction pipelines, use caution. The iterative and somewhat nondirected way that feature extraction is developed means that your feature extraction code will eventually become a mess. Don't expect deep learning to fix all your sins, either. Deep learning can take massive amounts of data to learn to recognize features. However, if you combine domain-specific features with a deep learning model, you can likely outperform a raw deep learning system alone.

Given that your feature extraction code is going to become messy, you might as well just plan to throw it away at some point and develop a cleaned-up version. Of course, retaining live data using a decoy model is a great way to build a test suite for the replacement.

### Annotate and Document

It may sound like a parent telling you to take your vitamins and wear galoshes, but annotating and documenting both current and potential model features and documenting signal sources and consumers pays huge dividends, far in excess of the return for doing the same sort of thing for normal code. It is often really difficult to get a team to do this annotation well, but since your mom and dad want you to do it, you might as well get started.

## Gift Wrap Your Models

One of the most difficult things to debug in a production model is when it produces slightly, but importantly, different results due to a difference in the development and production environments. Happily, this problem should largely be a thing of the past thanks to container technology. It is now possible to build a container that freezes the environment for your models—an environment you certify as

correct in development—and pass through to production, completely unchanged. Moreover, it is possible to equip the base container with scaffolding that fetches all input data from an input stream and returns results to the output stream, while maintaining all pertinent metrics. All that is left is to select the right container type and insert a model.

The result is a packaging process that doesn't overly distract from data science sorts of tasks and thus is more likely to be done reliably by data scientists or engineers. The result is also a package that can immutably implement a model to be deployed in any setting.

For stateless models, it really is that simple. For models with state persisted outside the model (external state) it's also that simple. For models with internal state, you have to be able to configure the container at runtime with the location of a snapshot for the internal state so you can roll forward the state from their. If the internal state is based on a relatively small window, no snapshot is necessary and you can simply arrange to process enough history on startup to recreate the state.

## Other Considerations

During development, the `raw` and `input` streams can be replicated into a development environment, as shown in Figure 4-1.

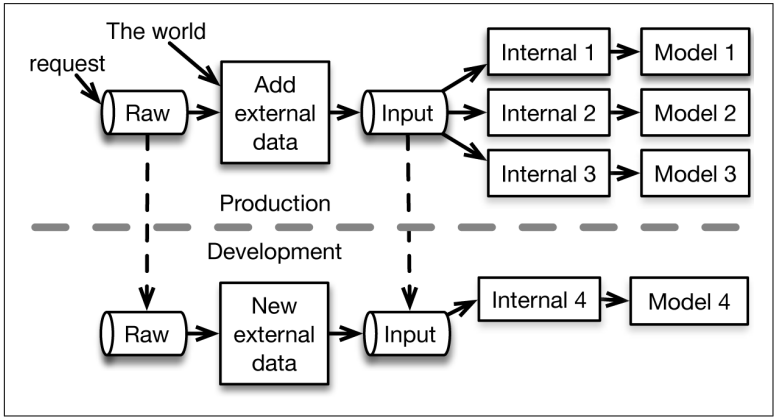



*Figure 4-1. Stream replication can be used to develop models in a completely realistic environment without risking damage to the production environment.*

If the external data available in the production environment is suitable for the new model, it is easiest to replicate just the `input` stream to avoid replicating the external data injector. On the other hand, if the new model requires a change to the external data injector, you should replicate the `raw` stream, instead. In either case, a new model, possibly with new internal state management can be developed entirely in a development environment but with real data presented in real time. This ability to faithfully replicate the production environment can dramatically decrease the likelihood of models being pulled from production due to simple configuration or runtime errors.

CHAPTER 5

# Machine Learning Model Evaluation

This chapter is about model evaluation in the context of *running production models*. You will learn practical tips to deal with model evaluation as an ongoing process of improvement—strengthening your models against real data, and solving real problems that change over time.

We won't be covering (offline) model evaluation in the traditional sense; to learn more about that topic, we recommend you check out Alice Zheng's free eBook *Evaluating Machine Learning Models: A Beginner's Guide to Key Concepts and Pitfalls* (O'Reilly).

When it comes to improving models *in production*, it becomes critical to compare the models against what they did yesterday, or last month—at some previous point in time. You may also want to compare your models against a known stable canary model, or against a known best model. So, let's dig in.

## Why Compare Instead of Evaluate Offline?

In a working production system, there will be many models already in production or in pre- or post-production. All of these models will be generating scores against live data. Some of the models will be the best ones known for a particular problem. Moreover, if that production system is based on a rendezvous-style architecture, it will be very easy and safe to deploy new models so that they score live data

in real-time. In fact, with a rendezvous architecture it is probably easier to deploy a model into a production setting than it is to gather training data and do offline evaluation.

With a rendezvous architecture, it is also be possible to replicate the input stream to a development machine without visibly affecting the production system. That lets you deploy models against real data with even lower risk.

The value in this is that you can take your new model's output and compare that output, request by request, against whatever benchmark system you want to look at. Because of the way that the rendezvous architecture is built, you know that both models will have access to exactly the same input variables and exactly the same external state. If you replicate the input stream to a development environment, you also can run your new model on the live data and pass data down to replicas of all downstream models, thus allowing you to quantify differences in results all the way down the dependency chain.

The first and biggest question to be answered by these comparisons is, "What is the business risk of deploying this model?" Very commonly, the new model is just a refinement of an older model still in production, so it is very likely that almost all of the requests will be almost identical results. To the extent that the results are identical, you know that the performance is trivially the same, as well. This means that you already have a really solid bound on the worst-case risk of rolling out this new model. If you do a soft roll out and give the new model only 10 percent of the bandwidth, the worst-case risk is smaller by a factor of 10.

You can refine that estimate of expected risk/benefit of the new model by examining a sample of the differing records that is stratified by the score difference. Sometimes, you can get ground truth data for these records by simply waiting a short time, but often, finding the true outcome can take longer than you have. In such cases, you might need to use a surrogate indicator. For example, if you are estimating likelihood of response to emails, response rate at two hours is likely a very accurate surrogate for the true value of response rate after 30 days.

# The Importance of Quantiles

Where a new model has a very different implementation from existing production models, the new model may not have anything like the same output range or calibration. For models that produce numerical scores, comparisons are still possible against the production models if you reduce all scores for the new and standard models to quantiles relative to an historical set of scores. You can then do stratified sampling on the difference in quantile instead of the potentially meaningless difference in scores.

In fact, reducing scores to quantiles relative to the distribution of recent scores is enormously helpful in comparing models. This is particularly handy for models with simple numerical scores as outputs because it allows us to use what is known as a *q-q* plot to compare the scores in terms of percentiles, which gets rid of problems to do with highly variable score calibration. Figure 5-1 shows the result of such a comparison on synthetic data both in terms of raw score and *q-q* diagram.

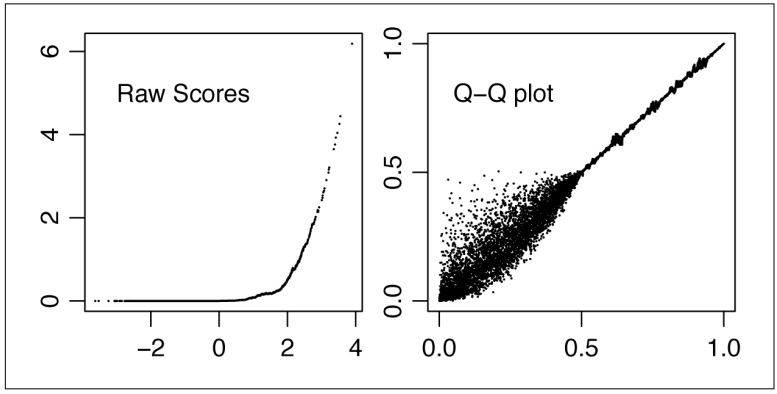



*Figure 5-1. Using quantiles allows better visualization of how two scores are related. Two score distributions are compared as raw scores (left) or quantiles (right).*

With the raw scores, the drastic differences in score calibration prevent any understanding of the correlation of scores below about 2 on the x-axis and obscure the quality of correlation even where scales are comparable. The *q-q* diagram on the right, however, shows that the same two scores are an extremely good match for the top 10 percent of all scores and a very good match for the top 50 percent.

If this were a fraud model, it is likely that we would be able to declare the risk of using the new model as negligible because fraud models typically care only about the largest scores and the *q-q* plot shows that for the top 5 percent, these models are functionally identical, if not in score calibration.

## Quantile Sketching with t-Digest

If we have only a few thousand queries, computing quantiles directly in R or Python is a nonissue. Even up to a million scores or so, it isn't all that big a deal. But what we need to do is much more nuanced. We need to be able to pick subsets of scores selected by arbitrary qualifiers over pretty much arbitrary times and compare the resulting distribution to other distributions. Without some sort of distribution sketch, we would need to read every score of interest and sort all of them to estimate quantiles. To avoid that, we suggest using the *t*-digest.

**NOTE**

The *t*-digest is an algorithm that computes a compact and efficient sketch of a distribution of numbers, especially for large-scale data.

You can combine multiple *t-digest* sketches easily. This means that we can store sketches for score distributions for each, say, 10-minute period for each of the unique combinations of request qualifiers such as time of day, or source geo-location, or client type. Then, we can retrieve and combine sketches for any time period and any combination of qualifiers to get a sketch of just the data we want. This capability lets us compare distributions for different models, for different times in terms of raw values, or in terms of quantiles. Furthermore, we can retrieve the *t*-digest for any period and condition we want and then convert a set of scores into quantiles by using that sketch.

Many of the comparisons described here would be very expensive to do with large numbers of measurements, but they can be completed in fractions of a second by storing *t*-digests in a database.

## The Rubber Hits the Road

After we are pretty sure that the downside risk of a model is small and that the operational footprint is acceptable, the best way to evaluate it is to put it into service by using its results a fraction of the time, possibly selected based on some user identity hash so that one user sees a consistent experience. As we begin to pay attention to the new candidate, we need to watch whatever fast surrogate for performance that we have for the effects of the new model.

Offline evaluation of models is a fine thing for qualifying candidates to be tried in production, but you should keep in mind the limits to offline evaluation. The fact is, machine learning systems can be sensitive to data coupling. When these systems make real-world decisions and then take training data from the real world, there is a risk that they couple to themselves, making offline evaluation have only marginal value. We have seen a number of times where adding noise to a model's output (which inevitably degrades its current performance) can broaden the training data that the system acquires and thus improve performance in the future. This can have a bigger effect than almost any algorithmic choice and can easily make a model that scores worse offline perform much better in the real world. Conversely, a model that scores better offline can couple to other systems and make them perform more poorly.

As such, rolling a model out to get some real-world data is really the only way to truly determine if a model is better than the others.

# CHAPTER 6
# Models in Production

> *...developing and deploying ML systems is relatively fast and cheap, but maintaining them over time is difficult and expensive*
>
> —D. Sculley et al., "Hidden Technical Debt in Machine Learning Systems", NIPS 2015

Modern machine learning systems are making it much easier to build a basic decisioning system. That ease, however, is a bit deceptive. Building and deploying a first decisioning system tends to go very well, and the results can be quite impressive for the right applications. Adding another system typically goes about as well. Before long, however, strange interactions can begin to appear in ways that should be impossible from a software engineering point of view. Changing one part of the system affects some other part of the system, even though tests in isolation might suggest that this is impossible.

The problem is that systems based on machine learning can have some very subtle properties that are very different from more traditional software systems. Partly, this difference comes from the fact that the outputs of machine learning systems have much more complex behaviors than typical software components. It also comes about, in part at least, because of the probabilistic nature of the judgments such systems are called upon to make.

This complexity and subtlety makes the management of such systems trickier than the management of traditional well-modularized software systems based on microservices. Complex machine learning systems can grow to exhibit pathological "change anything, change everything" behaviors—even though they superficially

appear to be well-designed microservices with high degrees of isolation.

There are no silver bullet answers to these problems. Available solutions are pragmatic and have to do with making basic operations *easier* so that you have more time to think about what is really happening. The rendezvous architecture is intended to make the day-to-day operations of deploying and retiring machine learning models easier and more consistent. Getting rid of mundane logistical problems by controlling those processes is a big deal because it can give you the time you need to think about and understand your systems. That is a prime goal of the rendezvous architecture.

The architecture is also designed to provide an enormous amount of information about the inputs to those models and what exactly those models are doing. Multiple models can be run at the same time for comparison and cross-checking as well as helping to meet latency and throughput guarantees.

**NOTE**

A common first impression by people unfamiliar with running models in production is that the rendezvous architecture goes to extremes in measuring and comparing results. Engineers experienced in fielding production machine learning models often ask if there are ways to get even more information about what is happening.

## Life with a Rendezvous System

There are a few basic procedures that cover most of what must happen operationally with a rendezvous architecture. These include bringing new models into preproduction (also known as staging), rolling models to production and retiring models. There are also a few critical processes that involve with doing maintenance on the rendezvous system itself.

It should be remembered that no machine learning system except the most trivial is ever really finished. There will always be new ideas to try, so model performance can be subject to decay over time. It's important to monitor ongoing performance of models to determine when to bring out a new edition.

## Model Life Cycle

The life cycle of a model consists of five main phases: development, preproduction, production, post-production, and retirement. The goal of development is to deliver a container-based model that meets the data science goals for the service.

> **NOTE**
>
> The key criterion for exiting the development phase is that the model should have been run on archived input data and behaved well in terms of accuracy and runtime characteristics such as memory footprint and latency distribution.

Models enter preproduction as a container description that specifies the environment for the model, including all scaffolding for integrating the container into the rendezvous architecture. In addition to specifying the environment, the container description should refer to a version-controlled form of the model itself. Putting a model into preproduction consists of inspecting and rebuilding the model container and starting it on production hardware. On startup, the model is connected to streams with production inputs and production outputs. Next, you scale the container with enough replicas to give the required performance. You also need to bring any internal state in the model up to date by supplying a snapshot of the internal state and replaying transactions after the snapshot. If the internal state depends only on a relatively short window, you can just replay transactions on top of an initially zero state.

At this point, the model is running in production except for the fact that the rendezvous server is ignoring its results. As far as the model is concerned, everything is exactly the same way it would be in production.

During the time a model is fully in preproduction and all internal state is live, you should monitor it for continued good behavior in terms of latency, and you can compare its output to the canary model and current production models. If the new model exhibits operational instability or has significantly lower accuracy than current champion and contender models, you may decide to send the model directly to retirement by stopping any containers running the model. Many models will go directly to retirement at this point, hopefully because your current champion is really difficult to beat.

After the preproduction model has enough runtime that you feel confident that it is stable and you have seen it give competitive accuracy relative to the current champion, you can soft-roll it into production by giving the rendezvous server new schedules that use the new model for progressively more and more production traffic, and use the previous champion for less and less traffic. You should keep the old champion running during this time and beyond in case you decide to roll back. The transition to the new champion should be slow enough for you to keep an eye out for any accidental data dependencies, but the transition should also be sudden enough to leave a sharp and easily detectable signature in the metrics for the rest of the system in case there is an adverse effect of the roll. You want to avoid a case in which rolling to the new model has a bad effect on the overall system, but happens so slowly that nobody notices the effect right away.

At this point, the model is in production, but nothing has changed as far as the model is concerned. It is still evaluating inputs and pumping out results exactly as it did in preproduction. The only difference is that the output of the model can now affect the rest of the world. For some models, their output inherently selects the training data for all the models and so rolling a new model into production can have a big effect data-wise. Especially with recommendation engines, this leads to a tension between giving the best recommendations that we know how to give versus giving recommendations that include some speculative material that we don't know as much about. Presenting the best results is good, but so is getting a broader set of training data. We have discussed this effect in our book *Practical Machine Learning: Innovations in Recommendation*. If you suspect that something like this is happening, you may want to randomly select between the decisions of several models to see if there is a perceptible effect on output quality.

Eventually, any champion model will be unseated by some new contender. This might be because the contender is more efficient or faster or more accurate. It could also happen because model performance commonly degrades over time as the world changes or, in the case of fraud models, due to innovation by adversaries.

Moving a model into post-production is inherent in the process of bringing the new champion into production and is done by giving the rendezvous server new schedules that ignore the old champion. In the post-production phase, the model will still handle all the pro-

duction traffic but its outputs will be ignored by the rendezvous server. Post-production models should still be monitored to verify that they are safe to keep around as a fallback. Typically, you keep the last champion to enter post-production in the rendezvous schedule as a fallback in case the new champion fails to produce a result in a timely manner.

Eventually, a former champion is completely removed from the rendezvous schedule. After you are confident that you won't need to use output of the old champion any more, you can retire it. A few models should be kept in post-production phase for a long time, serving as canary models. After a model is fully retired, you can stop all of the containers running it.

## Upgrading the Rendezvous System

Eventually, the services making up the rendezvous architecture itself will need to be updated. This is likely to be far less common than rolling new models, but it will happen. Happily, the entire architecture is designed to allow zero downtime upgrades of the system components.

Here are the key steps in upgrading the rendezvous system components:

1. Start any new components. This could be the external state systems or the rendezvous server. All new components should start in a quiescent state and should ignore all stream inputs.
2. Inject a state snapshot token into the input of the system. There should be one token per partition. As this snapshot token passes through the system, all external state components will snapshot their state to a location specified in the token and pass the snapshot token into their output stream.
3. The new versions of the external state maintenance systems will look for the snapshots to complete and will begin updating their internal state from the offsets specified in the snapshots. After the new external state systems are up to date, they will emit snapshot-complete tokens.
4. When all of the new external state systems have emitted their snapshot-complete tokens, transition tokens are injected into the input of the system, again one token per partition.

5. As the external state systems receive transition tokens, the old versions will stop augmenting input records, and the new versions will start. This will provide a zero-delay hand-off to the new systems. The transition tokens will be passed to the models, which should simply pass the tokens through.
6. When the old rendezvous server sees a transition token on the system input, it will stop creating new mailboxes for incoming transactions but will continue to watch for results that might apply to existing mailboxes.
7. When the new rendezvous server sees a transition token on the system input, it will start creating mailboxes for all transactions after the token and watching for results for those requests.

When this sequence of steps is complete, the system will have upgraded itself to new versions with no latency bumps or loss of state. You can keep old versions of the processes running until you are sure that you don't need to roll back to the previous version. At that time, you can stop the old versions.

## Beware of Hidden Dependencies

If you build a simple system in which you use a rendezvous architecture to run and monitor a single kind of model, things are likely to work just fine with no big surprises. Unfortunately, life never stays that simple. Before long, you will have new models that do other things, so you will light up some more rendezvous systems to maintain them. Next, you'll have models that compute some signal to be used as external state for other models. Then, you will have somebody who starts using a previously stable model as external state. Suddenly, one day, you will have a complicated system with models depending on other models.

When that day comes (and it probably will) you are likely to get some surprises due to model interactions. Even though the models are run completely independently in separate rendezvous systems, the fact that one model consumes the results of another can cause some very complex effects. Models can become coupled in very subtle ways purely by the data passing between them. The same thing could conceivably happen with ordinary microservices, but the data that passes between microservices doesn't normally carry as much

nuance as the data that passes between machine learning systems, thus limiting the amount of data coupling.

## A Simplified Example of Data Coupling

Let's look at a simplified example of a fairly common real-world scenario. In this example, we want to find fraud and let's just consider how much of the total fraud that we find and not consider the problem of false alarms where we mark a clean transaction as fraud. Now, let's suppose that there are actually two different kinds of transactions, red and blue each equally common. We can't actually see the difference, but we know that there are still two kinds. For simplicity, let's suppose that fraud is equally common in both colors.

We have two models that we want to use in a cascade such as that shown in Figure 6-1.

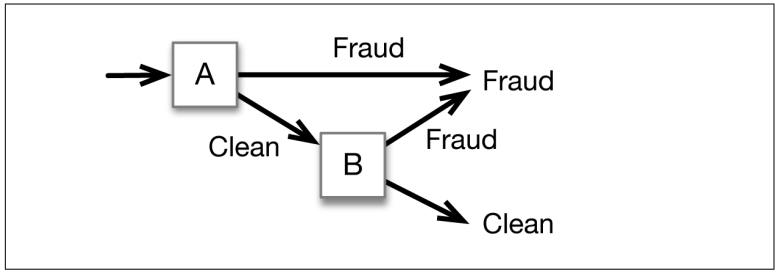



*Figure 6-1. Two models cascaded to find as much fraud as possible.*

Now, let's assume that A and B have complementary performance when looking for fraud. Model A marks 80 percent of red frauds as fraud but never finds any blue frauds. This means that A by itself can find 40 percent of all fraud because only half of all frauds are red. Model B is just the opposite. It marks all red transactions as clean and finds 80 percent of the blue frauds. Working together in our cascaded model, A and B can find 80 percent of all the fraud.

Now suppose we "improve" A so that it finds 70 percent of fraud instead of just 40 percent. This sounds much better! Under the covers, let's suppose that after the improvement, A finds 100 percent of all blue frauds, and 40 percent of all red frauds.

If we use A in combination with B as we did earlier, A will find 100 percent of the blue frauds and 40 percent of the red frauds. B, working on the transactions A said were clean, will have no blue frauds to find, and it never finds any red frauds anyway, so B won't find any

frauds at all. The final result will be that with our new and improved version of A, we now find 70 percent of all the fraud instead of the 80 percent we found with the original "lousy" version of A.

How is it that nearly doubling the raw performance of A made things worse? The problem is that the "improvement" in A came at the expense of making A's strength correlate with the strength of B. Originally, A and B complemented each other, and each compensated for the substantial weakness of the other. After the change, the strength of A completely overlaps and dominates the strength of B, but A has lost much of the strength it original had. The net is that B has nothing to offer and the overall result is worse than before.

In real life, unfortunately, this can easily happen. It seems easy to spot this kind of thing, but in practice we can't tell the red transactions from the blue transactions. All we get to see is raw performance and the new A looks unconditionally better than the old A. In such a situation, we might have a graph of frauds found over time that looks like Figure 6-2.

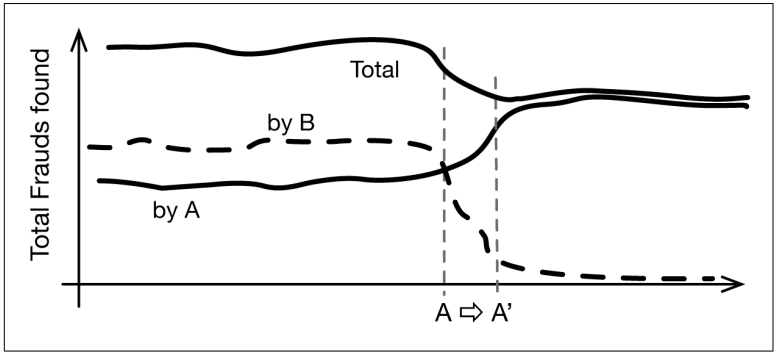



*Figure 6-2. The surprising effect of model dependencies. If improvements in model A cause performance of A to correlate with performance of model B, the cascaded performance may actually decrease even if A alone performs better.*

If we were to look at the graph without knowing that it was just A that changed, we would quite reasonably suspect that somebody had made a disastrous change to B because it was B that showed the drop in performance. Given that we know exactly when A' was brought live, we can guess that there is a causal connection between upgrading A and losing B. In practice, the team running A should know that B is downstream and should watch for changes.

Conversely, the team running B should know that A is upstream. With cascaded models like this, it is commonly the same team running A and B, but it is surprisingly common to have problems like this unawares.

The trick is to either avoid all data dependencies or to make sure you know as much as possible about any that do occur. One key way to limit surprise dependencies is to use permissions on streams to limit the consumers to systems that you know about instead of letting there be accidental dependencies propagating through your systems as other teams set up models.

Of course, it isn't feasible to completely eliminate all data dependencies because processes that are completely outside of your control might accidentally couple via data streams that you import from the outside world. For example, a competitor's marketing program can change the characteristics of the leads you get from your marketing program resulting in a coupling from their system to yours. Likewise, a fraudster's invention of a new attack could change the nature of frauds that you are seeing. You will see these effects and you will need to figure out how to deal with the results.

## Monitoring

Monitoring is a key part of running models in production. You need to be continually looking at your models, both from a data science perspective and from an operational perspective. In terms of data science, you need to monitor inputs and outputs looking for anything that steps outside of established norms for the model. From an operational perspective, you need be looking at latencies, memory size, and CPU usage for all containers running the models.

Chapter 7 presents more details on how to do this monitoring.

# CHAPTER 7
# Meta Analytics

*I know who I WAS when I got up this morning, but I think I must have been changed several times since then.*

—*Alice in Wonderland*, by Lewis Carroll

Just as we need techniques to determine whether the data science that we used to create models was soundly applied to produce accurate models, we need additional metrics and analytics to determine whether the models are functioning as intended. The question of whether the models are working breaks down into whether the hardware is working correctly, whether the models are running, and whether the data being fed into the models is as expected. We need metrics and analytics techniques for all of these. We also need to be able to synthesize all of this information into simple alerts that do not waste our time (more than necessary).

The rendezvous architecture is designed to throw off all kinds of diagnostic information about how the models in the system are working. Making sense of all of that information can be difficult, and there are some simple tricks of the trade that are worth knowing. One major simplification is that because we are excluding the data science question of whether the models are actually producing accurate results, we can simplify our problem a bit by assuming that the models were working correctly to begin with—or at least as correctly as they could be expected to work. This means that our problem reduces to the problem of determining whether the models are working like they used to do. That is a considerably easier problem.

Note that in doing this, we are analyzing the behavior of the analytical engines themselves. Because of that flavor of meta analysis, we call our efforts "meta analytics."

Generally, meta analytics can be divided into data monitoring and operational monitoring. *Data monitoring* has to do with looking at the input data and output results of models to see how things are going. *Operational monitoring* typically ignores the content and looks only at things like latencies and request frequency. Typically, data monitoring appeals and makes sense to data scientists and data engineers, whereas operational monitoring makes more sense to operations specialists and software engineers. Regardless, it is important that the entire team takes meta analytics seriously and looks at all the kinds of metrics to get a holistic view of what is working well and what is not.

# Basic Tools

There are a few basic techniques that are very useful for both data monitoring and operational monitoring for meta analytics, many of which are surprisingly little known. All of these methods have as a goal the comparison of an estimate of what should be (we call that "normal") with what is happening now. In model meta analytics, two key tools are an ability to look for changes in event rates and to estimate the distribution of a value. We can combine these tools in many ways with surprising results.

## Event Rate Change Detection

Events in time are a core concept in meta analytics. Simply realizing when an intermittent event has changed rate unexpectedly is a very useful kind of meta-analytical measure. For instance, you can look at requests coming to a model and mark times when the rate of requests has changed dramatically. If the rate of requests has dropped precipitously, that often indicates an upstream problem. If the rate of outgoing requests for external state changes in a fashion incompatible with the number of incoming requests, that is also a problem.

The good news is that detecting changes in event rates is pretty simple. The best way to detect these changes is typically to look at the times since the last or *n*th last event happened. Let's assume that you have a predicted rate (call this $\lambda$) and have seen the times of all events up to now (call these $t_1 ... t_i$). If you want to detect complete stoppage in the events, you can just use the time since the last event scaled by the predicted rate $\lambda (t - t_i)$, where $t$ is the current time. If

your predicted rate is a good estimate, this signal will be as good as you can get in terms of trading-off false positives and false negatives versus detection time.

If you are looking for an alarm when the rate goes up or down, you can use the *n*th event time difference or $\lambda(t - t_{i-n+1})$ as your measure. If $n$ is small, you will be able to detect decreases in rate. With a larger $n$ you can also detect increases in rate. Detecting small changes in rate requires a large value of $n$. Figure 7-1 shows how this can work.

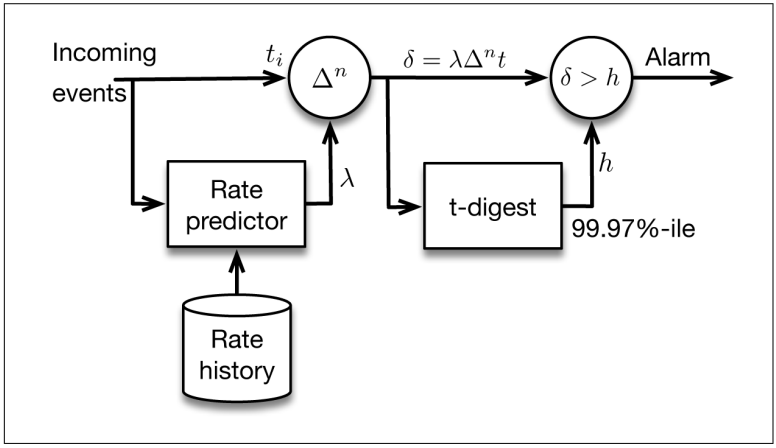



*Figure 7-1. Detecting shifts in rate is best done using n-th order difference in event time. The t-digest can help pick a threshold.*

You may have noticed the pattern that you can't see changes that are (too) small (too) quickly without paying a price in errors. By setting your thresholds, you can trade off detecting ghost changes or missing real changes, but you can't fundamentally have everything you want. This is a kind of Heisenberg principle that you can't get around with discrete events.

Similarly, all of the event time methods talked about here require an estimate of the current rate $\lambda$. In some cases, this estimate can be trivial. For instance, if each model evaluation requires a few database lookups, the request rate multiplied by an empirically determined constant is a great estimator of the rate for database lookups. In addition, the rate of website purchases should predict the rate of frauds detected. These trivial cross-checks between inputs and outputs sound silly but are actually very useful as monitoring signals.

Aside from such trivial rate predictions, more interesting rate predictions based on seasonality patterns that extend over days and weeks can be made by using computing hourly counts and building a model for the current hour's count based on counts for previous hours over the last week or so. Typically, it is easier to use the log of these counts than the counts themselves, but the principle is basically the same. Using a rate predictor of this sort, we can often predict the number of requests that should be received by a particular model within 10 to 20 percent relative error.

## t-Digest for One-Dimensional Score Binning

If we look into the output of a model, we often see a score (sometimes many scores). We can get some useful insights if we ask ourselves if the scores we are producing now look like the scores we have produced previously. For instance, the TensorChicken project produced a list of all potential things that it could see such as a chicken or an empty nest along with scores (possibly probabilities) for each possible object. The scores for each kind of thing separately form a distribution that should be roughly constant over time if all is going well. That is, the model should see roughly the same number of chickens or blue jays or open doors over time. This gives us a score distribution for each possible identification.

As an example, Figure 7-2 shows the TensorChicken output scores over time for "Buff Orpington." There is a clearly a huge change in the score distribution part way across the graph at about sample 120. What happened is that the model was updated in response to somebody noticing that the model had been trained incorrectly so that what it thought were Buff Orpington chickens were actually Plymouth Rocks. At about sample 120, the new model was put into service and the score for Orpingtons went permanently to zero.

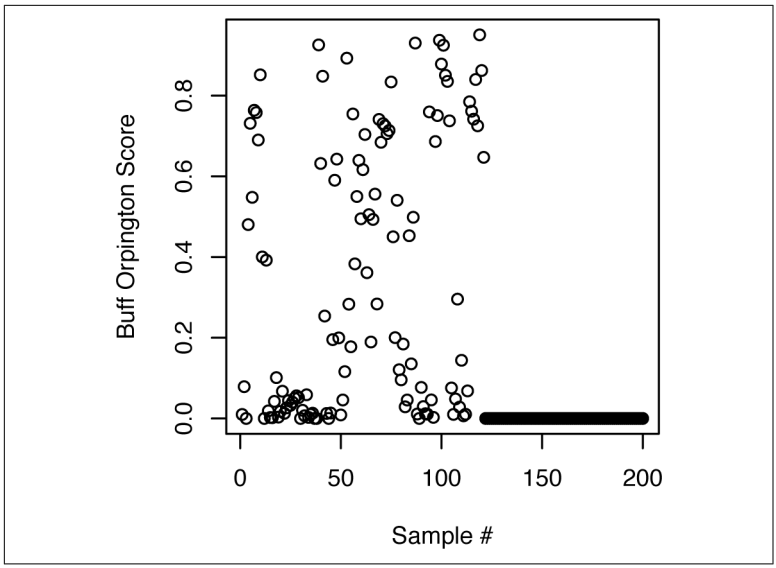



*Figure 7-2. The recognition scores for Buff Orpington chickens dropped dramatically at sample 120 when the model was updated to correct an error in labeling training data.*

From just the data presented here, it is absolutely clear that a change happened, but it isn't clear whether the world changed (i.e., Buff Orpington chickens disappeared) or whether the model was changed.

This is exactly the sort of event that one-dimensional distribution testing on output scores can detect and highlight. The distinction between the two options is something that having a canary model can help us distinguish.

A good way to highlight changes like this is to use a histogramming algorithm to bin the scores by score range. The appearance of a score in a particular bin is an event in time whose rate of occurrence can be analyzed using the rate detection methods described earlier in this chapter. If we were to use bins for each step of 0.1 from 0 to 1 in score, we would see nonzero event counts for all of the bins up to sample 120. From then on, all bins except for the 0.0–0.1 bin would get zero events.

The bins you choose can sometimes be picked based on your domain knowledge, but it is often much handier to use bins that are picked automatically. The *t*-digest algorithm does exactly this and

does it in such a way that the structure of the distribution near the extremes is very well preserved.

## K-Means Binning

Taking the issue of the model change in TensorChicken again, we can see that not only did the distribution of one of the scores change, the relationship between output scores changed. Figure 7-3 shows this.

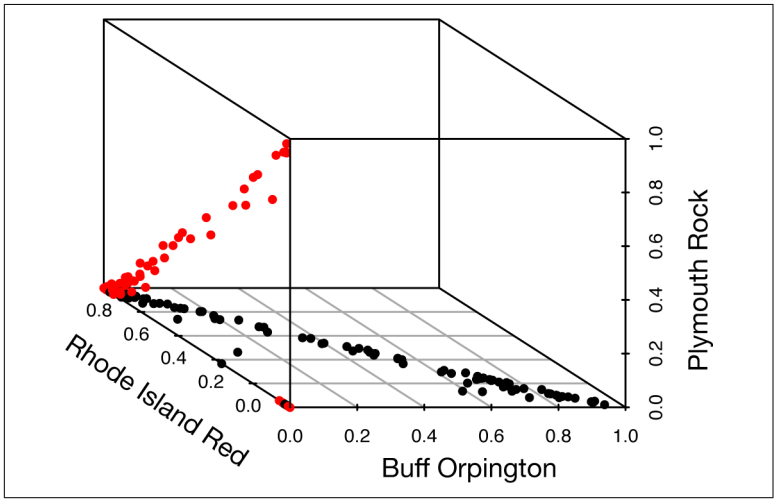



*Figure 7-3. The scores before the model change (black) were highly correlated but after the model change (red), the correlation changed dramatically. K-means clustering can help detect changes in distribution like this.*

A very effective way to measure the change in the relationship between scores like this is to cluster the historical data. In this figure, the old data are the black dots. As each new score is received, the distance to the nearest cluster is a one-dimensional indicator of how well the new score fits the historical record. It is clear from the figure that the red points would be nowhere near the clusters found using the black data points and the distance to nearest cluster would dramatically increase when the red scores began appearing. The rate for different clusters also would dramatically change, which can be detected as described earlier for event rates.

## Aggregated Metrics

Aggregating important metrics over short periods of time is a relatively low-impact kind of data to collect and store. Values that are aggregated by summing or averaging (such as number of queries or CPU load averages) can be sampled every 10 seconds or minute and driven into a time–series database such as Open TSDB or Influx or even ElasticSearch.

Other measurements such as latencies are important to aggregate in such a way that you understand the exact distribution of values that you have seen. Simple aggregates like min, max, mean, and standard deviations do not suffice.

The good news is that there are data structures like a `FloatHistogram` (available in the *t*-digest library) that do exactly what you need. The bad news is that commonly used visualization systems such as Grafana don't handle distributions well at all. The problem is that understanding latency distributions isn't as simple as just plotting a histogram. For instance, Figure 7-4 shows a histogram of latencies in which about 1 percent of the results have three times higher values than the rest of the results. Because both axes are linear, and because the bad values are spread across a wider range than the good values, it is nearly impossible to determine that something is going wrong.

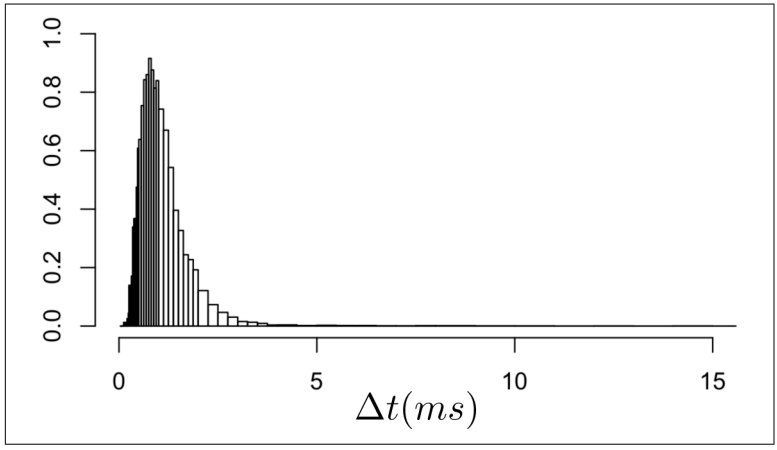



*Figure 7-4. The float histogram uses variable width bins. Here, we have synthetic data in which one percent of the data has high latency (horizontal axis). A linear scale for frequency (vertical axis) makes it hard to see the high latency samples.*

These problems can be highlighted by switching to nonlinear axes, and the nonuniform bins in the `FloatHistogram` also help. Figure 7-5 shows the same data with logarithmic vertical axis.

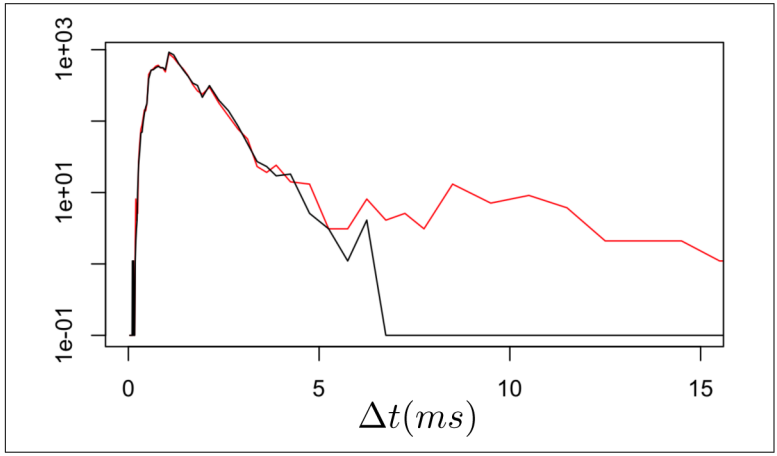



*Figure 7-5. With a logarithmic vertical axis, the anomalous latencies are clearly visible. The black line shows data without slow queries while the red line shows data with anomalous latencies.*

With the logarithmic axis, the small cluster of slow results becomes very obvious and the prevalence of the problem is easy to estimate.

Event rate detectors on the bins of the `FloatHistogram` could also be used to detect this change automatically, as opposed to visualizing the difference with a graph.

## Latency Traces

The latency distributions shown in the previous figures don't provide the specific timing information that we might need to debug certain issues. For instance, is the result selection policy in the rendezvous actually working the way that we think it is?

To answer this kind of question, we need to use a trace-based metrics system in the style of Google's Dapper (Zipkin and HTrace are open source replications of this library). The idea here is that the overall life cycle of a single request is broken down to show exactly what is happening in different phases to get something roughly like what is shown in Figure 7-6.

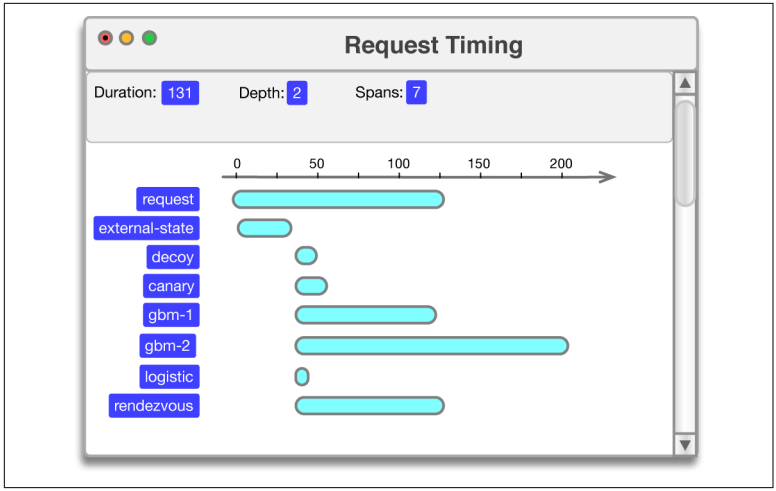



*Figure 7-6. A latency breakdown of a single request to a rendezvous architecture shows details of overlapping model evaluation.*

Trace-based visualization can spark really good insight into the operation of a single request. For instance, the visualization in Figure 7-6 shows how one model continues running even after a response is returned. That might not be good if computational resources are tight, but it also may provide valuable information to let the model run to completion as the gbm-2 model is tuned to make a good trade-off between accuracy and performance. The visualization also shows just how much faster the logistic model is. Such a model is often used as a baseline for comparison or as a fallback in case the primary model doesn't give a result in time. Latency traces are most useful for operational monitoring.

## Data Monitoring: Distribution of the Inputs

Now that we have seen how a few basic tools for monitoring work, we can talk about some of the ways that these tools can be applied.

Before that, however, it is good to take a bit of a philosophical pause. All of the examples so far looked at gross characteristics of the input to the model such as arrival rates, or they looked at distributional qualities of the output of the model. Why not look at the distribution of the input?

The answer is that the model outputs are, in some sense, where the important characteristics of the inputs are best exposed. By looking

at the seven-dimensional output of the TensorChicken model, for instance, we are effectively looking at the semantics of the original image space, which had hundreds of thousands of dimensions (one for each pixel and color). Moreover, because our team has gone to quite considerable trouble to build a model that makes sense for our domain and application, it stands to reason that the output will make sense if we want to look for changes in the data that matter in our domain. The enormously lower dimension of the output space, however, makes many analyses much easier. Some care still needs to be taken not to focus exclusively on the output, even of the canary model. The reason is that your models may imitate your own blind spots and thus be hiding important changes.

## Methods Specific for Operational Monitoring

Aside from monitoring the inputs and outputs of the model itself, it is also important to monitor various operational metrics, as well. Chief among these is the distribution of latencies of the model evaluations themselves. This is true even if the models are substantially outperforming any latency guarantees that have been made. The reason for this is that latency is a sensitive leading indicator for a wide variety of ills, so detecting latency changes early can give you enough time to react before customers are impacted.

There are a number of pathological problems that can manifest as latency issues that are almost invisible by other means.[1] Competition for resources such as CPU, memory bandwidth, memory hierarchy table entries and such can cause very difficult-to-diagnose performance difficulties. If you have good warning, however, often you can work around the problems by just adding resources even while continuing to diagnose them.

Latency is a bit special among other measurements that you might make in a production system in that we know a lot about how we want to analyze it. First, true latencies are never negative. Second, it is relative accuracy that we care about, not absolute accuracy and not accuracy in terms of quantiles. Moreover, 5 to 10 percent relative

1 This is a good example of a "feature" that can cause serious latency surprises: "Docker operations slowing down on AWS (this time it's not DNS)".
And this is an example of contention that is incredibly hard to see, except through latency: "Container isolation gone wrong".

accuracy is usually quite sufficient. Third, latency often displays a long-tailed distribution. Finally, for the purposes here, latencies are only of interest from roughly a millisecond to about 10 seconds. These characteristics make a `FloatHistogram` ideal for analyzing latencies.

An important issue to remember when measuring latencies is to avoid what Gil Tene calls "structured omission of results." The idea is that your system may have back pressure of some form so that when part of the system gets slow, it inherently handles fewer transactions which makes the misbehavior seem much rarer. In a classic example, if a system that normally does 1000 transactions per second in a 10-way parallel stream is completely paused for 30 seconds during a 10-minute test, there will be 10 transactions that show latencies of about 30 seconds and 60,000 transactions that show latencies of 10 milliseconds or so. It is only at the 99.95th percentile that the effect of this 30 second outage even shows up. By many measures, the system seems completely fine in spite of having been completely unavailable for five percent of the test time. A common solution for handling this is to measure latency distributions in short time periods of, say, five seconds. These windowed values are then combined after normalizing the counts in the window to a standardized value. Windows that have no counts are assigned counts that are duplicates of the next succeeding nonzero window. This method slightly over-emphasizes good performance in some systems that have slack periods, but it does highlight important pathological cases such as the 30-second hold.

## Combining Alerts

When you have a large number of monitors running against complex systems, the danger changes from the risk of not noticing problems because there are no measurements available to a risk of not noticing problems because there are too many measurements and too many false alarms competing for your attention.

In highly distributed systems, you also have a substantial freedom to ignore certain classes of problems since systems like the rendezvous architecture or production grade compute platforms can do a substantial amount of self-repair.

As such, it is useful to approach alerting with a troubleshooting time budget in mind. This is the amount of time that you plan to spend

on diagnosing backlog issues or paying down technical debt on the operational side of things. Now devote half of this time to a "false alarm budget." The idea is to set thresholds on all alarms so that you have things to look at for that much of the time, but you are not otherwise flooded with false alarms. For instance, if you are willing to spend 10 percent of your team's time fixing things, plan to spend five percent of your time (about two hours a week per person) chasing alarms. There are two virtues to this. One is that you have a solid criterion for setting thresholds (more about this in a moment). Second, you should automatically have a prioritization scheme that helps you focus in on the squeaky wheels. That's good because they may be squeaking because they need grease.

The trick then, is you need to have a single knob to turn in terms of hours per week of alarm time that reasonably prioritizes the different monitoring and alarm systems.

One way to do this is to normalize all of your monitoring signals to a uniform distribution by first estimating a medium-term distribution of values for the signal (say for a month or three) and then converting back to quantiles. Then combine all of the signals into a single composite by converting each to a log-odds scale and adding them all together. This allows extreme alerts to stand out or to have combinations of a number of slightly less urgent alerts to indicate a state of general light havoc. Converting to quantiles before transforming and adding the alert signals together has the property of calibrating all signals relative to their recent history, thus quieting wheels that squeak all the time. Another nice property of the log-odds scale is that if you use base-10 logs, the resulting value is the number of 9's in the quantile (for large values). Thus, log-odds (0.999) is very close to 3 and log-odds (0.001) is almost exactly –3. This makes it easy to set thresholds in terms of desired reliability levels.

# CHAPTER 8
# Lessons Learned

The shape of the computing world has changed dramatically in the past few years with a dramatic emergence of machine learning as a tool to do new and exciting things. Revolution is in the air. Software engineers who might once have scoffed at the idea that they would ever build sophisticated machine learning systems are now doing just that. Look at Ian with his TensorChicken system. There are lots more Ians out there who haven't yet started on that journey, but who soon will.

The question isn't whether these techniques are taking off. The question is how well prepared you will be when you need to build one of these systems.

## New Frontier

Machine learning, at scale, in practical business settings, is a new frontier, and it requires some rethinking of previously accepted methods of development, social structures, and frameworks. The emergence of the concept of DataOps—adding data science and data engineering skills to a DevOps approach—shows how team structure and communication change to meet the new frontier life. The rendezvous architecture is an example of the technical frameworks that are emerging to make it easier to manage machine learning logistics.

The old lessons and methods are still good, but they need to be updated to deal with the differences between effective machine learning applications and previous kinds of applications.

We have described a new approach that makes it easier to develop and deploy models, offers better model evaluation, and improves ability to respond.

## Where We Go from Here

Currently, machine learning systems are beginning to be able to do many cognitive tasks that humans can do, as long as those tasks are ones that humans can do at a glance and as long as sufficient training data is available.

One emerging trend is to use deep learning to build a base model from very large amounts of unlabeled data, or even labeled examples for some generic task. This base model can be refined by retraining with a relatively small number of examples that are labeled for your specific task. This semi-supervised kind of learning, together with the distribution of base models, is going to make it vastly easier to apply advanced machine learning to practical problems with only a few examples for training. As this approach becomes more prevalent, the entry costs of building complex machine learning systems are going to drop. That drop is, in turn, going to cause an even larger stampede of people jumping into machine learning to build new systems.

Beyond these semi-supervised systems, we see huge advances in reinforcement learning. These systems are working on very hard problems and aren't working nearly as well (yet) as the newly available image and speech understanding systems. The promise, however, is huge. Reinforcement learning holds a key to helping computers truly interact with the real world. Whether advances in reinforcement learning will happen over the next few years at the pace of other recent advances is an open question. If the pace stays the same, the revolution we have seen so far is going to seem miniscule in a few years.

We can't wait to see what happens. One thing that we do know from experience is that it will be the logistics, not the learning, that will be the key to make the next generation of advances truly valuable.

# APPENDIX A
# Additional Resources

These resources will help you plan, develop, and manage machine learning systems as well as explore the broader topic of stream-first design to build a global data fabric:

- *Computing Extremely Accurate Quantiles Using t-Digests*, by Ted Dunning and Otmar Ertl.
- "Update on the t-Digest: Finding Faults in Real Data". Video of talk by Ted Dunning at Berlin Buzzwords, 13 June 2017.
- "Using TensorFlow on a Raspberry Pi in a Chicken Coop". Tutorial by Ian Downard, 12 July 2017.
- Overview of the Rendezvous Architecture included in "Non-Flink Machine Learning on Flink". Video of talk by Ted Dunning at Flink Forward conference, 14 April 2017.
- "How Stream-1st Architecture & Emerging Technologies Provide a Competitive Edge". Video of talk by Ellen Friedman at Big Data London conference, 4 November 2016.
- "Getting Started with MapR Streams". Technical tutorial with sample code by Tugdual Grall, March 2016.
- *Evaluating Machine Learning Models*, free ebook by Alice Zheng (O'Reilly).

## Selected O'Reilly Publications by Ted Dunning and Ellen Friedman

- *Data Where You Want It: Geo-Distribution of Big Data and Analytics* (March 2017)
- *Streaming Architecture: New Designs Using Apache Kafka and MapR Streams* (March 2016)
- *Sharing Big Data Safely: Managing Data Security* (September 2015)
- *Real World Hadoop* (January 2015)
- *Time Series Databases: New Ways to Store and Access Data* (October 2014)
- *Practical Machine Learning: A New Look at Anomaly Detection* (June 2014)
- *Practical Machine Learning: Innovations in Recommendation* (January 2014)

## O'Reilly Publication by Ellen Friedman and Kostas Tzoumas

- *Introduction to Apache Flink: Stream Processing for Real Time and Beyond* (September 2016)

## About the Authors

**Ted Dunning** is Chief Applications Architect at MapR Technologies and active in the open source community, being a committer and PMC member of the Apache Mahout, Apache ZooKeeper, and Apache Drill projects, and serving as a mentor for the Storm, Flink, Optiq, and Datafu Apache incubator projects. He has contributed to Mahout clustering, classification, matrix decomposition algorithms, and the new Mahout Math library, and recently designed the *t*-digest algorithm used in several open source projects.

Ted was the chief architect behind the MusicMatch (now Yahoo Music) and Veoh recommendation systems, built fraud-detection systems for ID Analytics (LifeLock), and has 24 issued patents to date. Ted has a PhD in computing science from University of Sheffield. When he's not doing data science, he plays guitar and mandolin. Ted is on Twitter as @ted_dunning.

**Ellen Friedman** is Principal Technologist at MapR, and a well-known speaker and author, currently writing mainly about big data topics. She is a committer for the Apache Drill and Apache Mahout projects. With a PhD in biochemistry, she has years of experience as a research scientist and has written about a variety of technical topics, including molecular biology, nontraditional inheritance, and oceanography. Ellen is also coauthor of a book of magic-themed cartoons, *A Rabbit Under the Hat* (The Edition House). Ellen is on Twitter as @Ellen_Friedman.