

Analyzing Data in the Internet of Things

A Collection of Talks from
Strata + Hadoop World 2015



Alice LaPlante



San Jose



London



Beijing



New York



Singapore

Strata+ Hadoop WORLD

Make Data Work
strataconf.com

Presented by O'Reilly and Cloudera, Strata + Hadoop World helps you put big data, cutting-edge data science, and new business fundamentals to work.

- Learn new business applications of data technologies
- Develop new skills through trainings and in-depth tutorials
- Connect with an international community of thousands who work with data

Analyzing Data in the Internet of Things

*A Collection of Talks from
Strata + Hadoop World 2015*

Alice LaPlante

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Analyzing Data in the Internet of Things

by Alice LaPlante

Copyright © 2016 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

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

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

Editor: Shannon Cutt

Proofreader: Susan Moritz

Production Editor: Shiny Kalapurakkal

Interior Designer: David Futato

Copyeditor: Jasmine Kwityn

Cover Designer: Randy Comer

May 2016: First Edition

Revision History for the First Edition

2016-05-13: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Analyzing Data in the Internet of Things*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-95901-5

[LSI]

Table of Contents

Introduction.....	vii
-------------------	-----

Part I. Data Processing and Architecture for the IoT

1. Data Acquisition and Machine-Learning Models.....	1
Modeling Machine Failure	1
2. IoT Sensor Devices and Generating Predictions.....	9
Sampling Bias and Data Sparsity	10
Minimizing the Minimization Error	10
Constrained Throughput	11
Implementing Deep Learning	11
3. Architecting a Real-Time Data Pipeline with Spark Streaming... .	13
What Features Should a Smart City Have?	14
Designing a Real-Time Data Pipeline with the MemCity App	15
The Real-Time Trinity	16
Building the In-Memory Application	16
Streamliner for IoT Applications	17
The Lambda Architecture	18
4. Using Spark Streaming to Manage Sensor Data.....	19
Architectural Considerations	21
Visualizing Time-Series Data	22
The Importance of Sliding Windows	22

Checkpoints for Fault Tolerance	23
Start Your Application from the Checkpoint	24

Part II. Case Studies in IoT Data

5. Monitoring Traffic in Singapore Using Telco Data.....	27
Understanding the Data	27
Developing Real-Time Recommendations for Train Travel	28
Expressway Data	29
6. Oulu Smart City Pilot.....	31
Managing Emergency Vehicles, Weather, and Traffic	32
Creating Situation Awareness	32
7. An Open Source Approach to Gathering and Analyzing Device-Sourced Health Data.....	35
Generating Personal Health Data	36
Applications for Personal Health Data	36
The Health eHeart Project	38
8. Leverage Data Analytics to Reduce Human Space Mission Risks..	41
Over 300,000 Measurements of Data	42
Microsecond Timestamps	43
Identifying Patterns in the Data	43
Our Goal: A Flexible Analytics Environment	44
Using Real-Time and Machine Learning	44
Analytics Using Stream and Batch Processing	46

Part III. Ethics of Algorithms in IoT

9. How Are Your Morals? Ethics in Algorithms and IoT.....	49
Beta Representations of Values	50
Choosing How We Want to Be Represented	51

Introduction

Alice LaPlante

The Internet of Things (IoT) is growing quickly. More than 28 billion things will be connected to the Internet by 2020, according to the International Data Corporation (IDC).¹ Consider that over the last 10 years:²

- The cost of sensors has gone from \$1.30 to \$0.60 per unit.
- The cost of bandwidth has declined by 40 times.
- The cost of processing has declined by 60 times.

Interest as well as revenues has grown in everything from smart-watches and other wearables, to smart cities, smart homes, and smart cars. Let's take a closer look:

Smart wearables

According to IDC, vendors shipped 45.6 million units of wearables in 2015, up more than 133% from 2014. By 2019, IDC forecasts annual shipment volumes of 126.1 million units, resulting in a five-year compound annual growth rate (CAGR) of 45.1%.³ This is fueling streams of big data for healthcare research and development—both in academia and in commercial markets.

¹ Goldman Sachs, "Global Investment Research," September 2014.

² Ibid.

³ IDC, "Worldwide Quarterly Device Tracker," 2015.

Smart cities

With more than 60% of the world's population expected to live in urban cities by 2025, we will be seeing rapid expansion of city borders, driven by population increases and infrastructure development. By 2023, there will be 30 **mega cities** globally.⁴ This in turn will require an emphasis on smart cities: sustainable, connected, low-carbon cities putting initiatives in place to be more livable, competitive, and attractive to investors. The market will continue growing to \$1.5 trillion by 2020 through such diverse areas as transportation, buildings, infrastructure, energy, and security.⁵

Smart homes

Connected home devices will ship at a compound annual rate of more than 67% over the next five years, and will reach 1.8 billion units by 2019, according to BI Intelligence. Such devices include smart refrigerators, washers, and dryers, security systems, and energy equipment like smart meters and smart lighting.⁶ By 2019, it will represent approximately 27% of total IoT product shipments.⁷

Smart cars

Self-driving cars, also known as autonomous vehicles (AVs), have the potential to disrupt a number of industries. Although the exact timing of technology maturity and sales is unclear, AVs could eventually play a "profound" role in the global economy, according to McKinsey & Co. Among other advantages, AVs could reduce the incidence of car accidents by up to 90%, saving billions of dollars annually.⁸

In this O'Reilly report, we explore the IoT industry through a variety of lenses, by presenting you with highlights from the 2015 Strata + Hadoop World Conferences that took place in both the United States and Singapore. This report explores IoT-related topics

⁴ Frost & Sullivan. "Urbanization Trends in 2020: Mega Cities and Smart Cities Built on a Vision of Sustainability," 2015.

⁵ World Financial Symposiums, "Smart Cities: M&A Opportunities," 2015.

⁶ BI Intelligence. The Connected Home Report. 2014.

⁷ Ibid.

⁸ Michelle Bertoncello and Dominik Wee (McKinsey & Co.). Ten Ways Autonomous Driving Could Reshape the Automotive World. June 2015.

through a series of case studies presented at the conferences. Topics we'll cover include modeling machine failure in the IoT, the computational gap between CPU storage, networks on the IoT, and how to model data for the smart connected city of the future. Case studies include:

- Spark Streaming to predict failure in railway equipment
- Traffic monitoring in Singapore through the use of a new IoT app
- Applications from the smart city pilot in Oulu, Finland
- An ongoing longitudinal study using personal health data to reduce cardiovascular disease
- Data analytics being used to reduce risk in human space missions under NASA's Orion program

We finish with a discussion of ethics, related to the algorithms that control the *things* in the Internet of Things. We'll explore decisions related to data from the IoT, and opportunities to influence the moral implications involved in using the IoT.

PART I

Data Processing and Architecture for the IoT

CHAPTER 1

Data Acquisition and Machine-Learning Models

Danielle Dean

Editor's Note: At Strata + Hadoop World in Singapore, in December 2015, Danielle Dean (Senior Data Scientist Lead at Microsoft) presented a talk focused on the landscape and challenges of predictive maintenance applications. In her talk, she concentrated on the importance of data acquisition in creating effective predictive maintenance applications. She also discussed how to formulate a predictive maintenance problem into three different machine-learning models.

Modeling Machine Failure

The term *predictive maintenance* has been around for a long time and could mean many different things. You could think of predictive maintenance as predicting when you need an oil change in your car, for example—this is a case where you go every six months, or every certain amount of miles before taking your car in for maintenance.

But that is not very predictive, as you're only using two variables: how much time has elapsed, or how much mileage you've accumulated. With the IoT and streaming data, and with all of the new data we have available, we have a lot more information we can leverage to make better decisions, and many more variables to consider when predicting maintenance. We also have many more opportunities in terms of what you can actually predict. For example, with all the

data available today, you can predict not just when you need an oil change, but when your brakes or transmission will fail.

Root Cause Analysis

We can even go beyond just predicting when something will fail, to also predicting *why* it will fail. So predictive maintenance includes *root cause analysis*.

In aerospace, for example, airline companies as well as airline engine manufacturers can predict the likelihood of flight delay due to mechanical issues. This is something everyone can relate to: sitting in an airport because of mechanical problems is a very frustrating experience for customers—and is easily avoided with the IoT.

You can do this on the component level, too—asking, for example, when a particular aircraft component is likely to fail next.

Application Across Industries

Predictive maintenance has applications throughout a number of industries. In the utility industry, when is my solar panel or wind turbine going to fail? How about the circuit breakers in my network? And, of course, all the machines in consumers' daily lives. Is my local ATM going to dispense the next five bills correctly, or is it going to malfunction? What maintenance tasks should I perform on my elevator? And when the elevator breaks, what should I do to fix it?

Manufacturing is another obvious use case. It has a huge need for predictive maintenance. For example, doing predictive maintenance at the component level to ensure that it passes all the safety checks is essential. You don't want to assemble a product only to find out at the very end that something down the line went wrong. If you can be predictive and rework things as they come along, that would be really helpful.

A Demonstration: Microsoft Cortana Analytics Suite

We used the Cortana Analytics Suite to solve a real-world predictive maintenance problem. It helps you go from data, to intelligence, to actually acting upon it.

The Power BI dashboard, for example, is a visualization tool that enables you to see your data. For example, you could look at a scenario to predict which aircraft engines are likely to fail soon. The dashboard might show information of interest to a flight controller, such as how many flights are arriving during a certain period, how many aircrafts are sending data, and the average sensor values coming in.

The dashboard may also contain insights that can help you answer questions like “Can we predict the remaining useful life of the different aircraft engines?” or “How many more flights will the engines be able to withstand before they start failing?” These types of questions are where the machine learning comes in.

Data Needed to Model Machine Failure

In our flight example, how does all of that data come together to make a visually attractive dashboard?

Let's imagine a guy named Kyle. He maintains a team that manages aircrafts. He wants to make sure that all these aircrafts are running properly, to eliminate flight delays due to mechanical issues.

Unfortunately, airplane engines often show signs of wear, and they all need to be proactively maintained. What's the best way to optimize Kyle's resources? He wants to maintain engines before they start failing. But at the same time, he doesn't want to maintain things if he doesn't have to.

So he does three different things:

- He looks over the historical information: how long did engines run in the past?
- He looks at the present information: which engines are showing signs of failure today?
- He looks to the future: he wants to use analytics and machine learning to say which engines are likely to fail.

Training a Machine-Learning Model

We took publicly available data that NASA publishes on engine run-to-failure data from aircraft, and we trained a machine-learning model. Using the dataset, we built a model that looks at the relationship between all of the sensor values, and whether an engine is going to fail. We built that machine-learning model, and then we used Azure ML Studio to turn it into an API. As a standard web service, we can then integrate it into a production system that calls out on a regular schedule to get new predictions every 15 minutes, and we can put that data back into the visualization.

To simulate what would happen in the real world, we take the NASA data, and use a data generator that sends the data in real time, to the cloud. This means that every second, new data is coming in from the aircrafts, and all of the different sensor values, as the aircrafts are running. We now need to process that data, but we don't want to use every single little sensor value that comes in every second, or even subsecond. In this case, we don't need that level of information to get good insights. What we need to do is create some aggregations on the data, and then use the aggregations to call out to the machine-learning model.

To do that, let's look at numbers like the average sensor values, or the rolling standard deviation; we want to then predict how many cycles are left. We ingest that data through Azure Event Hub and use Azure Stream Analytics, which lets you do simple SQL queries on that real-time data. You can then do things like select the average over the last two seconds, and output that to Power BI. We then do some SQL-like real-time queries in order to get insights, and show that right to Power BI.

We then take the aggregated data and execute a second batch, which uses Azure Data Factory to create a *pipeline* of services. In this example, we're scheduling an aggregation of the data to a flight level, calling out to the machine-learning API, and putting the results back in SQL database so we can visualize them. So we have information about the aircrafts and the flights, and then we have lots of different sensor information about it, and this training data is actually run-to-failure data, meaning we have data points until the engine actually fails.

Getting Started with Predictive Maintenance

You might be thinking, “This sounds great, but how do I know if I’m ready to do machine learning?” Here are five things to consider before you begin doing predictive maintenance:

What kind of data do you need?

First, you must have a very “sharp” question. You might say, “We have a lot of data. Can we just feed the data in and get insights out?” And while you can do lots of cool things with visualization tools and dashboards, to really build a useful and impactful machine-learning model, you must have that question first. You need to ask something specific like: “I want to know whether this component will fail in the next X days.”

You must have data that measures what you care about

This sounds obvious, but at the same time, this is often not the case. If you want to predict things such as failure at the component level, then you have to have component-level information. If you want to predict a door failure within a car, you need door-level sensors. It’s essential to measure the data that you care about.

You must have accurate data

It’s very common in predictive maintenance that you want to predict a failure occurring, but what you’re actually predicting in your data is not a real failure. For example, predicting fault. If you have faults in your dataset, those might sometimes be failures, but sometimes not. So you have to think carefully about what you’re modeling, and make sure that that is what you want to model. Sometimes modeling a proxy of failure works. But if sometimes the faults are failures, and sometimes they aren’t, then you have to think carefully about that.

You must have connected data

If you have lots of usage information—say maintenance logs—but you don’t have identifiers that can connect those different datasets together, that’s not nearly as useful.

You must have enough data

In predictive maintenance in particular, if you’re modeling machine failure, you must have enough examples of those machines *failing*, to be able to do this. Common sense will tell you that if you only have a couple of examples of things failing,

you're not going to learn very well; having enough raw examples is essential.

Feature Engineering Is Key

Feature engineering is where you create extra features that you can bring into a model. In our example using NASA data, we don't want to just use that raw information, or aggregated information—we actually want to create extra features, such as change from the initial value, velocity of change, and frequency count. We do this because we don't want to know simply what the sensor values are at a certain point in time—we want to look back in the past, and look at *features*. In this case, any kinds of features that can capture degradation over time are very important to include in the model.

Three Different Modeling Techniques

You've got a number of modeling techniques you can choose from. Here are three we recommend:

Binary classification

Use binary classification if you want to do things like predict whether a failure will occur in a certain period of time. For example, will a failure occur in the next 30 days or not?

Multi-class classification

This is for when you want to predict buckets. So you're asking if an engine will fail in the next 30 days, next 15 days, and so forth.

Anomaly detection

This can be useful if you actually don't have failures. You can do things like *smart thresholding*. For example, say that a door's closing time goes above a certain threshold. You want an alert to tell you that something's changed, and you also want the model to learn what the new threshold is for that indicator.

These are relatively simplistic, but effective techniques.

Start Collecting the Right Data

A lot of IoT data is not used currently. The data that is used is mostly for anomaly detection and control, not prediction, which is what can provide us with the greatest value. So it's important to think about what you will want to do in the future. It's important to

collect good quality data over a long enough period of time to enable your predictive analytics in the future. The analytics that you're going to be doing in two or five years is going to be using today's data.

CHAPTER 2

IoT Sensor Devices and Generating Predictions

Bruno Fernandez-Ruiz

Editor's Note: At Strata + Hadoop World in San Jose, in February 2015, Bruno Fernandez-Ruiz (Senior Fellow at Yahoo!) presented a talk that explores two issues that arise due to the computational resource gap between CPUs, storage, and network on IoT sensor devices: (a) undefined prediction quality, and (b) latency in generating predictions.

Let's begin by defining the resource gap we face in the IoT by talking about wearables and the data they provide. Take, for example, an optical heart rate monitor in the form of a GPS watch. These watches measure the conductivity of the photocurrent, through the skin, and infer your actual heart rate, based on that data.

Essentially, it's an input and output device, that goes through some "black box" inside the device. Other devices are more complicated. One example is **Mobileye**, which is a combination of radar/lidar cameras embedded in a car that, in theory, detects pedestrians in your path, and then initiates a braking maneuver. **Tesla** is going to start shipping vehicles with this device.

Likewise, Mercedes has an on-board device called Sonic Cruise, which is essentially a **lidar** (similar to a Google self-driving car). It sends a beam of light, and measures the reflection that comes back. It will tell you the distance between your car and the next vehicle, to

initiate a forward collision warning or even a maneuver to stop the car.

In each of these examples, the device follows the same pattern—collecting metrics from a number of data sources, and translating those signals into actionable information. Our objective in such cases is to find the best function that minimizes the *minimization error*.

To help understand minimization error, let's go back to our first example—measuring heart rate. Consider first that there is an actual value for your real heart rate, which can be determined through an EKG. If you use a wearable to calculate the *inferred value* of your heart rate, over a period of time, and then you sum the samples, and compare them to the EKG, you can measure the difference between them, and minimize the minimization error.

What's the problem with this?

Sampling Bias and Data Sparsity

The key issue is you're only looking at a limited number of scenarios: what you can measure using your device, and what you can compare with the EKG. But there could be factors impacting heart rate that involve temperature, humidity, or capillarity, for example. This method therefore suffers from two things: sampling bias and data sparsity. With *sampling bias*, you've only looked at *some* of the data, and you've never seen examples of things that happen only in the field. So how do you collect those kinds of samples? The other issue is one of *data sparsity*, which takes into account that some events actually happen very rarely.

The moral is: train with as much data as you can. By definition, there is a subsampling bias, and you don't know what it is, so keep training and train with more data; this is *continuous learning*—you're just basically going in a loop all of the time.

Minimizing the Minimization Error

Through the process of collecting data from devices, we minimize error by considering our existing data samples, and we infer values through a family of functions. A key property of all these functions is that they can be parametrized by a vector—which we will call w . We

find out all of these functions, we calculate the error, and one of these functions will minimize the error.

There are two key techniques for this process; the first is *gradient descent*. Using gradient descent, you look at the gradient from one point, walk the curve, and calculate for all of the points that you have, and then you keep descending toward the minimum. This is a slow technique, but it is more accurate than the second option we'll describe.

Stochastic jumping is a technique by which you look at one sample at a time, calculate the gradient for that sample, then jump, and jump again—it keeps approximating. This technique moves faster than gradient descent, but is less accurate.

Constrained Throughput

In computational advertising, which is what we do at Yahoo!, we know that we need two billion samples to achieve a good level of accuracy for a click prediction. If you want to detect a pedestrian, for example, you probably need billions of samples of situations where you have encountered a pedestrian. Or, if you're managing electronic border control, and you want to distinguish between a coyote and a human being, again, you need billions of samples.

That's a lot of samples. In order to process all of this data, normally what happens is we bring all of the data somewhere, and process it through a GPU, which gives you your optimal learning speed, because the memory and processing activities are in the same place. Another option is to use a CPU, where you move data between the CPU and the memory. The slowest option is to use a network.

Can we do something in between, though, and if so, what would that look like? What we can do is create something like a true *distributed hash table*, which says to every computational node, “I'm going to spin off the storage node,” and you start routing requests.

Implementing Deep Learning

Think about dinosaurs. They were so big that the electrical impulses that went through their neurons to their backbone would take too long. If a dinosaur encountered an adversary, by the time the signals went to the brain and made a decision, and then went to the tail,

there was a lag of several milliseconds that actually mattered to survival. This is why dinosaurs had two or more brains—or really, approximations of brains—which could make fast decisions without having to go to “the main CPU” of the dinosaur (the brain). Each brain did not have all of the data that the main brain had, but they could be fast—they could move the dinosaur’s limbs in times of necessity.

While deep learning may not always be fast, the number of applications that it opens up is quite immense. If you think about sensor-area networks and wireless sensor networks in applications from 5–10 years ago, you’ll see that this is the first time where machine-to-machine data is finally becoming possible, thanks to the availability of cheap compute, storage, and sensory devices.

CHAPTER 3

Architecting a Real-Time Data Pipeline with Spark Streaming

Eric Frenkiel

Editor’s Note: At Strata + Hadoop World in Singapore, in December 2015, Eric Frenkiel (CEO and cofounder at MemSQL) presented a talk that explores modeling the smart and connected city of the future with Kafka and Spark.

Hadoop has solved the “volume” aspect of big data, but “velocity” and “variety” are two aspects that still need to be tackled. In-memory technology is important for addressing velocity and variety, and here we’ll discuss the challenges, design choices, and architecture required to enable smarter energy systems, and efficient energy consumption through a real-time data pipeline that combines Apache Kafka, Apache Spark, and an in-memory database.

What does a smart city look like? Here’s a familiar-looking vision: it’s definitely something that is futuristic, ultra-clean, and for some reason there are always highways that loop around buildings. But here’s the reality: we have a population of almost four billion people living in cities, and unfortunately, very few cities can actually enact the type of advances that are necessary to support them.

A full 3.9 billion people live in cities today; by 2050, we’re expected to add another 2.5 billion people. It’s critical that we get our vision of a smart city right, because in the next few decades we’ll be adding billions of people to our urban centers. We need to think about how

we can design cities and use technology to help people, and deliver real value to billions of people worldwide.

The good news is that the technology of today *can* build smart cities. Our current ecosystem of data technologies—including Hadoop, data warehouses, streaming, and in-memory—can deliver phenomenal technology at a city-level scale.

What Features Should a Smart City Have?

At its most minimum, a smart city should have four features:

- City-wide WiFi
- A city app to report issues
- An open data initiative to share data with the public
- An adaptive IT department

Free Internet Access

With citywide WiFi, anyone in the city should be able to connect for free. This should include support for any device that people happen to own. We're in a time when we should really consider access to the Internet as a fundamental human right. The ability to communicate and to share ideas across cities and countries is something that should be available for all. While we're seeing some initiatives across the world where Internet is offered for free, in order to build the applications we need today, we have to blanket every city with connectivity.

Two-Way Communication with City Officials

Every city should have an application that allows for two-way communication between city officials and citizens. Giving citizens the ability to log in to the city app and report traffic issues, potholes, and even crime, is essential.

Data Belongs to the Public

When it comes to the data itself, we have to remember that it belongs to the public. Therefore, it's incumbent upon the city to make that data available. San Francisco, for example, does a phe-

nomenal job of giving public data to the community to use in any way. When we look at what a smart city should become, it means sharing data so that everyone can access it.

Empower Cities to Hire Great Developers

Most importantly, every city that is serious about becoming smart and connected needs to have an adaptive, fast-moving IT department. If we want to get our public sector moving quickly, we have to empower cities with budgets that let them hire great developers to work for the city, and build applications that change people's lives.

Designing a Real-Time Data Pipeline with the MemCity App

Let's discuss an example that utilizes a real-time data pipeline—the application called MemCity. This application is designed to capture data from 1.4 million households, with data streaming from eight devices, in each home, every minute. What this will do is let us pump 186,000 transactions per second from Kafka, to Spark, to MemSQL.

That's a lot of data. But it's actually very cheap to run an application like this because of the cloud—either using Amazon or other cloud services. Our example is only going to cost \$2.35 an hour to run, which means that you're looking at about \$20,000 annually to operate this type of infrastructure for a city. This is very cost-affordable, and a great way to demonstrate that big data can be empowering to more than just big companies.

In this example, we're going to use a portfolio of products that we call the Real-Time Trinity—Kafka, Spark, and MemSQL—which will enable us to avoid disk as we build the application. Why avoid disk? Because disk is the enemy of real-time processing. We are building *memory-oriented architectures* precisely because disk is glacially slow.

The real-time pipeline we'll discuss can be applied across any type of application or use case. In this particular example, we're talking about smart cities, but there are many applications that this architecture will support.

The Real-Time Trinity

The goal of using these three solutions—Kafka, Spark, and MemSQL—is to create an end-to-end data pipeline in under one second.

Kafka is a very popular, open source high-throughput distributed messaging system, with a strong community of support. You can publish and subscribe to Kafka “topics,” and use it as the centralized data transport for your business.

Spark is an in-memory execution engine that is transient (so it’s not a database). Spark is good for high-level operations for procedural and programmatic analytics. It’s much faster than MapReduce, and you’re able to do things that aren’t necessarily expressible in a conventional declarative language such as SQL. You have the ability to model anything you want inside this environment, and perform machine learning.

MemSQL is an in-memory distributed database that lets you store your state of the model, capture the data, and build applications. It has a SQL interface for the data streaming in, and lets you build real-time, performant applications.

Building the In-Memory Application

The first step is to subscribe to Kafka, and then Kafka serializes the data. In this example, we’re working with an event that has some information we need to resolve. We publish it to the Kafka topic, and it gets zipped up, serialized, and added to the event queue. Next, we go to Spark, where we’ll deserialize the data and do some enrichment. Once you’re in the Spark environment, you can look up a city’s zip code, for example, or map a certain ID to a kitchen appliance.

Now is the time for doing our real-time ingest; we set up the Kafka feed, so data is flowing in, and we’re doing real-time transformations in the data—cleaning it up, cleansing it, getting it in good order. Next, we save the data and log in to the MemCity database, where you can begin looking at the data itself using Zoomdata. You can also connect it to a business intelligent application, and in this case, you can compress your development timelines, because you have the data flowing in through Spark and Kafka, and into MemSQL. So in

effect, you're moving away from the concept of analyzing data via reports, and toward real-time applications where you can interact with live data.

Streamliner for IoT Applications

Streamliner is a new open source application that gives you the ability to have one-click deployment of Apache Spark. The goal is to offer users a simple way to reduce data loading latency to zero, and start manipulating data. For example, you can set up a GUI pipeline, click on it, and create a new way to consume data into the system. You can have multiple data pipelines flowing through, and the challenge of “how do I merge multiple data streams together?” becomes trivial, because you can just do a basic join.

But if we look at what justifies in-memory technology, it's really the fact that we can eliminate extract, transform, and load (ETL) activities. For example, you might look at a batch process and realize that it takes 12 hours to load the data. Any query that you execute against that dataset is now at least 12 hours too late to affect the business.

Now, many database technologies, even in the ecosystem of Hadoop, are focused on reducing query execution latency, but the biggest improvements you can make involve reducing data *loading* latency —meaning that the faster you get access to the data, the faster you can start responding to your business.

From an architectural perspective, it's a very simple deployment process. You start off with a raw cluster, and then deploy MemSQL so that you can have a database cluster running in your environment, whether that's on-premise, in the cloud, or even on a laptop. The next step is that one-click deployment of Spark. So you now have two processes (a MemSQL process and Spark process) co-located on the same machine.

The benefit of having two processes on the same machine is that you can avoid an extra network hop. To complete this real-time data pipeline, you simply connect Kafka to each node in the cluster, and then you get a multi-threaded, highly parallelized write into the system. What you're seeing here is memory-to-memory-to-memory, and then behind the scenes MemSQL operates the disk in the background.

The Lambda Architecture

All of this touches on something broader than in-memory—it's about extending analytics with what is called a *Lambda architecture*. The Lambda architecture enables a real-time data pipeline going into your systems, so that you can manipulate the data very quickly. If your business is focused around information, using in-memory technology is critical to out-maneuver in the marketplace.

With Lambda architecture, you get analytic applications, not Excel reports. An Excel report will come out of a data warehouse and it will arrive in your inbox. An analytic application is live data for you to analyze, and of course, it's all predicated on real-time analytics. You have the ability to look at live data and change the outcome.

The notion of getting a faster query is nice. It might save you a cup of coffee or a trip around the block while you are waiting, but the real benefit is that you can leverage that analytic to respond to what's happening now in your business or market. Real-time analytics has the potential to change the game in how businesses strategize, because if you know things faster than competitors, you're going to outcompete them in the long run.

CHAPTER 4

Using Spark Streaming to Manage Sensor Data

Hari Shreedharan and Anand Iyer

Editor’s Note: At Strata + Hadoop World in New York, in September 2015, Hari Shreedharan (Software Engineer at Cloudera) and Anand Iyer (Senior Product Manager at Cloudera) presented this talk, which applies Spark Streaming architecture to IoT use cases, demonstrating how you can manage large volumes of sensor data.

Spark Streaming takes a continuous stream of data and represents it as an *abstraction*, called a *discretized stream*. This is commonly referred to as a DStream. A DStream takes the continuous stream of data and breaks it up into disjoint chunks called microbatches. The data that fits within a microbatch—essentially the data that streamed in within the time slot of that microbatch—is converted to a resilient distributed dataset (RDD). Spark then processes that RDD with regular RDD operations.

Spark Streaming has seen tremendous adoption over the past year, and is now used for a wide variety of use cases. Here, we’ll focus on the application of Spark Streaming to a specific use case—*proactive maintenance* and *accident prevention* in railways.

To begin, let’s keep in mind that the IoT is all about *sensors*—sensors that are continuously producing data, with all of that data streaming into your data center. In our use case, we fitted sensors to railway locomotives and railway carriages. We wanted to resolve two different issues from the sensor data: (a) identifying when there is damage

to the axle or wheels of the railway locomotive or railway carriages; and (b) identifying damage on the rail tracks.

The primary goal in our work was to prevent derailments, which result in the loss of both lives and property. Though railway travel is one of the safest forms of travel, any loss of lives and property is preventable.

Another goal was to lower costs. If you can identify issues early, then you can fix them early; and in almost all cases, fixing issues early costs you less.

The sensors placed on the railway carriages are continuously sending data, and there is a unique ID that represents each sensor. There's also a unique ID that represents each locomotive. We want to know how fast the train was going and the temperature, because invariably, if something goes wrong, the metal heats up. In addition, we want to measure pressure—because when there's a problem, there may be excessive weight on the locomotive or some other form of pressure that's preventing the smooth rotation of the wheels.

The sound of the regular hum of an engine or the regular rhythmic spinning of metal wheels on metal tracks is very different from the sound that's produced when something goes wrong—that's why acoustic signals are also useful. Additionally, GPS coordinates are necessary so that we know where the trains are located as the signals stream in. Last, we want a timestamp to know when all of these measurements are taken. As we capture all of this data, we're able to monitor the readings to see when they increase from the baseline and get progressively worse—that's how we know if there is damage to the axle or wheels.

Now, what about damage to the rail tracks? Damage on a railway track occurs at a specific location. With railway tracks you have a left and right track, and damage is likely on one side of the track, not both. When a wheel goes over a damaged area, the sensor associated with that wheel will see a spike in readings. And the readings are likely to be acoustic noise, because you'll have the metal clanging sound, as well as pressure. Temperature may not come into play as much because there probably needs to be a sustained period of damage in order to affect this reading. So in the case of a damaged track, acoustic noise and pressure readings are likely to go up, but it will be a *spike*. The minute the wheel passes that damaged area, the readings

will come back down—and that's our cue for damage on a railway track.

Architectural Considerations

In our example, all of these sensor readings have to go from the locomotive to the data center. The first thing we do when the data arrives is write it to a reliable, high-throughput streaming channel, or streaming transportation layer—in this case, we use Kafka. With the data in Kafka, we can read it in Spark Streaming, using the direct Kafka connector.

The first thing we do when these events come into the data center is enrich them with relevant metadata, to help determine if there is potential damage. For example, based on the locomotive ID, we want to fetch information about the locomotive, such as the type—for example, we would want to know if it's a freight train, if it's carrying human passengers, how heavy it is, and so on. And if it is a freight train, is it carrying hazardous chemicals? If that's the case, we would probably need to take action at any hint of damage. If it's a freight train that's just coming back empty, with no cargo, then it's likely to be less critical. For these reasons, information about the locomotive is critical.

Similarly, information about each sensor is critical. You want to know where the sensor is on the train (i.e., is it on the left wheel or the right wheel?). GPS information is also important because if the train happens to be traveling on a steep incline, you might expect temperature readings to go up. The Spark HBase model, which is now a part of the HBase code base, is what we recommend for pulling in this data.

After you've enriched these events with all the relevant metadata, the next task in our example is to determine whether a signal indicates damage—either through a simple rule-based or predictive model. Once you've identified a potential problem, you write an event to a Kafka queue. You'll have an application that's continuously listening to alerts in the queue, and when it sees an event, the application will send out a physical alert (i.e., a pager alert, an email alert, or a phone call) notifying a technician that something's wrong.

One practical concern here is with regard to data storage—it's helpful to dump all of the raw data into HDFS, for two reasons. First,

keeping the raw data allows data scientists to play with the data, and possibly uncover new insights. Second, there will likely be bugs in your application, and in your code, and you'll want to do an audit when things go wrong. Having the raw data in HDFS lets you write simple batch jobs to figure out when things are wrong, either in your application logic, or in certain cases, where the sensors might have gone wrong.

Visualizing Time-Series Data

Once a technician knows that there's a potential problem, it's time to diagnose the issue. In order to diagnose the issue, the technician will have to look at readings from the sensors as *time-series data*—over different windows of time. Being able to visualize *when* readings occurred is enormously helpful; [Grafana](#) is one open source tool for doing this, and you can always build something quickly using JavaScript. Once the technician has diagnosed the issue—depending on what the problem is—he can either specify that the train be sent for regular maintenance, or that it be stopped because it is carrying passengers or hazardous chemicals.

The Importance of Sliding Windows

Sliding windows are critical in Spark Streaming. You always want to specify a time period on which you want to apply your operation. Rather than writing a custom code variant for looking into each piece of data, to query whether anything happened in the last five hours or last five minutes, you can implement a *windowed structure*.

A window DStream basically has a window interval, which is the window in which you want to look at all of your previous events. You also have a sliding interval that you can keep moving forward. When you apply an operation, you apply it to individual windows. Instead of applying operations on individual RDDs, you apply the operation on all RDDs that arrive within a *specified window*. You can have this window as any multiple of your microbatch interval. You don't want these windows to be long, because most of this data is either cached in memory or written to local disk. If your window's size becomes, say, more than 24 hours, and you're getting a million events per hour, then you're going to see a lot of data being stashed in memory or written to disk, and your performance is going to suffer.

There is an API called `updateStateByKey` that is very useful. Given a key, you can apply any random operation on the previous value with new information that you received over the last n minutes. So you can combine windowing and `updateStateByKey` to apply these operations for windows. You would take your incoming data, put it into a window DStream with a specified window, and then apply the state transformations using `updateStateByKey`.

Checkpoints for Fault Tolerance

One of the most important things about `updateStateByKey` or windowing is that you always want to enable *checkpointing*. Checkpoints are used primarily for *fault tolerance*.

Think about it: what happens if an RDD goes missing from memory? If you've used 60% or 70% of your memory, Spark will drop the RDDs. At that point, you want to reconstruct those RDDs.

The Spark idea of failure tolerance is to get the original data and apply the series of transformations that led to that RDD in the first place. The problem is that it has to apply a large number of operations on the *original* data. If this original data is from several weeks ago, you could possibly use up all of your stack by just applying operations. You could end up with a stack overflow and your operation would never complete. In that case, you want to truncate that chain of events, that chain of transformations, over the last n days, and pick up the latest (as late as possible) value of the keys.

Spark will checkpoint the state of that RDD at any point in time, and do a persistent storage like HDFS. So when you have an RDD that has a long chain of events, but has a checkpoint, Spark will simply recover from the checkpoint rather than trying to apply all of the operations. So checkpointing will save your application from either long-chain processing or huge stack overflow errors. And because a checkpoint is in the state of the application when it died, you recover from where the failure happened.

It's fairly simple to write an application that restarts from a checkpoint. Instead of just creating a new streaming context, you apply a function to create a new streaming context. And that function looks at the checkpoint. If the checkpoint is there, it reads from that instead of reading directly—creating a new Spark Streaming context directly.

Start Your Application from the Checkpoint

Checkpoints are terrific for fault tolerance or restarting your applications, but they're not good for upgrades. Checkpoints in Spark are Java-serialized. Anyone who has ever used Java serialization knows that if you upgrade your application, you change your code, you change your classes, and your serialization is then useless.

The problem is that your checkpoint had all of your data. If you change your application, suddenly all that data has disappeared—not good. Most users want an application that does checkpointing, but they also want to upgrade their applications.

In that case, what do we do? How do you upgrade a checkpoint? The challenge is that your data would need to be separated from your code. Because checkpoints are serialized classes, your data is now tied into your code.

The answer is pretty simple if you think about it. *Your application has its own data.* You know what you want to keep track of; it's usually some RDD that has been generated from your operations. It is in some state that you generated from `updateStateByKey`, and it's usually the last offsets from Kafka that were reliably processed and written out to HDFS. So if you know what you want to process and save, why not do it separately?

Enable checkpointing in your application so that the truncation of your chain happens all the time, but don't use Spark Streaming context. Instead, get a create method to start your application from the checkpoint. When you start your application, you start off fresh—don't use the get or create. Instead, read the state that you wrote out, and then apply your operations from that point on.

PART II

Case Studies in IoT Data

CHAPTER 5

Monitoring Traffic in Singapore Using Telco Data

Thomas Holleczek

Editor's Note: At Strata + Hadoop World in Singapore, in December 2015, Thomas Holleczek (Data Scientist at Singtel) outlined this case study to illustrate how telecommunications companies are using location data to develop a system for subway and expressway traffic monitoring.

People take a lot of factors into consideration when they travel on mass transit or a highway. They don't always take the shortest route. Perhaps they want a less-crowded bus or subway, they want to take a scenic route, or they don't want to have to change buses or subways more than once.

At Singtel, we found that we could use telco location data to understand how people travel on transportation networks. We studied the Singapore transportation system using data from Singtel.

Understanding the Data

Singtel maintains a *location-based system*. Every time you use your cell phone in Singapore, the location gets recorded. This happens in both active and passive events. An active event is when you text someone. Both your and your correspondent's locations are recorded. The location in this case would be the cell towers to which the phones are connected. This happens in real time, and the data streams into the system through Kafka. The data is anonymized, but

because the ID of the phone is constant, we can follow a person over time, and learn about the behavior of people.

The Singapore transportation system generates about 200 million records per day. This translates to a location point every 15 minutes, per user. If you travel, there tend to be a lot more records, because your phone updates the location (what we refer to as passive events). If you travel on the train, for example, this might happen every two stations. If you travel on expressways, this also happens frequently. So, we can follow people as they move through the city.

Developing Real-Time Recommendations for Train Travel

A recent client asked us to determine the size of crowds in the Singapore subway system (MRT) in both stations and trains, using real-time cell phone location data. Our task was to determine the number of people who travel inside the MRT network, how long they travel, and how long it takes them to get from point A to point B.

We developed a system based on Kafka, to detect people on trains—we know where they get on a train, where they get off, how they interchange based on our location data. We found that the busiest connections in Singapore are usually between Raffles Place and City Hall (two stations located in the downtown finance center, where most people work).

Most MRT stations in Singapore are underground. In some of these stations we have 25–40 cell towers, and others have only 8–10. The cell towers specifically serve the platforms of each station and the tunnels. When we see a network event, we can tell the person is inside the station, because you can't connect to indoor cell towers when you're outdoors. We can also tell whether you're in the tunnel or on the platform, because the tunnel has particular cell towers that only serve the tunnels.

When you start moving along a tunnel, the cell phones produce location updates. This typically happens every two to three minutes. At almost every station, there's a location update. The trains are usually pretty crowded. When you travel in the morning, you can't pull out a phone. You can't surf the Web, because it's just too crowded.

But your phone still produces updates just because the phone updates location with the cell towers.

The result: an accurate understanding of how crowded individual stations and trains are at any given moment in the Singapore MRT system. Based on this, we are currently developing an app that recommends routes to subway riders. When we release it, you'll be able to tell the app where you are, and where you want to travel, and the app will provide you with options based on real-time data, including additional information including the current available capacity on a train, whether there are seats available, and estimated travel time.

Expressway Data

When we do this kind of research project, we usually start off with experimentation—we take phones, and we head out and make experiments; then we look at the data that we record.

For this part of the project, we drove around on the expressways. Our fear was that not enough data would be generated, because most people don't use their phones when they drive (or they shouldn't); they don't text and they don't use data. This meant that we would be completely dependent on *passive* updates.

The terrific thing we found out was that when you start driving your car, your phone produces a lot of *location* updates. In our experiment, we found handovers happening between cell towers along the expressways, every three to five minutes. We were also able to detect people who travel on buses, as most buses in Singapore are equipped with machine-to-machine SIM cards, which allow the operators to know bus locations. Most of the buses also have a GPS device, and they transmit their location through the Singtel 3G network.

Getting a full view of what's happening with transportation in Singapore allows us to address several challenges—it can help commuters choose more efficient routes, it can serve as an aid in city planning, and allow the subway system to improve operations, maintenance, and planning of the network.

CHAPTER 6

Oulu Smart City Pilot

Susanna Pirttikangas

Editor's Note: At Strata + Hadoop World in New York, in September 2015, Susanna Pirttikangas (Project Researcher at the University of Oulu) outlined the fully integrated use of IoT data in one of the top seven smart cities in the world, the city of Oulu, in Finland. Oulu continuously collects data from transportation, infrastructure, and people, and develops services on top of the ecosystem that benefit the city, the ecology, the economy, and the people. This talk presents selected examples from a four-year project called "Data to Intelligence," based on a smart traffic pilot, as a testing platform.

The Intelligence Community Forum (ICF)—a New York-based think tank—has voted Oulu as one of the top seven of the smartest cities of the world, twice in a row. According to the ICF's definition, smart cities are cities and regions that use technology not just to save money, or make things work better, but also to create high-quality employment, increase citizen participation, and in general be great places to live and work. We're a very small city—only 200,000 inhabitants, but this is a good thing, as it allows us to pilot new services in an easy and agile manner.

Managing Emergency Vehicles, Weather, and Traffic

One of the things we've done in Oulu is a preemption for emergency vehicles. Whenever there's an alarm for an emergency vehicle, the system that operates the traffic lights locates the ambulance, police car, or whatever emergency vehicle is in action, and the location is sent to the servers. The turning signal for each vehicle is detected, so the system knows if the ambulance is going to turn, and it will be given a green light across the road in real time.

Within Oulu, we collect *magnetic loop data* from below the pavement, or asphalt, and we use this data to control the traffic signaling system. We also can get data from traffic cameras, public transports, and bus location data. We use Digiroad, which is a national spatial database, but also Google maps and Open Street Maps.

We also collect road weather data, which is really important in Finland, in the wintertime. The city uses laser-range measurement devices to detect the speed of the vehicles, and the distance between them. We can even detect the profile of a vehicle and the amount of snow on the road. In addition, we receive location data from taxis operating throughout Finland, using onboard diagnostic connectors that gather detailed information from the vehicles' engines. We also have real-time information about construction projects throughout the city, which comes from city authorities, or crowd-sourced workers.

Creating Situation Awareness

All of this data collection is centered around the idea of *situation awareness*. In order to avoid traffic congestion, decrease emissions, and increase safety on the roads, you need information about traffic speeds, construction, weather, and even available parking spaces. You also need information about other drivers, bus locations, vehicles, vehicles behind corners, and so on.

It's also important to have a high *volume* of data from each of these sources. With enough data, you can make reliable assumptions about the situation, and in traffic, where situations emerge fast, you need the information to be updated and delivered quickly. One

example relates to braking distance. In wintertime, the roads are more slippery and the necessary distance you need in order to stop your vehicle can be more than five times longer than usual.

We developed a system that detects the speed of the vehicles, the distance between the vehicles, and the condition of the road, and then sends a warning to the driver if there's too short a distance between their car and the next, according to the circumstances. To do this, we used an onboard diagnostics device, with data delivered through Bluetooth. For measuring the distance between vehicles, we use the laser-range measurement, either in the infrastructure, or in the car. We also used a camera—if you put a camera in the front window, you can estimate the distance between vehicles from the images (but that is actually not as reliable as the laser-range measurement, because the weather conditions affect this considerably).

The biggest data in this example is the weather prediction. The Finnish Meteorological Institute collects a huge amount of information from the atmosphere, gathered from different kinds of equipment radars. Because this data is so big, the estimation—the data prediction—cannot be done more than four times a day. Each prediction brings four terabytes of data into the system.

We have two kinds of approaches for processing all of this data: (a) distributed reasoning, with lightweight RDF data, and (b) mobile agents that process data in different locations in the network, based on available resources. After testing several types of big data platforms, we selected the Lambda architecture. We are also testing distributed Flume ingestion for better ingestion rates, and higher availability.

CHAPTER 7

An Open Source Approach to Gathering and Analyzing Device-Sourced Health Data

Ian Eslick

Editor's Note: At Strata + Hadoop World in New York, in September 2015, Ian Eslick (CEO and cofounder of VitalLabs) presented a case study that uses an open source technology framework for capturing and routing device-based health data. This data is used by healthcare providers and researchers, focusing particularly on the Health eHeart initiative at the University of California San Francisco.

This project started by looking at the ecosystems that are emerging around the IoT—at data being collected by companies like Validic and Fitbit. Think about it: one company has sold a billion dollars' worth of pedometers, and every smartphone now collects your step count. What can this mean for healthcare? Can we transform clinical care, and the ways in which research is accomplished?

The Robert Wood Johnson Foundation (RWJ) decided to do an experiment. It funded a deep dive into one problem surrounding research in healthcare. Here, we give an overview of what we learned, and some of our suggestions for how the open source community, as well as commercial vendors, can play a role in transforming the future of healthcare.

Generating Personal Health Data

Personal health data is the “digital exhaust” that is created in the process of your everyday life, your posting behaviors, your phone motion patterns, your GPS traces. There is an immense amount of data that we create just by waking up and moving around in the modern world, and the amount of that data is growing exponentially.

Mobile devices now allow us to elicit data directly from patients, bringing huge potential for clinicians to provide better care, based on the actual data collected. As we pull in all of this personal data, we also have data that’s flowing in through the traditional medical channels, such as medical device data. For example, it’s becoming common for implanted devices to produce data that’s available to the patient and physician. There is also the more traditional healthcare data, including claims histories and electronic medical records.

So when researchers look at clinical data, we’re accustomed to living in a very particular kind of world. It’s an episodic world, of low volume—at least relatively low volume by IoT standards. Healthcare tends to be a reactive system. A patient has a problem. He or she arranges a visit. They come in. They generate some data. When a payer or a provider is looking at a population, what you have are essentially the notes and lab tests from these series of visits, which might occur at 3-, 6-, or 12-month intervals.

Personal health data, on the other hand, is consistent, longitudinal, high volume, and noisy. We can collect data over a period of time and then look back on and try to learn from it. The availability of personal health data is changing the model of how healthcare, as a clinical operation, looks at data. It’s also changing how researchers process and analyze that data to ask questions about health. Interestingly, it is relatively cheap to produce, compared to what it costs to produce data in traditional healthcare.

Applications for Personal Health Data

There is a whole set of applications that come out of the personal health data ecosystem. We are at the beginning of what is a profound shift in the way healthcare is going to operate. There is potential for both an open source and commercial ecosystem that supports “ultra scale” research and collaboration within the tradi-

tional healthcare system, and which supports novel applications of personal health data.

The five “C’s” of healthcare outline some of the key topics to consider in this field:

Complexity

Healthcare data can be based on models that are completely different from models commonly used in enterprise data. The sheer complexity of the data models, and the assumptions that you can make in healthcare, are unique.

Computing

There are reasons why healthcare is so difficult to do well. Interoperability is a challenge that we’re still trying to figure out 20 years after EMR was introduced. Those in healthcare are still asking questions like “How do I get my record from one hospital to another?” and “How do I aggregate records across multiple hospitals into a single data center?”

Context

The context in which a particular data item was collected is usually found in notes, not metadata. So again, you can’t filter out those bad data points based on some metadata, because nobody cares about entering the data for the purposes of automated analysis. They care about doing it for purposes of communicating to another human being.

Culture

Many times, an IT department at a hospital already has a tool that will allow, for example, interoperability—but they may not know about it. Accountants, not IT innovators, often run IT departments, because there is a huge liability associated with getting anything wrong, which notably, can be a counterbalance to innovation.

Commerce

In healthcare, payment models don’t change quickly, and payment due must be proven through clinical evidence. You need to find a revenue stream that exists and figure out how to plug into that—and that severely limits innovation.

The Health eHeart Project

Health eHeart started a few years ago at UCSF, with the aim to replace the Framingham Study. Framingham is a decades-old, longitudinal study of a population of 3,000 people in Framingham, Massachusetts. It is the gold standard of evidence that's used to drive almost all cardiovascular care worldwide. People in small, rural towns in India are being treated based on risk models calculated by these 3,000 people in Framingham. Why? Because it is the only dataset that looks at a multi-decade-long evolution of cardiovascular health.

The hypothesis of the UCSF team was that with technology we could dramatically lower the cost of doing things like Framingham over the long term while adding tremendous depth to our understanding of individual patients. All we would need to do is grab data off of their phones. The phone can become a platform for testing new mobile health devices.

Anybody can sign up to volunteer for Health eHeart. It'll send you six-month follow-ups, and you can also share all of your Fitbit data.

The goals of Health eHeart include:

- Improve clinical research cycle time
- Provide a test bed for new health technology innovations on a well-characterized cohort
- Derive new prediction, prevention, and treatment strategies
- Pilot new healthcare delivery systems for cardiovascular disease

Now, how do we test out what we learn and actually deliver it into the clinical care system?

This is an example of the kind of profile that's created when you look at a contributor to Health eHeart. We have blood pressure, which is a key measure of cardiovascular disease, and we have people with Bluetooth blood-pressure cuffs uploading their data on a longitudinal basis to the cloud. We're also following weight.

By late 2014, Health eHeart had 25,000 registered users, 11,000 ECG traces, and 160,000 Fitbit measures from the 1,000 users who were giving us longitudinal data, and these numbers are climbing aggressively; the goal is to get to one million.

You have two colliding worlds here in the Health eHeart context. Clinical researchers understand population data, and they understand the physiology. They understand what is meaningful at the time, but they don't understand it from the standpoint of doing time-series analysis. It's a qualitatively different kind of analysis that you have to do to make sense of a big longitudinal dataset, both at an individual level and at a population level. For example, is there a correlation between your activity patterns as measured by a Fitbit and your A-fib events, as measured by your ECG with the AliveCor ECG device? That's not a question that has ever been asked before. It couldn't possibly be asked until this dataset existed.

What you immediately start to realize is that data is multi-modal. How do you take a 300-Hertz signal and relate that to an every few minutes summary of your pedometer data, and then measure that back against the clinical record of that patient to try to make sense of it?

This data is also multi-scale. There is daily data, and sometimes the time of day matters. Sometimes the time of month matters. Data has a certain inherent periodicity. You're also dealing with three-month physical follow-ups with doctors, so you want to try to take detailed, deep, longitudinal data and mash it up against these clinical records.

Registration is a surprisingly interesting challenge—particularly when considering: what is my baseline? If time of day is important, and you're trying to look at the correlations of activity to an event that happened within the next hour, you might want to align all the data points by hour. But then the number of such aggregated points that you get is small. The more that you try to aggregate your individual data, the more general your dataset becomes, and then it's harder to ask specific questions, where you're dealing with time and latency.

Registration problems require a deep understanding of the question you're trying to answer, which is not something the data scientists usually know, because it's a deep sort of physiological question about what is likely to be meaningful. And obviously, you've got lots of messy data, missing data, and data that's incorrect. One of the things you realize as you dig into this, is the scale that you need to get enough good data—and this is ignoring issues of selection bias—that you can really sink your teeth into and start to identify interesting phenomena.

The big takeaway is that naive analysis breaks down pretty quickly. All assumptions about physiology are approximations. For any given patient, they're almost always wrong. And none of us, turns out, is the average patient. We have different responses to drugs, different side effects, different patterns. And if you build a model based on these assumptions, when you try to apply it back to an individual case, it turns out to be something that only opens up more questions.

Health eHeart Challenges

At an ecosystem level, the challenges we faced in the Health eHeart project included limited resources, limited expertise in large-scale data analysis, and even just understanding how to approach these problems. The working model for Health eHeart has been to find partnerships that are going to allow us to do this.

This is where we started running into big problems with the health-care ecosystem. I can go to UCSF, plug my computer in, pull data down to my system, and perform an analysis on it. But if I try to do that from an outside location, that's forbidden because of the different kinds of regulations being placed on the data in the Health eHeart dataset. What you can do with some of the data is also limited by HIPAA.

This was one of the first problems we addressed, by creating the "trusted analytics container." We created a platform where we can take the medical data from the IRB study, and vendor data from a third-party system, and bring them together in the cloud, where analysts can do their calculations, do their processing, and essentially queue up the resulting aggregated data. Then the UCSF owner of the data reviews the results to make sure you're not leaking personal health information. This process is completed within a data-use agreement. Ultimately, however, commercial collaborators still need a way to create commercial collaboration around scaled access to longitudinal time-series data.

CHAPTER 8

Leverage Data Analytics to Reduce Human Space Mission Risks

Haden Land and Jason Loveland

Editor's Note: At Strata + Hadoop World in New York, in September 2015, Haden Land (Vice President, Research & Technology at Lockheed Martin) and Jason Loveland (Software Engineer at Lockheed Martin) presented a case study that uses data analytics for system testing in an online environment, to reduce human space flight risk for NASA's Orion spacecraft.

NASA's Orion spacecraft is designed to take humans farther than they've ever been able to travel before. Orion is a multi-purpose crew vehicle, and the first spacecraft designed for long-duration space flight and exploration. The program is focused on a sustainable and affordable effort for both human and robotic exploration, to extend the presence of human engagement in the solar system and beyond.

In December 2014, the Delta IV rocket launched from Cape Canaveral carrying the Orion vehicle. The mission was roughly four hours and 24 minutes—a fairly quick mission. It orbited the earth twice. The distance was approximately 15 times farther than the International Space Station.

There were tremendous data points that were gathered. One that was particularly interesting was that Orion traveled twice through the Van Allen radiation belt. That's a pretty extreme test, and this vehicle did that twice, and was exposed to fairly substantial radia-

tion. Toward the end of the flight, upon entering the atmosphere, the vehicle was going 20,000 miles per hour, and sustained heat of an excess of 4,000 degrees Fahrenheit. As the parachute deployed, it slowed down to 20 miles per hour before it splashed in the ocean, about 640 miles south-southwest of San Diego. A ship gathered it and brought it back home.

Over 300,000 Measurements of Data

Our job is to enable the Orion program to capture all the information coming off the test rigs, and to help those working on the rigs to understand the characteristics of every single component of the Orion spacecraft. The goal is to make launches like this successful. The next mission is going to focus on human space flight: making it *human rated*—that is, able to build a vehicle that can go up into space and carry astronauts. With EFT-1 (the first launch that we just described), there were *350,000 measurements*—from sensors for everything from temperature control systems, to altitude control systems. We were collecting two terabytes of data per hour from 1,200 telemetry sensors that reported from the spacecraft, 40 times per second.

When NASA tests from the ground, it's the same story. They're testing the exact same software and hardware that they're going to fly with in the labs. And the downlink from the Orion spacecraft is 1,000 times faster than the International Space Station, which means we can send a lot more data back.

Where really big data comes into play in the Orion spacecraft is the development of the vehicle. The downlink is actually pretty small compared to what the test labs can produce. And so when we talk about big data on Orion, we talk about petabytes of data, and one gig networks that are running full time in seven different labs across the country.

Telemetry is a sensor measurement that's typically measured at a remote location. So for the test labs, picture a room with components everywhere wired together on different racks. The test engineers connect all of the wires, and run various scenarios; scenarios on test rigs can run for weeks.

Microsecond Timestamps

These telemetry measurements are *microsecond timestamped*, so this is not your typical time-series data. There are also different time *sources*. The average spacecraft has a “space time” up on the vehicle, and a ground time. With Orion, there are 12 different sources of time, and they’re all computed differently based on different measurements. It’s a highly complex time series, because there’s correlation across all of the different sensors, at different times. And of course, because it’s human flight, it requires a very high degree of fault tolerance.

In the EFT-1, there were about 350,000 measurements possible. On EM-1, which is the next mission, there are *three million different types of measurements*. So it’s a lot of information for the spacecraft engineers to understand and try to consume. They have subsystem engineers that know specific sensor measurements, and they focus on those measurements. Out of the three million measurements, subsystem engineers are only going to be able to focus on a handful of them when they do their analyses—that is where data analytics is needed. We need algorithms that can parse through all of the different sensor measurements.

Identifying Patterns in the Data

NASA has seven labs across the country, and does different tests at each lab. One of the goals as NASA builds this vehicle is to catalog all of the test history and download it into a big data architecture. That way, they can run post-analysis on all the tests, and correlate across multiple tests. The idea is that this post-analysis will allow NASA to see if they can identify different trending activities or patterns, to indicate that the vehicle is being built properly, and whether it will operate properly in space.

The EM-1 is expected to be ready in 2018. It’ll have four times as many computers, and twice as many instruments and subsystems as the spacecraft used in EFT-1. Although they’re not sending a human to space yet, all the subsystems need to be rated for human flight.

Orion and Lockheed Martin are building data analytics organizations, which means that we have technology and platform developers. We also have “ponderers”—people who want to ask questions of the data and want to understand the patterns and abnormalities. In

an organization like this, you also need subject matter experts—people on the programs who understand the different subsystems and components of the subsystems and what they expect to be normal.

Our Goal: A Flexible Analytics Environment

Our goal is to provide NASA with a flexible analytics environment that is able to use different programming languages to analyze telemetry as data is streaming off of test rigs. We use a Lambda architecture. This means we have data ingest coming into a speed layer, where we do stream processing. This is all done in parallel—we apply our analytics on the stream, to identify test failure as soon as possible.

As data comes in, we're persisting raw sensor measurements down to a batch layer, or a persistent object store, where we're able to store all the raw data so that we can go back and reprocess it over and over again. We're also creating real-time views for the data analytics and visualization tools, so that subsystem engineers can analyze data in near real time.

In addition to helping subsystem engineers randomly access data in low latency, we want to enable the data scientists to work with this data after it's been ingested off the rig. We built a system we call MACH-5 Insight™, which allows the data to tell the story about how we're building Orion, and how Orion should behave in normal operating conditions. The key is that we're capturing all of the data—we're indexing it and are processing it at parallel—allowing the data to tell us how the system is performing.

Using Real-Time and Machine Learning

We can also—once we store the data—replay it as if it were live. So we can replay tests over and over from the raw data. The data is also able to be queried across the real-time layer, the speed layer, and the historical layer, so we can do comparisons of the live data coming through the system against the historical datasets.

We're also doing machine learning, using unsupervised and supervised learning to identify anomalies within windows of time. Then, the output of all that processing then gets dumped to HBase, so that we can do random access into all the results coming off the stream. We're starting to use a standard formatted data unit for the space

telemetry data. It's a CCSDS standard for space domain. So anybody running a ground system for space can now push data into our platform, in this format. The interesting piece about this is that this format that we construct on ingest has to be queryable 25 years from now, when NASA comes back to do analysis.

We have a header and a body. A header has metadata. And the body is the payload. So you can do analytics just on the header, without having to explode the whole telemetry measurements. That makes stream processing efficient. We used protocol buffers to take the header and serialize that into an object, and then serialize the payload into the object. That payload in the body is a list of sensor measurements given a time range. And they call that a packet.

The job for MACH-5 Insight™ is to take that packet, take out all the different measurands with measurements within that packet, break it out to individual rows, and propagate that into HBase. Then we use Kafka, which allows us to scale the ingest so that, if we have multiple tasks running, they could all be flowing data into the system. We could be processing individual tests at individual rates based on the infrastructure we allocate to a given test. So it processes all our data for us.

Then, we can do downstream processing in ingest. We use Spark to specifically perform real-time analytics and batch analytics. The same code we write for our stream-processing jobs that do the conversion from SFDU into our internal format, we can do in a batch manner. If we have long-term trending analytics that we need to run across multiple tests or across multiple weeks or even years of data, we could write one Spark job and be able to execute that on all the data. And then, even propagate and share logic into the speed layer where we're doing stream processing.

The other beauty is that Spark is running on YARN. YARN effectively allows you to manage large cluster resources and have multiple components of the Hadoop ecosystem running in the same infrastructure. The Kafka direct connect from Spark is especially exciting. We can pull data directly out of Kafka, and guarantee that data gets processed and ingested. And we can manage the offsets ourselves in our Spark job using the Kafka direct connect.

Analytics Using Stream and Batch Processing

Processing analytics is where it gets really important to NASA, and Orion, and other customers of Lockheed Martin. We need to provide stream processing and batch processing to do things like limit checking on all the measurements coming off of the test rigs. We need to understand the combination of all of these different measurements and how they relate.

When designing a system for human space flight, you need to validate that what you're building meets the requirements of the contract and of the system itself. So what we do when we build the system, and when we do our tests and integration, we go through and run a whole bunch of validation tests to make sure that what the output is producing is according to the specs.

We're working on supervised and unsupervised learning approaches to identifying anomalies in the datasets. And we're seeing some really interesting results. What we ultimately want to do is be able to predict failure before it even happens, when Orion's flying in 2018.

PART III

Ethics of Algorithms in IoT

CHAPTER 9

How Are Your Morals? Ethics in Algorithms and IoT

Majken Sander and Joerg Blumtritt

Editor's Note: At Strata + Hadoop World in Singapore, in December 2015, Majken Sander (Business Analyst at BusinessAnalyst.dk) and Joerg Blumtritt (CEO at Datarella) examined important questions about the transparency of algorithms, including our ability to change or affect the way an algorithm views us.

The codes that make things into smart things are not objective. Algorithms bear value judgments—making decisions on methods, or presets of the program’s parameters; these choices are based on how to deal with tasks according to social, cultural, legal rules, or personal persuasion. These underlying value judgments imposed on users are not visible in most contexts. How can we direct the moral choices in the algorithms that impact the way we live, work, and play?

As data scientists, we know that behind any software that processes data is the raw data that you can see in one way or another. What we are usually not seeing are the hidden value judgments that drive the decisions about what data to show and how—these are judgments that someone made on our behalf.

Here’s an example of the kind of value judgment and algorithms that we will be facing within months, rather than years—self-driving cars. Say you are in a self-driving car, and you are about to be in an accident. You have the choice: will you be hit straight on from a

huge truck, or from the side? You would choose sideways, because you think that will give you the biggest opportunity to survive, right? But what if your child is in the car, and sitting next to you? But how do you tell an algorithm to change the choice because of your values? We *might* be able to figure that out.

One variation in algorithms already being taken into account is that cars will obey the laws in the country in which they're driving. For example, if you buy a self-driving car and bring it to the United Kingdom, it will obey the laws in the United Kingdom, but that same car should adhere to different laws when driving in Germany.

That sounds fairly easy to put in an algorithm, but what about differences in culture and style—how do we put that in the algorithms? How aggressively would you expect a car to merge into the flow of the traffic? Well, that's very different from one country to the next. In fact, it could even be different from the northern part of a country to the southern, so how would you map that?

Beta Representations of Values

Moving beyond liabilities and other obvious topics with the self-driving car, we would like to suggest some solutions that involve taking *beta representations* of our values, and using those to teach the machines that we deal with who we are and what we want.

Actually, that's not too exotic. We have that already. For example, we have ad preferences from Google. Google assigns properties to us so it can target ads (hygiene, toiletry, tools, etc.), but what if I'm a middle-aged man working for an ad agency that has a lingerie company as a client, with a line especially targeted for little girls? Google would see me as weird. Google forces views on us based on the way it looks at us.

What about a female journalist who is writing a story about the use of IoT—and her fridge tells her that, because she's pregnant, she may not drink beer, because pregnant people are not allowed to drink beer. And her grocery store that delivers groceries to her a couple of times each week, suddenly adds orange juice, because everyone knows that pregnant women like orange juice. And her smart TV starts showing her ads for diapers. The problem is, the journalist is not pregnant, but there's nowhere she can go to say, "Hey, you've got

it wrong—I’m not pregnant! Give me my beer and drop the orange juice!”

And that’s the thing that we want to propose—we need these kinds of interfaces to deal with these algorithmic judgments.

Choosing How We Want to Be Represented

There are the three ingredients for doing this. First, we have *training data*—that’s the most important. We have to collect data on how we act, so the machine can learn who we are. Next, we have the *algorithms*—usually some kind of classification and regression algorithm, such as decision trees, neural networks, or nearest neighbor. You could just see who is like me, and then see how people who are like me would have acted, and then extrapolate from that.

And then there’s the third ingredient: the *boundary conditions*—the initial distribution that you would expect, and this is the really the tricky part, because it’s always built into these kinds of probabilistic machines, and it’s the least obvious of the three.

For example, take the self-driving car (again). One of the challenges could be judging your state of mind when you get in the car. One day, you’re going to work and just want to get there fast. Is there an algorithm for it? Then there are days when you want to put the kids in the car and drive around and see trees and buildings and fun places. If you only do that on Sundays, that’s easy for an algorithm to understand, but maybe it’s Thursday and you want this feeling of Sunday.

Some software companies would solve it by having an assistant pop up when you get in your car. You would have to click through 20 questions in a survey interface before your car would start. How are you feeling today? Of course, no one wants to do that. Still, there must be some easy way to suggest different routes and abstractions for that.

There is. It’s already there. It’s just not controllable—by you. You can’t teach Google Maps. It’s the guys in Mountain View that make these decisions for you, and you really don’t see how they did it. But it should not be a nerd thing—it should be easy to train these algorithms ourselves.

Some companies are already experimenting with this idea. Netflix is a very good example. It's the poster child for recommendation engines. There's a very open discussion about how Netflix does it, and the interesting thing is that Netflix is really aware of how important social context is for your decisions. After all, we don't make decisions on our own. Those who are near to us, like family, friends, or neighbors, influence us. Also, society influences our decisions.

If you type in "target group" into Google's image search, you get images that show an anonymous mass of people, and actually, that is how marketing teams tend to see human beings—as a target that you shoot at. The idea of representation is closely tied to a target group, because it gives you a meaningful aggregate—a set of people who could be seen as homogeneous enough to be represented by one specimen. You could do that by saying, well, as a market researcher, I take a sample of 2,000 women, and then I take 200 of them that might be women 20 to 39 years old. That's how we do market research; that's how we do marketing.

We would take these 200 women, build the mean of their properties, and all other women would be generalized as being like them. But this is not really the world we live in. If a recommendation engine, a search engine, or a targeting engine is done well, we don't see people represented as aggregated. We see each one represented as an individual. And we could use that for democracy also.

We have these kind of aggregates also in democracies, in the constituencies. It's a one-size-fits-all, Conservative Party program. It's a one-size-fits-all Labor Party program, or Green Party program. Maybe 150 years ago this would define who you are in terms of the policies you would support. That made sense. But after the 1980s, that changed. We can see it now. We can see that this no longer fits. And our algorithmic representation might be a solution to scale, because you can't scale grassroots democracy.

A grassroots democracy is very demanding. You have everybody always having to decide for every policy that's on the table. That's not feasible. You can't even do that in small villages like we have in Switzerland. Some things have to be decided by a city council, so if we want a nonrepresentative way of doing policies, of doing politics, we could try using algorithmic representation to bulk suggest policies that we would support.

That need not be party programs. It could be very granular—single decisions that we would tick off one by one. There are some downsides, some problems that we have to solve.

For example, these algorithms tend to have a snowball effect for some decisions. We made agent-based simulation models, and that was one of the outcomes. And in general, democracies and societies—even nondemocratic societies—don’t work by just doing majority representation. We know that. We need some kind of minority protection. We need to represent a multitude of opinions in every social system.

Second, there are positive feedback loops. I might see the effect of my voting together with others, and that’s like jumping on the bandwagon. That’s also seen in simulations. It’s very strong. It’s the conforming trap.

And third, your data is always lagging behind. Your data is your past self. How could it represent changes of your opinion? You might think, well, last election, I don’t know, I was an angry, disappointed employee, but now, I’m self-employed, and really self-confident. I might change my views. That would not necessarily be mapped into the data. So these are three things we should be careful about.

The fourth one is that we have to take care of the possibility that the algorithm of me is slightly off. It could be in a trivial way, like what I buy for groceries. It could be my movie preferences. So I have to actually give my “algorithmic me” feedback. I have to adjust it, maybe just a little bit, but I have to be able to deliver the feedback that, in the earlier examples, we were lacking the skills to do.

As users, we actually need to ask questions. Instead of just accepting that Google gives me the wrong product ads compared to my temperament, my fridge orders orange juice that I dislike, or my self-driving car drives in a way that I find annoying, we need to say, hey, the data is there. It’s my data. Ask me for it and I will deliver this, so you can paint the picture of who I really am.

About the Author

Alice LaPlante is an award-winning writer who has been writing about technology, and the business of technology, for more than 20 years. The former news editor of *InfoWorld*, and a contributing editor to *ComputerWorld*, *InformationWeek*, and other national publications, Alice was a Wallace Stegner Fellow at Stanford University and taught writing at Stanford for more than two decades. She is the author of six books, including *Playing for Profit: How Digital Entertainment is Making Big Business Out of Child's Play*.