O'REILLY®

2016 Edition

Big Data Now

Current Perspectives from O'Reilly Media

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

Job # D2044

Big Data Now: 2016 Edition

Current Perspectives from O'Reilly Media

O'Reilly Media, Inc.

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

Big Data Now: 2016 Edition
by O'Reilly Media, Inc.

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://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Nicole Tache
Production Editor: Nicholas Adams
Copyeditor: Gillian McGarvey
Proofreader: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Randy Comer

February 2017: First Edition

Revision History for the First Edition
2017-01-27: First Release

978-1-491-97748-4

[LSI]

Table of Contents

Introduction

Big data pushed the boundaries in 2016. It pushed the boundaries of tools, applications, and skill sets. And it did so because it's bigger, faster, more prevalent, and more prized than ever.

According to O'Reilly's *2016 Data Science Salary Survey*, the top tools used for data science continue to be SQL, Excel, R, and Python. A common theme in recent tool-related blog posts on *oreilly.com* is the need for powerful storage and compute tools that can process high-volume, often streaming, data. For example, Federico Castanedo's blog post "Scalable Data Science with R" describes how scaling R using distributed frameworks—such as RHadoop and SparkR—can help solve the problem of storing massive data sets in RAM.

Focusing on storage, more organizations are looking to migrate their data, and storage and compute operations, from warehouses on proprietary software to managed services in the cloud. There is, and will continue to be, a lot to talk about on this topic: building a data pipeline in the cloud, security and governance of data in the cloud, cluster-monitoring and tuning to optimize resources, and of course, the three providers that dominate this area—namely, Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure.

In terms of techniques, machine learning and deep learning continue to generate buzz in the industry. The algorithms behind natural language processing and image recognition, for example, are incredibly complex, and their utility, in the enterprise hasn't been fully realized. Until recently, machine learning and deep learning have been largely confined to the realm of research and academics. We're now seeing a surge of interest in organizations looking to

apply these techniques to their business use case to achieve automated, actionable insights. Evangelos Simoudis discusses this in his O'Reilly blog post "Insightful applications: The next inflection in big data." Accelerating this trend are open source tools, such as TensorFlow from the Google Brain Team, which put machine learning into the hands of any person or entity who wishes to learn about it.

We continue to see smartphones, sensors, online banking sites, cars, and even toys generating more data, of varied structure. O'Reilly's *Big Data Market report* found that a surprisingly high percentage of organizations' big data budgets are spent on Internet-of-Things-related initiatives. More tools for fast, intelligent processing of real-time data are emerging (Apache Kudu and FiloDB, for example), and organizations across industries are looking to architect robust pipelines for real-time data processing. Which components will allow them to efficiently store and analyze the rapid-fire data? Who will build and manage this technology stack? And, once it is constructed, who will communicate the insights to upper management? These questions highlight another interesting trend we're seeing—the need for cross-pollination of skills among technical and non-technical folks. Engineers are seeking the analytical and communication skills so common in data scientists and business analysts, and data scientists and business analysts are seeking the hard-core technical skills possessed by engineers, programmers, and the like.

Data science continues to be a hot field and continues to attract a range of people—from IT specialists and programmers to business school graduates—looking to rebrand themselves as data science professionals. In this context, we're seeing tools push the boundaries of accessibility, applications push the boundaries of industry, and professionals push the boundaries of their skill sets. In short, data science shows no sign of losing momentum.

In *Big Data Now: 2016 Edition*, we present a collection of some of the top blog posts written for *oreilly.com* in the past year, organized around six key themes:

- Careers in data
- Tools and architecture for big data
- Intelligent real-time applications
- Cloud infrastructure
- Machine learning: models and training

- Deep learning and AI

Let's dive in!

CHAPTER 1

Careers in Data

In this chapter, Michael Li offers five tips for data scientists looking to strengthen their resumes. Jerry Overton seeks to quash the term "unicorn" by discussing five key habits to adopt that develop that magical combination of technical, analytical, and communication skills. Finally, Daniel Tunkelang explores why some employers prefer generalists over specialists when hiring data scientists.

Five Secrets for Writing the Perfect Data Science Resume

By Michael Li

You can read this post on oreilly.com here.

Data scientists are in demand like never before, but nonetheless, getting a job as a data scientist requires a resume that shows off your skills. At The Data Incubator, we've received tens of thousands of resumes from applicants for our free Data Science Fellowship. We work hard to read between the lines to find great candidates who happen to have lackluster CVs, but many recruiters aren't as diligent. Based on our experience, here's the advice we give to our Fellows about how to craft the perfect resume to get hired as a data scientist.

Be brief: A resume is a summary of your accomplishments. It is not the right place to put your Little League participation award. Remember, you are being judged on something a lot closer to the

average of your listed accomplishments than their *sum*. Giving unnecessary information will only dilute your average. Keep your resume to no more than one page. Remember that a busy HR person will scan your resume for about 10 seconds. Adding more content will only distract them from finding key information (as will that second page). That said, don't play font games; keep text at 11-point font or above.

Avoid weasel words: "Weasel words" are subject words that create an impression but can allow their author to "weasel" out of any specific meaning if challenged. For example "talented coder" contains a weasel word. "Contributed 2,000 lines to Apache Spark" can be verified on GitHub. "Strong statistical background" is a string of weasel words. "Statistics PhD from Princeton and top thesis prize from the American Statistical Association" can be verified. Self-assessments of skills are inherently unreliable and untrustworthy; finding others who can corroborate them (like universities, professional associations) makes your claims a lot more believable.

Use metrics: Mike Bloomberg is famous for saying "If you can't measure it, you can't manage it and you can't fix it." He's not the only manager to have adopted this management philosophy, and those who have are all keen to see potential data scientists be able to quantify their accomplishments. "Achieved superior model performance" is weak (and weasel-word-laden). Giving some specific metrics will really help combat that. Consider "Reduced model error by 20% and reduced training time by 50%." Metrics are a powerful way of avoiding weasel words.

Cite specific technologies in context: Getting hired for a technical job requires demonstrating technical skills. Having a list of technologies or programming languages at the top of your resume is a start, but that doesn't give context. Instead, consider weaving those technologies into the narratives about your accomplishments. Continuing with our previous example, consider saying something like this: "Reduced model error by 20% and reduced training time by 50% by using a warm-start regularized regression in scikit-learn." Not only are you specific about your claims but they are also now much more believable because of the specific techniques you're citing. Even better, an employer is much more likely to believe you understand in-demand scikit-learn, because instead of just appearing on a list of technologies, you've spoken about how you used it.

Talk about the data size: For better or worse, big data has become a "mine is bigger than yours" contest. Employers are anxious to see candidates with experience in large data sets—this is not entirely unwarranted, as handling truly "big data" presents unique new challenges that are not present when handling smaller data. Continuing with the previous example, a hiring manager may not have a good understanding of the technical challenges you're facing when doing the analysis. Consider saying something like this: "Reduced model error by 20% and reduced training time by 50% by using a warm-start regularized regression in scikit-learn streaming over 2 TB of data."

While data science is a hot field, it has attracted a lot of newly rebranded data scientists. If you have real experience, set yourself apart from the crowd by writing a concise resume that quantifies your accomplishments with metrics and demonstrates that you can use in-demand tools and apply them to large data sets.

There's Nothing Magical About Learning Data Science

By Jerry Overton

You can read this post on oreilly.com here.

There are people who can imagine ways of using data to improve an enterprise. These people can explain the vision, make it real, and affect change in their organizations. They are—or at least strive to be—as comfortable talking to an executive as they are typing and tinkering with code. We sometimes call them "unicorns" because the combination of skills they have are supposedly mystical, magical… and imaginary.

But I don't think it's unusual to meet someone who wants their work to have a real impact on real people. Nor do I think there is anything magical about learning data science skills. You can pick up the basics of machine learning in about 15 hours of lectures and videos. You can become reasonably good at most things with about 20 hours (45 minutes a day for a month) of focused, deliberate practice.

So basically, being a unicorn, or rather a *professional* data scientist, is something that can be taught. Learning all of the related skills is difficult but straightforward. With help from the folks at O'Reilly, we

designed a tutorial for Strata + Hadoop World New York, 2016, "Data science that works: best practices for designing data-driven improvements, making them real, and driving change in your enterprise," for those who aspire to the skills of a unicorn. The premise of the tutorial is that you can follow a direct path toward professional data science by taking on the following, most distinguishable habits:

Put Aside the Technology Stack

The tools and technologies used in data science are often presented as a technology *stack*. The stack is a problem because it encourages you to to be motivated by *technology*, rather than *business problems*. When you focus on a technology stack, you ask questions like, "Can this tool connect with that tool" or, "What hardware do I need to install this product?" These are important concerns, but they aren't the kinds of things that motivate a professional data scientist.

Professionals in data science tend to think of tools and technologies as part of *an insight utility*, rather than a technology stack (Figure 1-1). Focusing on building a utility forces you to select components based on the insights that the utility is meant to generate. With *utility thinking*, you ask questions like, "What do I need to discover an insight?" and, "Will this technology get me closer to my business goals?"

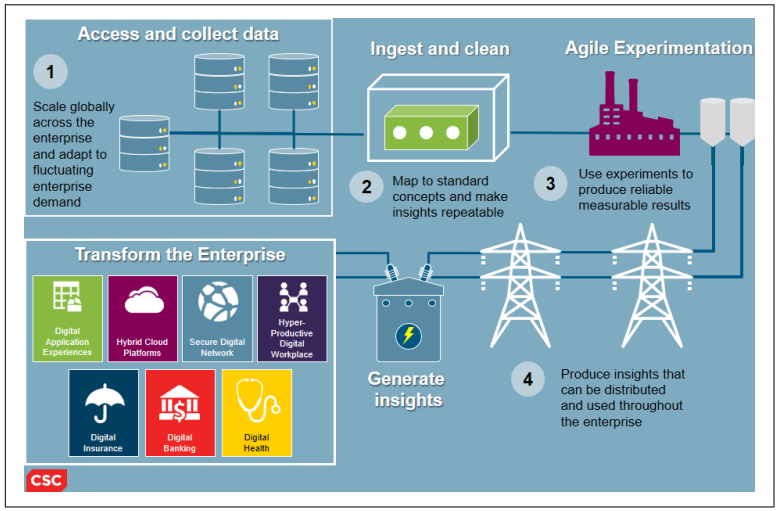

Figure 1-1. Data science tools and technologies as components of an insight utility, rather than a technology stack. Credit: Jerry Overton.

In the Strata + Hadoop World tutorial in New York, I taught simple strategies for shifting from technology-stack thinking to insight-utility thinking.

Keep Data Lying Around

Data science stories are often told in the reverse order from which they happen. In a well-written story, the author starts with an important question, walks you through the data gathered to answer the question, describes the experiments run, and presents resulting conclusions. In real data science, the process usually starts when someone looks at data they already have and asks, "Hey, I wonder if we could be doing something cool with this?" That question leads to tinkering, which leads to building something useful, which leads to the search for someone who might benefit. Most of the work is devoted to bridging the gap between the insight discovered and the stakeholder's needs. But when the story is told, the reader is taken on a smooth progression from stakeholder to insight.

The questions you ask are usually the ones for which you have access to enough data to answer. Real data science usually requires a healthy stockpile of discretionary data. In the tutorial, I taught techniques for building and using data pipelines to make sure you always have enough data to do something useful.

Have a Strategy

Data strategy gets confused with data governance. When I think of strategy, I think of chess. To *play* a game of chess, you have to know the rules. To *win* a game of chess, you have to have a strategy. Knowing that "the D2 pawn can move to D3 unless there is an obstruction at D3 or the move exposes the king to direct attack" is necessary to play the game, but it doesn't help me pick a winning move. What I really need are *patterns* that put me in a better position to win—"If I can get my knight and queen connected in the center of the board, I can force my opponent's king into a trap in the corner."

This lesson from chess applies to *winning with data*. Professional data scientists understand that to win with data, you need a strategy, and to build a strategy, you need a map. In the tutorial, we reviewed ways to build maps from the most important business questions, build data strategies, and execute the strategy using utility thinking (Figure 1-2).

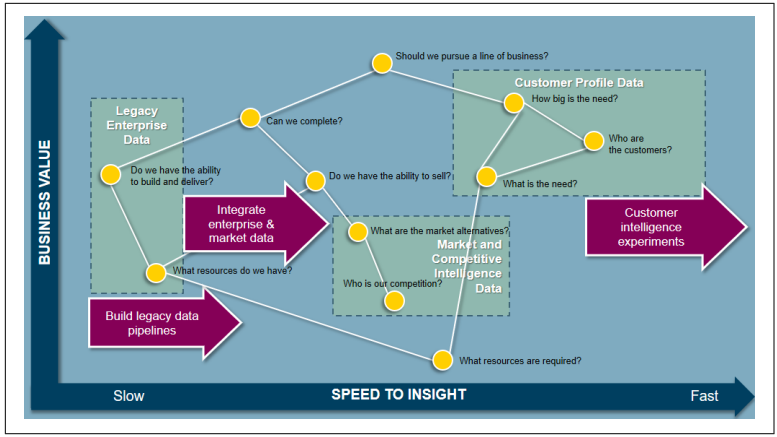

Figure 1-2. A data strategy map. Data strategy is not the same as data governance. To execute a data strategy, you need a map. Credit: Jerry Overton.

Hack

By hacking, of course, I don't mean subversive or illicit activities. I mean cobbling together useful solutions. Professional data scientists constantly need to build things quickly. Tools can make you more productive, but tools alone won't bring your productivity to anywhere near what you'll need.

To operate on the level of a professional data scientist, you have to master the art of the hack. You need to get good at producing new, minimum-viable, data products based on adaptations of assets you already have. In New York, we walked through techniques for hacking together data products and building solutions that you understand and are fit for purpose.

Experiment

I don't mean experimenting as simply trying out different things and seeing what happens. I mean the more formal experimentation as prescribed by the scientific method. Remember those experiments you performed, wrote reports about, and presented in grammar-school science class? It's like that.

Running experiments and evaluating the results is one of the most effective ways of making an impact as a data scientist. I've found that great stories and great graphics are not enough to convince others to

adopt new approaches in the enterprise. The only thing I've found to be consistently powerful enough to affect change is a successful example. Few are willing to try new approaches until they have been proven successful. You can't prove an approach successful unless you get people to try it. The way out of this vicious cycle is to run a series of small experiments (Figure 1-3).

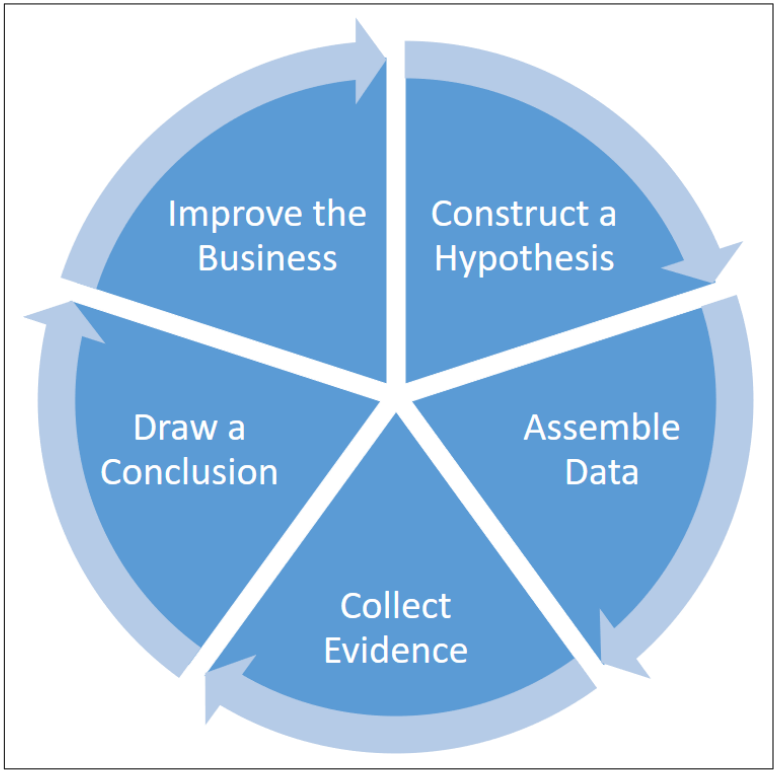

Figure 1-3. Small continuous experimentation is one of the most powerful ways for a data scientist to affect change. Credit: Jerry Overton.

In the tutorial at Strata + Hadoop World New York, we also studied techniques for running experiments in very short sprints, which forces us to focus on discovering insights and making improvements to the enterprise in small, meaningful chunks.

We're at the beginning of a new phase of big data—a phase that has less to do with the technical details of massive data capture and storage and much more to do with producing impactful scalable insights. Organizations that adapt and learn to put data to good use

will consistently outperform their peers. There is a great need for people who can imagine data-driven improvements, make them real, and drive change. I have no idea how many people are actually interested in taking on the challenge, but I'm really looking forward to finding out.

Data Scientists: Generalists or Specialists?

By Daniel Tunkelang

You can read this post on oreilly.com here.

Editor's note: This is the second in a three-part series of posts by Daniel Tunkelang dedicated to data science as a profession. In this series, Tunkelang will cover the recruiting, organization, and essential functions of data science teams.

When LinkedIn posted its first job opening for a "data scientist" in 2008, the company was clearly looking for generalists:

> Be challenged at LinkedIn. We're looking for superb analytical minds of all levels to expand our small team that will build some of the most innovative products at LinkedIn.
>
> No specific technical skills are required (we'll help you learn SQL, Python, and R). You should be extremely intelligent, have quantitative background, and be able to learn quickly and work independently. This is the perfect job for someone who's really smart, driven, and extremely skilled at creatively solving problems. You'll learn statistics, data mining, programming, and product design, but you've gotta start with what we can't teach—intellectual sharpness and creativity.

In contrast, most of today's data scientist jobs require highly specific skills. Some employers require knowledge of a particular programming language or tool set. Others expect a PhD and significant academic background in machine learning and statistics. And many employers prefer candidates with relevant domain experience.

If you are building a team of data scientists, should you hire generalists or specialists? As with most things, it depends. Consider the kinds of problems your company needs to solve, the size of your team, and your access to talent. But, most importantly, consider your company's stage of maturity.

Early Days

Generalists add more value than specialists during a company's early days, since you're building most of your product from scratch, and something is better than nothing. Your first classifier doesn't have to use deep learning to achieve game-changing results. Nor does your first recommender system need to use gradient-boosted decision trees. And a simple t-test will probably serve your A/B testing needs.

Hence, the person building the product doesn't need to have a PhD in statistics or 10 years of experience working with machine-learning algorithms. What's more useful in the early days is someone who can climb around the stack like a monkey and do whatever needs doing, whether it's cleaning data or native mobile-app development.

How do you identify a good generalist? Ideally this is someone who has already worked with data sets that are large enough to have tested his or her skills regarding computation, quality, and heterogeneity. Surely someone with a STEM background, whether through academic or on-the-job training, would be a good candidate. And someone who has demonstrated the ability and willingness to learn how to use tools and apply them appropriately would definitely get my attention. When I evaluate generalists, I ask them to walk me through projects that showcase their breadth.

Later Stage

Generalists hit a wall as your products mature: they're great at developing the first version of a data product, but they don't necessarily know how to improve it. In contrast, machine-learning specialists can replace naive algorithms with better ones and continuously tune their systems. At this stage in a company's growth, specialists help you squeeze additional opportunity from existing systems. If you're a Google or Amazon, those incremental improvements represent phenomenal value.

Similarly, having statistical expertise on staff becomes critical when you are running thousands of simultaneous experiments and worrying about interactions, novelty effects, and attribution. These are first-world problems, but they are precisely the kinds of problems that call for senior statisticians.

How do you identify a good specialist? Look for someone with deep experience in a particular area, like machine learning or experimentation. Not all specialists have advanced degrees, but a relevant academic background is a positive signal of the specialist's depth and commitment to his or her area of expertise. Publications and presentations are also helpful indicators of this. When I evaluate specialists in an area where I have generalist knowledge, I expect them to humble me and teach me something new.

Conclusion

Of course, the ideal data scientist is a strong generalist who also brings unique specialties that complement the rest of the team. But that ideal is a unicorn—or maybe even an alicorn. Even if you are lucky enough to find these rare animals, you'll struggle to keep them engaged in work that is unlikely to exercise their full range of capabilities.

So, should you hire generalists or specialists? It really does depend—and the largest factor in your decision should be your company's stage of maturity. But if you're still unsure, then I suggest you favor generalists, especially if your company is still in a stage of rapid growth. Your problems are probably not as specialized as you think, and hiring generalists reduces your risk. Plus, hiring generalists allows you to give them the opportunity to learn specialized skills on the job. Everybody wins.

CHAPTER 2

Tools and Architecture for Big Data

In this chapter, Evan Chan performs a storage and query cost-analysis on various analytics applications, and describes how Apache Cassandra stacks up in terms of ad hoc, batch, and time-series analysis. Next, Federico Castanedo discusses how using distributed frameworks to scale R can help solve the problem of storing large and ever-growing data sets in RAM. Daniel Whitenack then explains how a new programming language from Google—Go—could help data science teams overcome common obstacles such as integrating data science in an engineering organization. Whitenack also details the many tools, packages, and resources that allow users to perform data cleansing, visualization, and even machine learning in Go. Finally, Nicolas Seyvet and Ignacio Mulas Viela describe how the telecom industry is navigating the current data analytics environment. In their use case, they apply both Kappa architecture and a Bayesian anomaly detection model to a high-volume data stream originating from a cloud monitoring system.

Apache Cassandra for Analytics: A Performance and Storage Analysis

By Evan Chan

You can read this post on oreilly.com here.

This post is about using Apache Cassandra for analytics. Think time series, IoT, data warehousing, writing, and querying large swaths of data—not so much transactions or shopping carts. Users thinking of

Cassandra as an event store and source/sink for machine learning/modeling/classification would also benefit greatly from this post.

Two key questions when considering analytics systems are:

1. How much storage do I need (to buy)?
2. How fast can my questions get answered?

I conducted a performance study, comparing different storage layouts, caching, indexing, filtering, and other options in Cassandra (including FiloDB), plus Apache Parquet, the modern gold standard for analytics storage. All comparisons were done using Spark SQL. More importantly than determining data modeling versus storage format versus row cache or DeflateCompressor, I hope this post gives you a useful framework for predicting storage cost and query speeds for your own applications.

I was initially going to title this post "Cassandra Versus Hadoop," but honestly, this post is not about Hadoop or Parquet at all. Let me get this out of the way, however, because many people, in their evaluations of different technologies, are going to think about one technology stack versus another. Which is better for which use cases? Is it possible to lower total cost of ownership (TCO) by having just one stack for everything? Answering the storage and query cost questions are part of this analysis.

To be transparent, I am the author of FiloDB. While I do have much more vested on one side of this debate, I will focus on the analysis and let you draw your own conclusions. However, I hope you will realize that Cassandra is not just a key-value store; it can be—and is being—used for big data analytics, and it can be very competitive in both query speeds and storage costs.

Wide Spectrum of Storage Costs and Query Speeds

Figure 2-1 summarizes different Cassandra storage options, plus Parquet. Farther to the right denotes higher storage densities, and higher up the chart denotes faster query speeds. In general, you want to see something in the upper-right corner.

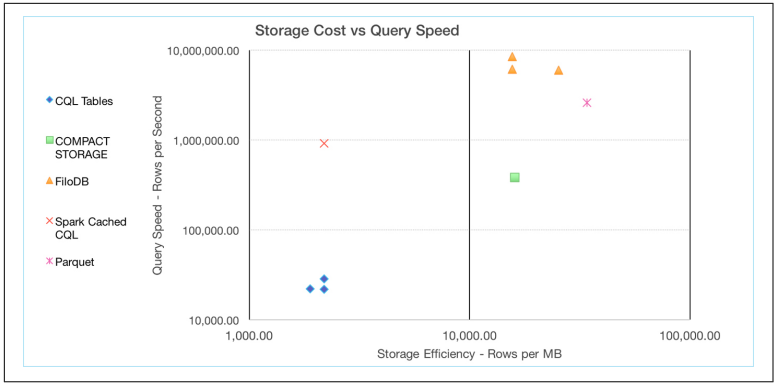

Figure 2-1. Storage costs versus query speed in Cassandra and Parquet. Credit: Evan Chan.

Here is a brief introduction to the different players used in the analysis:

- Regular Cassandra version 2.x CQL tables, in both narrow (one record per partition) and wide (both partition and clustering keys, many records per partition) configurations
- COMPACT STORAGE tables, the way all of us Cassandra old timers did it before CQL (0.6, baby!)
- Caching Cassandra tables in Spark SQL
- FiloDB, an analytical database built on C* and Spark
- Parquet, the reference gold standard

What you see in Figure 2-1 is a wide spectrum of storage efficiency and query speed, from CQL tables at the bottom to FiloDB, which is up to 5x faster in scan speeds than Parquet and almost as efficient storage-wise. Keep in mind that the chart has a log scale on both axes. Also, while this article will go into the tradeoffs and details about different options in depth, we will not be covering the many other factors people choose CQL tables for, such as support for modeling maps, sets, lists, custom types, and many other things.

Summary of Methodology for Analysis

Query speed was computed by averaging the response times for three different queries:

```
df.select(count("numarticles")).show
```

```
SELECT Actor1Name, AVG(AvgTone) as tone FROM gdelt GROUP BY
Actor1Name ORDER BY tone DESC

SELECT AVG(avgtone), MIN(avgtone), MAX(avgtone) FROM gdelt WHERE
monthyear=198012
```

The first query is an all-table-scan simple count. The second query measures a grouping aggregation. And the third query is designed to test filtering performance with a record count of 43.4K items, or roughly 1% of the original data set. The data set used for each query is the GDELT public data set: 1979–1984, 57 columns x 4.16 million rows, recording geopolitical events worldwide. The source code for ingesting the Cassandra tables and instructions for reproducing the queries are available in my cassandra-gdelt repo.

The storage cost for Cassandra tables is computed by running compaction first, then taking the size of all stable files in the data folder of the tables.

To make the Cassandra CQL tables more performant, shorter column names were used (for example, `a2code` instead of `Actor2Code`).

All tests were run on my MacBook Pro 15-inch, mid-2015, SSD/16 GB. Specifics are as follows:

- Cassandra 2.1.6, installed using CCM
- Spark 1.4.0 except where noted, run with master = 'local[1]' and spark.sql.shuffle.partitions=4
- Spark-Cassandra-Connector 1.4.0-M3

Running all the tests essentially single threaded was done partly out of simplicity and partly to form a basis for modeling performance behavior (see "A Formula for Modeling Query Performance" on page 18).

Scan Speeds Are Dominated by Storage Format

OK, let's dive into details! The key to analytics query performance is the *scan speed*, or how many records you can scan per unit time. This is true for whole table scans, and it is true when you filter data, as we'll see later. Figure 2-2 shows the data for all query times, which are whole table scans, with relative speed factors for easier digestion.

Technology	Storage Cost (MB)	Query speed (one col) (approx. time)	Relative	Query speed (top N actor1names by avgtone)	Relative	Filter query speed (avg tones WHERE MonthYear=XXX)	Relative	Rows per MB	Rows per second
Cassandra CQL - Narrow rows	1900	185	1.0	197	1.0	186	1.0	2184.21	21919.01
Cass CQL Narrow + RowCache	1900	145	1.3	149	1.3	144	1.3	2184.21	28424.66
Cassandra CQL Tables - Wide rows / LZ4	2200	278	0.7	281	0.7	2.7	68.9	1886.36	22164.86
Cassandra CQL Wide + Deflate	1100	234	0.8	233	0.8	3.7	50.3	3772.73	26449.97
Cassandra COMPACT STORAGE	260	16	11.6	16	12.3	0.22	845.5	15961.54	386405.96
Cassandra CQL + Spark CacheTable	1900	4.2	44.0	5	39.4	4.4	42.3	2184.21	915441.18
FiloDB + C*LZ4	266	0.45	411.1	1.5	131.3	0.08	2325.0	15601.50	6133004.93
FiloDB + C*Deflate	164	0.5	370.0	1.5	131.3	0.08	2325.0	25304.88	5985576.92
FiloDB InMemoryColumnStore	266	0.24	770.8	1.2	164.2	0.03	6200.0	15601.50	8469387.76
Parquet + GZIP	122	1.2	154.2	2.4	82.1	1.2	155.0	34016.39	2593750.00

Figure 2-2. All query times with relative speed factors. All query times run on Spark 1.4/1.5 with local[1]; C 2.1.6 with 512 MB row cache. Credit: Evan Chan.*

NOTE

To get more accurate scan speeds, one needs to subtract the baseline latency in Spark, but this is left out for simplicity. This actually slightly disfavors the fastest contestants.

Cassandra's COMPACT STORAGE gains an order-of-magnitude improvement in scan speeds simply due to more efficient storage. FiloDB and Parquet gain another order of magnitude due to a columnar layout, which allows reading only the columns needed for analysis, plus more efficient columnar blob compression. Thus, storage format makes the biggest difference in scan speeds. More details follow, but for regular CQL tables, the scan speed should be inversely proportional to the number of columns in each record, assuming simple data types (not collections).

Part of the speed advantage of FiloDB over Parquet has to do with the InMemory option. You could argue this is not fair; however, when you read Parquet files repeatedly, most of that file is most likely in the OS cache anyway. Yes, having in-memory data is a bigger advantage for networked reads from Cassandra, but I think part of the speed increase is because FiloDB's columnar format is optimized more for CPU efficiency, rather than compact size. Also, when you cache Parquet files, you are caching an entire file or blocks thereof, compressed and encoded; FiloDB relies on small chunks, which can be much more efficiently cached (on a per-column basis, and allows for updates). Folks at Databricks have repeatedly told me that caching Parquet files in-memory did not result in significant speed gains, and this makes sense due to the format and compression.

Wide-row CQL tables are actually less efficient than narrow-row due to additional overhead of clustering column-name prefixing. Spark's cacheTable should be nearly as efficient as the other fast solutions but suffers from partitioning issues.

Storage Efficiency Generally Correlates with Scan Speed

In Figure 2-2, you can see that these technologies list in the same order for storage efficiency as for scan speeds, and that's not an accident. Storing tables as COMPACT STORAGE and FiloDB yields a roughly 7–8.5x improvement in storage efficiency over regular CQL tables for this data set. Less I/O = faster scans!

Cassandra CQL wide-row tables are less efficient, and you'll see why in a minute. Moving from LZ4 to Deflate compression reduces storage footprint by 38% for FiloDB and 50% for the wide-row CQL tables, so it's definitely worth considering. DeflateCompressor actually sped up wide-row CQL scans by 15%, but slowed down the single partition query slightly.

Why Cassandra CQL tables are inefficient

Let's say a Cassandra CQL table has a primary key that looks like (pk, ck1, ck2, ck3) and other columns designated c1, c2, c3, c4 for creativity. This is what the physical layout looks like for one partition ("physical row"):

Column header	ck1:ck2:ck3a:c1	ck1:ck2:ck3a:c2	ck1:ck2:ck3a:c3	ck1:ck2:ck3a:c4
pk : value	v1	v2	v3	v4

Cassandra offers ultimate flexibility in terms of updating any part of a record, as well as inserting into collections, but the price paid is that each column of every record is stored in its own cell, with a very lengthy column header consisting of the entire clustering key, plus the name of each column. If you have 100 columns in your table (very common for data warehouse fact tables), then the clustering key *ck1:ck2:ck3* is repeated 100 times. It is true that compression helps a lot with this, but not enough. Cassandra 3.x has a new, trimmer storage engine that does away with many of these inefficiencies, at a reported space savings of up to 4x.

COMPACT STORAGE is the way that most of us who used Cassandra prior to CQL stored our data: as one blob per record. It is extremely efficient. That model looks like this:

Column header	ck1:ck2:ck3	ck1:ck2:ck3a
pk	value1_blob	value2_blob

You lose features such as secondary indexing, but you can still model your data for efficient lookups by partition key and range scans of clustering keys.

FiloDB, on the other hand, stores data by grouping columns together, and then by clumping data from many rows into its own efficient blob format. The layout looks like this:

	Column 1		Column 2	
pk	Chunk 1	Chunk 2	Chunk 1	Chunk 2

Columnar formats minimize I/O for analytical queries, which select a small subset of the original data. They also tend to remain compact, even in-memory. FiloDB's internal format is designed for fast random access without the need to deserialize. On the other hand, Parquet is designed for very fast linear scans, but most encoding types require the entire page of data to be deserialized—thus, filtering will incur higher I/O costs.

A Formula for Modeling Query Performance

We can model the query time for a single query using a simple formula:

> Predicted queryTime = Expected number of records / (# cores * scan speed)

Basically, the query time is proportional to how much data you are querying, and inversely proportional to your resources and raw scan speed. Note that the scan speed previously mentioned is single-core scan speed, such as was measured using my benchmarking methodology. Keep this model in mind when thinking about storage formats, data modeling, filtering, and other effects.

Can Caching Help? A Little Bit.

If storage size leads partially to slow scan speeds, what about taking advantage of caching options to reduce I/O? Great idea. Let's review the different options.

- Cassandra row cache: I tried row cache of 512 MB for the narrow CQL table use case—512 MB was picked as it was a quarter of the size of the data set on disk. Most of the time, your data won't fit in cache. This increased scan speed for the narrow CQL table by 29%. If you tend to access data at the beginning of your partitions, row cache could be a huge win. What I like best about this option is that it's really easy to use and ridiculously simple, and it works with your changing data.
- DSE has an in-memory tables feature. Think of it, basically, as keeping your SSTables in-memory instead of on disk. It seems to me to be slower than row cache (since you still have to

decompress the tables), and I've been told it's not useful for most people.
- Finally, in Spark SQL you can cache your tables (CACHE TABLE in spark-sql, sqlContext.cacheTable in spark-shell) in an on-heap, in-memory columnar format. It is really fast (44x speedup over base case above), but suffers from multiple problems: the entire table has to be cached, it cannot be updated, and it is not high availability (if any executor or the app dies, kaboom!). Furthermore, you have to decide what to cache, and the initial read from Cassandra is still really slow.

None of these options is anywhere close to the wins that better storage format and effective data modeling will give you. As my analysis shows, FiloDB, without caching, is faster than all Cassandra caching options. Of course, if you are loading data from different data centers or constantly doing network shuffles, then caching can be a big boost, but most Spark on Cassandra setups are collocated.

The Future: Optimizing for CPU, Not I/O

For Spark queries over regular Cassandra tables, I/O dominates CPU due to the storage format. This is why the storage format makes such a big difference, and also why technologies like SSD have dramatically boosted Cassandra performance. Due to the dominance of I/O costs over CPU, it may be worth it to compress data more. For formats like Parquet and FiloDB, which are already optimized for fast scans and minimized I/O, it is the opposite—the CPU cost of querying data actually dominates over I/O. That's why the Spark folks are working on code-gen and Project Tungsten.

If you look at the latest trends, memory is getting cheaper; NVRAM, 3DRAM, and very cheap, persistent DRAM technologies promise to make I/O bandwidth no longer an issue. This trend obliterates decades of database design based on the assumption that I/O is much, much slower than CPU, and instead favors CPU-efficient storage formats. With the increase in IOPs, optimizing for linear reads is no longer quite as important.

Filtering and Data Modeling

Remember our formula for predicting query performance:

> Predicted queryTime = Expected number of records / (# cores * scan speed)

Correct data modeling in Cassandra deals with the first part of that equation—enabling fast lookups by reducing the number of records that need to be looked up. Denormalization, writing summaries instead of raw data, and being smart about data modeling all help reduce the number of records. Partition- and clustering-key filtering are definitely the most effective filtering mechanisms in Cassandra. Keep in mind, though, that scan speeds are still really important, even for filtered data—unless you are really only doing single-key lookups.

Look back at Figure 2-2. What do you see? Using partition-key filtering on wide-row CQL tables proved very effective—100x faster than scanning the whole wide-row table on 1% of the data (a direct plugin in the formula of reducing the number of records to 1% of original). However, since wide rows are a bit inefficient compared to narrow tables, some speed is lost. You can also see in Figure 2-2 that scan speeds still matter. FiloDB's in-memory execution of that same filtered query was still 100x faster than the Cassandra CQL table version—taking only 30 *milliseconds* as opposed to nearly three seconds. Will this matter? For serving concurrent, web-speed queries, it will certainly matter.

Note that I only modeled a very simple equals predicate, but in reality, many people need much more flexible predicate patterns. Due to the restrictive predicates available for partition keys (= only for all columns except last one, which can be IN), modeling with regular CQL tables will probably require multiple tables, one each to match different predicate patterns (this is being addressed in C* version 2.2 a bit, maybe more in version 3.x). This needs to be accounted for in the storage cost and TOC analysis. One way around this is to store custom index tables, which allows application-side custom scan patterns. FiloDB uses this technique to provide arbitrary filtering of partition keys.

Some notes on the filtering and data modeling aspect of my analysis:

- The narrow rows layout in CQL is one record per partition key, thus partition-key filtering does not apply. See discussion of secondary indices in the following section.
- Cached tables in Spark SQL, as of Spark version 1.5, only does whole table scans. There might be some improvements coming, though—see SPARK-4849 in Spark version 1.6.

- FiloDB has roughly the same filtering capabilities as Cassandra —by partition key and clustering key—but improvements to the partition-key filtering capabilities of C are planned.
- It is possible to partition your Parquet files and selectively read them, and it is supposedly possible to sort your files to take advantage of intra-file filtering. That takes extra effort, and since I haven't heard of anyone doing the intra-file sort, I deemed it outside the scope of this study. Even if you were to do this, the filtering would not be anywhere near as granular as is possible with Cassandra and FiloDB—of course, your comments and enlightenment are welcome here.

Cassandra's Secondary Indices Usually Not Worth It

How do secondary indices in Cassandra perform? Let's test that with two count queries with a WHERE clause on Actor1CountryCode, a low cardinality field with a hugely varying number of records in our portion of the GDELT data set:

- WHERE Actor1CountryCode = 'USA': 378k records (9.1% of records)
- WHERE Actor1CountryCode = 'ALB': 5,005 records (0.1% of records)

	Large country	Small country	2i scan rate
Narrow CQL table	28s / 6.6x	0.7s / 264x	13.5k records/sec
CQL wide rows	143s / 1.9x	2.7s / 103x	2,643 records/sec

If secondary indices were perfectly efficient, one would expect query times to reduce linearly with the drop in the number of records. Alas, this is not so. For the CountryCode = USA query, one would expect a speedup of around 11x, but secondary indices proved very inefficient, especially in the wide-rows case. Why is that? Because for wide rows, Cassandra has to do a lot of point lookups on the same partition, which is very inefficient and results in only a small drop in the I/O required (in fact, much more random I/O), compared to a full table scan.

Secondary indices work well only when the number of records is reduced to such a small amount that the inefficiencies do not matter and Cassandra can skip most partitions. There are also other opera-

tional issues with secondary indices, and they are not recommended for use when the cardinality goes above 50,000 items or so.

Predicting Your Own Data's Query Performance

How should you measure the performance of your own data and hardware? It's really simple, actually:

1. Measure your scan speed for your base Cassandra CQL table. Number of records/time to query, single-threaded.
2. Use the formula given earlier—Predicted queryTime = Expected number of records/(# cores * scan speed).
3. Use relative speed factors for predictions.

The relative factors in the preceding table are based on the GDELT data set with 57 columns. The more columns you have (data warehousing applications commonly have hundreds of columns), the greater you can expect the scan speed boost for FiloDB and Parquet. (Again, this is because, unlike for regular CQL/row-oriented layouts, columnar layouts are generally insensitive to the number of columns.) It is true that concurrency (within a single query) leads to its own inefficiencies, but in my experience, that is more like a 2x slowdown, and not the order-of-magnitude differences we are modeling here.

User concurrency can be modeled by dividing the number of available cores by the number of users. You can easily see that in FAIR scheduling mode, Spark will actually schedule multiple queries at the same time (but be sure to modify *fair-scheduler.xml* appropriately). Thus, the formula becomes:

> Predicted queryTime = Expected number of records * # users / (# cores * scan speed)

There is an important case where the formula needs to be modified, and that is for single-partition queries (for example, where you have a WHERE clause with an exact match for all partition keys, and Spark pushes down the predicate to Cassandra). The formula assumes that the queries are spread over the number of nodes you have, but this is not true for single-partition queries. In that case, there are two possibilities:

1. The number of users is less than the number of available cores. Then, the query time = number_of_records/scan_speed.

2. The number of users is >= the number of available cores. In that case, the work is divided amongst each core, so the original query time formula works again.

Conclusions

Apache Cassandra is one of the most widely used, proven, and robust distributed databases in the modern big data era. The good news is that there are multiple options for using it in an efficient manner for ad hoc, batch, time-series analytics applications.

For (multiple) order-of-magnitude improvements in query and storage performance, consider the storage format carefully, and model your data to take advantage of partition and clustering key filtering/predicate pushdowns. Both effects can be combined for maximum advantage—using FiloDB plus filtering data improved a three-minute CQL table scan to response times less than 100 ms. Secondary indices are helpful only if they filter your data down to, say, 1% or less—and even then, consider them carefully. Row caching, compression, and other options offer smaller advantages up to about 2x.

If you need a lot of individual record updates or lookups by individual record but don't mind creating your own blob format, the COMPACT STORAGE/single column approach could work really well. If you need fast analytical query speeds with updates, fine-grained filtering and a web-speed in-memory option, FiloDB could be a good bet. If the formula previously given shows that regular Cassandra tables, laid out with the best data-modeling techniques applied, are good enough for your use case, kudos to you!

Scalable Data Science with R

By Federico Castanedo

You can read this post on oreilly.com here.

R is among the top five data-science tools in use today, according to O'Reilly research; the latest KDnuggets survey puts it in first; and IEEE Spectrum ranks it as the fifth most popular programming language.

The latest Rexer Data Science Survey revealed that in the past eight years, there has been an three-fold increase in the number of

respondents using R, and a seven-fold increase in the number of analysts/scientists who have said that R is their primary tool.

Despite its popularity, the main drawback of vanilla R is its inherently "*single-threaded*" nature and its need to fit all the data being processed in RAM. But nowadays, data sets are typically in the range of GBs, and they are growing quickly to TBs. In short, current growth in data volume and variety is demanding more efficient tools by data scientists.

Every data-science analysis starts with preparing, cleaning, and transforming the raw input data into some tabular data that can be further used in machine-learning models.

In the particular case of R, data size problems usually arise when the input data do not fit in the RAM of the machine and when data analysis takes a long time because parallelism does not happen automatically. Without making the data smaller (through sampling, for example), this problem can be solved in two different ways:

1. Scaling-out vertically, by using a machine with more available RAM. For some data scientists leveraging cloud environments like AWS, this can be as easy as changing the instance type of the machine (for example, AWS recently provided an instance with 2 TB of RAM). However, most companies today are using their internal data infrastructure that relies on commodity hardware to analyze data—they'll have more difficulty increasing their available RAM.
2. Scaling-out horizontally: in this context, it is necessary to change the default R behavior of loading all required data in memory and access the data differently by using a distributed or parallel schema with a divide-and-conquer (or in R terms, split-apply-combine) approach like MapReduce.

While the first approach is obvious and can use the same code to deal with different data sizes, it can only scale to the memory limits of the machine being used. The second approach, by contrast, is more powerful, but it is also more difficult to set up and adapt to existing legacy code.

There is a third approach. Scaling-out horizontally can be solved by using R as an interface to the most popular distributed paradigms:

- Hadoop: through using the set of libraries or packages known as RHadoop. These R packages allow users to analyze data with Hadoop through R code. They consist of *rhdfs* to interact with HDFS systems; *rhbase* to connect with HBase; *plyrmr* to perform common data transformation operations over large data sets; *rmr2* that provides a map-reduce API; and *ravro* that writes and reads avro files.
- Spark: with SparkR, it is possible to use Spark's distributed computation engine to enable large-scale data analysis from the R shell. It provides a distributed data frame implementation that supports operations like selection, filtering, and aggregation. on large data sets.
- Programming with Big Data in R: (pbdR) is based on MPI and can be used on high-performance computing (HPC) systems, providing a true parallel programming environment in R.

Novel distributed platforms also combine batch and stream processing, providing a SQL-like expression language—for instance, Apache Flink. There are also higher levels of abstraction that allow you to create a data processing language, such as the recently open-sourced project Apache Beam from Google. However, these novel projects are still under development, and so far do not include R support.

After the data preparation step, the next common data science phase consists of training machine-learning models, which can also be performed on a single machine or distributed among different machines. In the case of distributed machine-learning frameworks, the most popular approaches using R, are the following:

- Spark MLlib: through SparkR, some of the machine-learning functionalities of Spark are exported in the R package. In particular, the following machine-learning models are supported from R: generalized linear model (GLM), survival regression, naive Bayes, and k-means.
- H2O framework: a Java-based framework that allows building scalable machine-learning models in R or Python. It can run as standalone platform or with an existing Hadoop or Spark implementation. It provides a variety of supervised learning models, such as GLM, gradient boosting machine (GBM), deep learning, Distributed Random Forest, naive Bayes, and unsupervised learning implementations like PCA and k-means.

Sidestepping the coding and customization issues of these approaches, you can seek out a commercial solution that uses R to access data on the frontend but uses its own big-data-native processing under the hood:

- Teradata Aster R is a massively parallel processing (MPP) analytic solution that facilitates the data preparation and modeling steps in a scalable way using R. It supports a variety of data sources (text, numerical, time series, graphs) and provides an R interface to Aster's data science library that scales by using a distributed/parallel environment, avoiding the technical complexities to the user. Teradata also has a partnership with Revolution Analytics (now Microsoft R) where users can execute R code inside of Teradata's platform.
- HP Vertica is similar to Aster, but it provides On-Line Analytical Processing (OLAP) optimized for large fact tables, whereas Teradata provides On-Line Transaction Processing (OLTP) or OLAP that can handle big volumes of data. To scale out R applications, HP Vertica relies on the open source project Distributed R.
- Oracle also includes an R interface in its advanced analytics solution, known as Oracle R Advanced Analytics for Hadoop (ORAAH), and it provides an interface to interact with HDFS and access to Spark MLlib algorithms.

Teradata has also released an open source package in CRAN called toaster that allows users to compute, analyze, and visualize data with (on top of) the Teradata Aster database. It allows computing data in Aster by taking advantage of Aster distributed and parallel engines, and then creates visualizations of the results directly in R. For example, it allows users to execute K-Means or run several cross-validation iterations of a linear regression model in parallel.

Also related is MADlib, an open source library for scalable in-database analytics currently in incubator at Apache. There are other open source CRAN packages to deal with big data, such as biglm, bigpca, biganalytics, bigmemory, or pbdR—but they are focused on specific issues rather than addressing the data science pipeline in general.

Big data analysis presents a lot of opportunities to extract hidden patterns when you are using the right algorithms and the underlying technology that will help to gather insights. Connecting new scales

of data with familiar tools is a challenge, but tools like Aster R offer a way to combine the beauty and elegance of the R language within a distributed environment to allow processing data at scale.

This post was a collaboration between O'Reilly Media and Teradata. View our statement of editorial independence.

Data Science Gophers

By Daniel Whitenack

You can read this post on oreilly.com here.

If you follow the data science community, you have very likely seen something like "language wars" unfold between Python and R users. They seem to be the only choices. But there might be a somewhat surprising third option: Go, the open source programming language created at Google.

In this post, we are going to explore how the unique features of Go, along with the mindset of Go programmers, could help data scientists overcome common struggles. We are also going to peek into the world of Go-based data science to see what tools are available, and how an ever-growing group of data science gophers are already solving real-world data science problems with Go.

Go, a Cure for Common Data Science Pains

Data scientists are already working in Python and R. These languages are undoubtedly producing value, and it's not necessary to rehearse their virtues here, but looking at the community of data scientists as a whole, certain struggles seem to surface quite frequently. The following pains commonly emerge as obstacles for data science teams working to provide value to a business:

1. **Difficulties building "production-ready" applications or services:** Unfortunately, the very process of interactively exploring data and developing code in notebooks, along with the dynamically typed, single-threaded languages commonly used in data science, cause data scientists to produce code that is almost impossible to productionize. There could be a huge amount of effort in transitioning a model off of a data scientist's laptop into an application that could actually be deployed, handle errors, be tested, and log properly. This barrier of effort often causes data

scientists' models to stay on their laptops or, possibly worse, be deployed to production without proper monitoring, testing, etc. Jeff Magnussen at Stitchfix and Robert Chang at Twitter have each discussed these sorts of cases.

2. **Applications or services that don't behave as expected:** Dynamic typing and convenient parsing functionality can be wonderful, but these features of languages like Python or R can turn their back on you in a hurry. Without a great deal of forethought into testing and edge cases, you can end up in a situation where your data science application is behaving in a way you did not expect and cannot explain (e.g., because the behavior is caused by errors that were unexpected and unhandled). This is dangerous for data science applications whose main purpose is to provide actionable insights within an organization. As soon as a data science application breaks down without explanation, people won't trust it and thus will cease making data-driven decisions based on insights from the application. The Cookiecutter Data Science project is one notable effort at a "logical, reasonably standardized but flexible project structure for doing and sharing data science work" in Python—but the static typing and nudges toward clarity of Go make these workflows more likely.
3. **An inability to integrate data science development into an engineering organization:** Often, data engineers, DevOps engineers, and others view data science development as a mysterious process that produces inefficient, unscalable, and hard-to-support applications. Thus, data science can produce what Josh Wills at Slack calls an "infinite loop of sadness" within an engineering organization.

Now, if we look at Go as a potential language for data science, we can see that, for many use cases, it alleviates these struggles:

1. Go has a proven track record in production, with widespread adoption by DevOps engineers, as evidenced by game-changing tools like Docker, Kubernetes, and Consul being developed in Go. Go is just plain simple to deploy (via static binaries), and it allows developers to produce readable, efficient applications that fit within a modern microservices architecture. In contrast, heavyweight Python data science applications may need readability-killing packages like Twisted to fit into modern event-driven systems and will likely rely on an ecosystem of

tooling that takes significant effort to deploy. Go itself also provides amazing tooling for testing, formatting, vetting, and linting (gofmt, go vet, etc.) that can easily be integrated in your workflow (see here for a starter guide with Vim). Combined, these features can help data scientists and engineers spend most of their time building interesting applications and services, without a huge barrier to deployment.

2. Next, regarding expected behavior (especially with unexpected input) and errors, Go certainly takes a different approach, compared to Python and R. Go code uses error values to indicate an abnormal state, and the language's design and conventions encourage you to explicitly check for errors where they occur. Some might take this as a negative (as it can introduce some verbosity and a different way of thinking). But for those using Go for data science work, handling errors in an idiomatic Go manner produces rock-solid applications with predictable behavior. Because Go is statically typed and because the Go community encourages and teaches handling errors gracefully, data scientists exploiting these features can have confidence in the applications and services they deploy. They can be sure that integrity is maintained over time, and they can be sure that, when something does behave in an unexpected way, there will be errors, logs, or other information helping them understand the issue. In the world of Python or R, errors may hide themselves behind convenience. For example, Python pandas will return a maximum value or a merged dataframe to you, even when the underlying data experiences a profound change (e.g., 99% of values are suddenly null, or the type of a column used for indexing is unexpectedly inferred as float). The point is not that there is no way to deal with issues (as readers will surely know). The point is that there seem to be a million of these ways to shoot yourself in the foot when the language does not force you to deal with errors or edge cases.
3. Finally, engineers and DevOps developers already love Go. This is evidenced by the growing number of small and even large companies developing the bulk of their technology stack in Go. Go allows them to build easily deployable and maintainable services (see points 1 and 2 in this list) that can also be highly concurrent and scalable (important in modern microservices environments). By working in Go, data scientists can be unified with their engineering organization and produce data-driven

applications that fit right in with the rest of their company's architecture.

Note a few things here. The point is not that Go is perfect for every scenario imaginable, so data scientists should use Go, or that Go is fast and scalable (which it is), so data scientists should use Go. The point is that Go can help data scientists produce deliverables that are actually useful in an organization and that they will be able to support. Moreover, data scientists really should love Go, as it alleviates their main struggles while still providing them the tooling to be productive, as we will see next (with the added benefits of efficiency, scalability, and low memory usage).

The Go Data Science Ecosystem

OK, you might buy into the fact that Go is adored by engineers for its clarity, ease of deployment, low memory use, and scalability, but can people actually do data science with Go? Are there things like pandas, numpy, etc. in Go? What if I want to train a model—can I do that with Go?

Yes, yes, and yes! In fact, there are already a great number of open source tools, packages, and resources for doing data science in Go, and communities and organization such as the high energy physics community and The Coral Project are actively using Go for data science. I will highlight some of this tooling shortly (and a more complete list can be found here). However, before I do that, let's take a minute to think about what sort of tooling we actually need to be productive as data scientists.

Contrary to popular belief, and as evidenced by polls and experience (see here and here, for example), data scientists spend most of their time (around 90%) gathering data, organizing data, parsing values, and doing a lot of basic arithmetic and statistics. Sure, they get to train a machine-learning model on occasion, but there are a huge number of business problems that can be solved via some data gathering/organization/cleaning and aggregation/statistics. Thus, in order to be productive in Go, data scientists must be able to gather data, organize data, parse values, and do arithmetic and statistics.

Also, keep in mind that, as gophers, we want to produce clear code over being clever (a feature that also helps us as scientists or data scientists/engineers) and introduce a little copying rather than a little dependency. In some cases, writing a for loop may be preferable

over importing a package just for one function. You might want to write your own function for a chi-squared measure of distance metric (or just copy that function into your code) rather than pulling in a whole package for one of those things. This philosophy can greatly improve readability and give your colleagues a clear picture of what you are doing.

Nevertheless, there are occasions where importing a well-understood and well-maintained package saves considerable effort without unnecessarily reducing clarity. The following provides something of a "state of the ecosystem" for common data science/analytics activities. See here for a more complete list of active/maintained Go data science tools, packages, libraries, etc.

Data Gathering, Organization, and Parsing

Thankfully, Go has already proven itself useful at data gathering and organization, as evidenced by the number and variety of databases and datastores written in Go, including InfluxDB, Cayley, LedisDB, Tile38, Minio, Rend, and CockroachDB. Go also has libraries or APIs for all of the commonly used datastores (Mongo, Postgres, etc.).

However, regarding parsing and cleaning data, you might be surprised to find out that Go also has a lot to offer here as well. To highlight just a few:

- GJSON—quick parsing of JSON values
- ffjson—fast JSON serialization
- gota—data frames
- csvutil—registering a CSV file as a table and running SQL statements on the CSV file
- scrape—web scraping
- go-freeling—NLP

Arithmetic and Statistics

This is an area where Go has greatly improved over the last couple of years. The Gonum organization provides numerical functionality that can power a great number of common data-science-related computations. There is even a proposal to add multidimensional slices to the language itself. In general, the Go community is produc-

ing some great projects related to arithmetic, data analysis, and statistics. Here are just a few:

- math—stdlib math functionality
- gonum/matrix—matrices and matrix operations
- gonum/floats—various helper functions for dealing with slices of floats
- gonum/stats—statistics including covariance, PCA, ROC, etc.
- gonum/graph or gograph—graph data structure and algorithms
- gonum/optimize—function optimizations, minimization

Exploratory Analysis and Visualization

Go is a compiled language, so you can't do exploratory data analysis, right? Wrong. In fact, you don't have to abandon certain things you hold dear like Jupyter when working with Go. Check out these projects:

- gophernotes—Go kernel for Jupyter notebooks
- dashing-go—dashboarding
- gonum/plot—plotting

In addition to this, it is worth noting that Go fits in so well with web development that powering visualizations or web apps (e.g., utilizing D3) via custom APIs, etc. can be extremely successful.

Machine Learning

Even though the preceding tooling makes data scientists productive about 90% of the time, data scientists still need to be able to do some machine learning (and let's face it, machine learning is awesome!). So when/if you need to scratch that itch, Go does not disappoint:

- sajari/regression—multivariable regression
- goml, golearn, and hector—general-purpose machine learning
- bayesian—Bayesian classification, TF-IDF
- sajari/word2vec—word2vec
- go-neural, GoNN, and Neurgo—neural networks

And, of course, you can integrate with any number of machine-learning frameworks and APIs (such as H2O or IBM Watson) to enable a whole host of machine-learning functionality. There is also a Go API for Tensorflow in the works.

Get Started with Go for Data Science

The Go community is extremely welcoming and helpful, so if you are curious about developing a data science application or service in Go, or if you just want to experiment with data science using Go, make sure you get plugged into community events and discussions. The easiest place to start is on Gophers Slack, the golang-nuts mailing list (focused generally on Go), or the gopherds mailing list (focused more specifically on data science). The #data-science channel is extremely active and welcoming, so be sure to introduce yourself, ask questions, and get involved. Many larger cities have Go meetups as well.

Thanks to Sebastien Binet for providing feedback on this post.

Applying the Kappa Architecture to the Telco Industry

By Nicolas Seyvet and Ignacio Mulas Viela

You can read this post on oreilly.com here.

Ever-growing volumes of data, shorter time constraints, and an increasing need for accuracy are defining the new analytics environment. In the telecom industry, traditional user and network data coexists with machine-to-machine (M2M) traffic, media data, social activities, and so on. In terms of volume, this can be referred to as an "explosion" of data. This is a great business opportunity for telco operators and a key angle to take full advantage of current infrastructure investments (4G, LTE).

In this blog post, we will describe an approach to quickly ingest and analyze large volumes of streaming data, the *Kappa architecture*, as well as how to build a Bayesian online-learning model to detect novelties in a complex environment. Note that novelty does not necessarily imply an undesired situation; it indicates a change from previously known behaviors.

We apply both Kappa and the Bayesian model to a use case using a data stream originating from a telco cloud-monitoring system. The stream is composed of telemetry and log events. It is high volume, as many physical servers and virtual machines are monitored simultaneously.

The proposed method quickly detects anomalies with high accuracy while adapting (learning) over time to new system normals, making it a desirable tool for considerably reducing maintenance costs associated with the operability of large computing infrastructures.

What Is Kappa Architecture?

In a 2014 blog post, Jay Kreps accurately coined the term *Kappa architecture* by pointing out the pitfalls of the Lambda architecture and proposing a potential software evolution. To understand the differences between the two, let's first observe what the Lambda architecture looks like, shown in Figure 2-3.

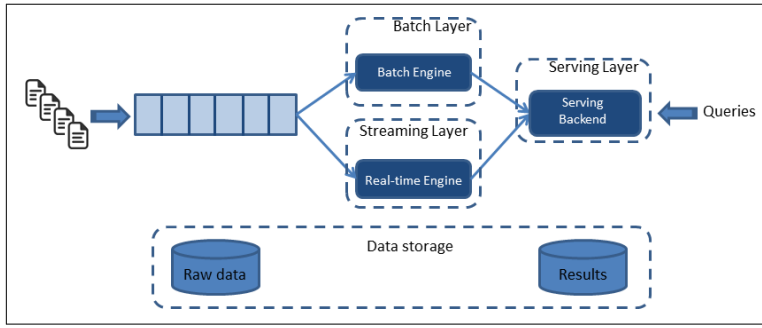

Figure 2-3. Lambda architecture. Credit: Ignacio Mulas Viela and Nicolas Seyvet.

As shown in Figure 2-3, the Lambda architecture is composed of three layers: a batch layer, real-time (or streaming) layer, and serving layer. Both the batch and real-time layers receive a copy of the event, in parallel. The serving layer then aggregates and merges computation results from both layers into a complete answer.

The batch layer (aka, historical layer) has two major tasks: managing historical data and recomputing results such as machine-learning models. Computations are based on iterating over the entire historical data set. Since the data set can be large, this produces accurate results at the cost of high latency due to high computation time.

The real-time layer (speed layer, streaming layer) provides low-latency results in near real-time fashion. It performs updates using incremental algorithms, thus significantly reducing computation costs, often at the expense of accuracy.

The Kappa architecture simplifies the Lambda architecture by removing the batch layer and replacing it with a streaming layer. To understand how this is possible, one must first understand that a batch is a data set with a start and an end (bounded), while a stream has no start or end and is infinite (unbounded). Because a batch is a bounded stream, one can conclude that batch processing is a subset of stream processing. Hence, the Lambda batch layer results can also be obtained by using a streaming engine. This simplification reduces the architecture to a single streaming engine capable of ingesting the needed volumes of data to handle both batch and real-time processing. Overall system complexity significantly decreases with Kappa architecture. See Figure 2-4.

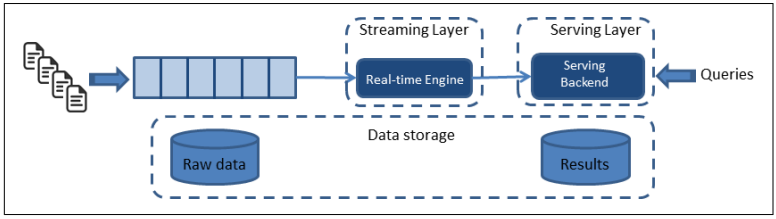

Figure 2-4. Kappa architecture. Credit: Ignacio Mulas Viela and Nicolas Seyvet.

Intrinsically, there are four main principles in the Kappa architecture:

1. **Everything is a stream:** batch operations become a subset of streaming operations. Hence, everything can be treated as a stream.
2. **Immutable data sources:** raw data (data source) is persisted and views are derived, but a state can always be recomputed, as the initial record is never changed.
3. **Single analytics framework:** keep it short and simple (KISS) principle. A single analytics engine is required. Code, maintenance, and upgrades are considerably reduced.
4. **Replay functionality:** computations and results can evolve by replaying the historical data from a stream.

In order to respect principle four, the data pipeline must guarantee that events stay in order from generation to ingestion. This is critical to guarantee consistency of results, as this guarantees deterministic computation results. Running the same data twice through a computation must produce the same result.

These four principles do, however, put constraints on building the analytics pipeline.

Building the Analytics Pipeline

Let's start concretizing how we can build such a data pipeline and identify the sorts of components required.

The first component is a scalable, distributed messaging system with events ordering and at-least-once delivery guarantees. Kafka can connect the output of one process to the input of another via a publish-subscribe mechanism. Using it, we can build something similar to the Unix pipe systems where the output produced by one command is the input to the next.

The second component is a scalable stream analytics engine. Inspired by Google's "Dataflow Model" paper, Flink, at its core is a streaming dataflow engine that provides data distribution, communication, and fault tolerance for distributed computations over data streams. One of its most interesting API features allows usage of the event timestamp to build time windows for computations.

The third and fourth components are a real-time analytics store, Elasticsearch, and a powerful visualization tool, Kibana. Those two components are not critical, but they're useful to store and display raw data and results.

Mapping the Kappa architecture to its implementation, Figure 2-5 illustrates the resulting data pipeline.

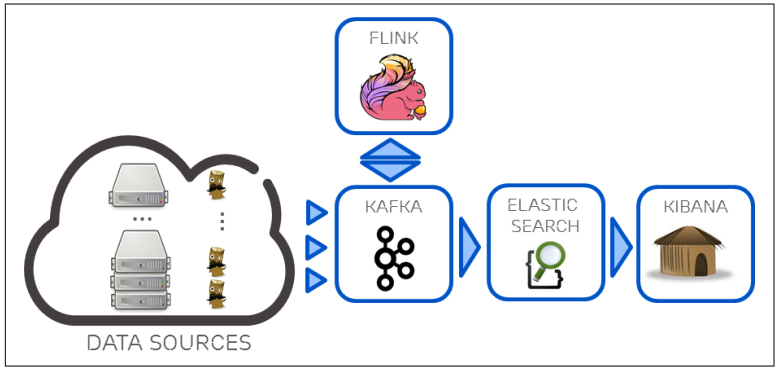

Figure 2-5. Kappa architecture reflected in a data pipeline. Credit: Ignacio Mulas Viela and Nicolas Seyvet.

This pipeline creates a composable environment where outputs of different jobs can be reused as inputs to another. Each job can thus be reduced to a simple, well-defined role. The composability allows for fast development of new features. In addition, data ordering and delivery are guaranteed, making results consistent. Finally, event timestamps can be used to build time windows for computations.

Applying the above to our telco use case, each physical host and virtual machine (VM) telemetry and log event is collected and sent to Kafka. We use collectd on the hosts, and ceilometer on the VMs for telemetry, and logstash-forwarder for logs. Kafka then delivers this data to different Flink jobs that transform and process the data. This monitoring gives us both the physical and virtual resource views of the system.

With the data pipeline in place, let's look at how a Bayesian model can be used to detect novelties in a telco cloud.

Incorporating a Bayesian Model to Do Advanced Analytics

To detect novelties, we use a Bayesian model. In this context, novelties are defined as unpredicted situations that differ from previous observations. The main idea behind Bayesian statistics is to compare statistical distributions and determine how similar or different they are. The goal here is to:

1. Determine the distribution of parameters to detect an anomaly.
2. Compare new samples for each parameter against calculated distributions and determine if the obtained value is expected or not.
3. Combine all parameters to determine if there is an anomaly.

Let's dive into the math to explain how we can perform this operation in our analytics framework. Considering the anomaly A, a new sample z, θ observed parameters, $P(\theta)$ the probability distribution of the parameter, $A(z|\theta)$ the probability that z is an anomaly, and X the samples, the Bayesian Principal Anomaly can be written as:

$$A\ (z \mid X) = \int A(\theta) P(\theta|X)$$

A principal anomaly as defined is valid also for multivariate distributions. The approach taken evaluates the anomaly for each variable separately, and then combines them into a total anomaly value.

An anomaly detector considers only a small part of the variables, and typically only a single variable with a simple distribution like Poisson or Gauss, can be called a *micromodel*. A micromodel with Gaussian distribution will look like Figure 2-6.

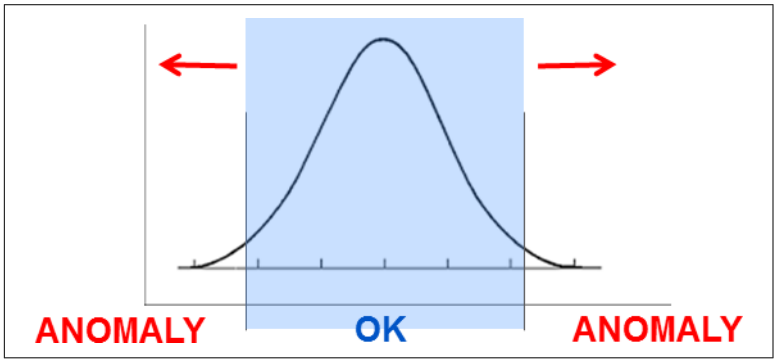

Figure 2-6. Micromodel with Gaussian distribution. Credit: Ignacio Mulas Viela and Nicolas Seyvet.

An array of micromodels can then be formed, with one micromodel per variable (or small set of variables). Such an array can be called a *component*. The anomaly values from the individual detectors then have to be combined into one anomaly value for the whole component. The combination depends on the use case. Since accuracy is important (avoid false positives) and parameters can be assumed to be fairly independent from one another, then the principal anomaly for the component can be calculated as the maximum of the micromodel anomalies, but scaled down to meet the correct false alarm rate (i.e., weighted influence of components to improve the accuracy of the principal anomaly detection).

However, there may be many different "normal" situations. For example, the normal system behavior may vary within weekdays or time of day. Then, it may be necessary to model this with several components, where each component learns the distribution of one cluster. When a new sample arrives, it is tested by each component. If it is considered anomalous by all components, it is considered anomalous. If any component finds the sample normal, then it is normal.

Applying this to our use case, we used this detector to spot errors or deviations from normal operations in a telco cloud. Each parameter θ is any of the captured metrics or logs resulting in many micromo-

dels. By keeping a history of past models and computing a principal anomaly for the component, we can find statistically relevant novelties. These novelties could come from configuration errors, a new error in the infrastructure, or simply a new state of the overall system (i.e., a new set of virtual machines).

Using the number of generated logs (or log frequency) appears to be the most significant feature to detect novelties. By modeling the statistical function of generated logs over time (or log frequency), the model can spot errors or novelties accurately. For example, let's consider the case where a database becomes unavailable. At that time, any applications depending on it start logging recurring errors, (e.g., "Database X is unreachable..."). This raises the log frequency, which triggers a novelty in our model and detector.

The overall data pipeline, combining the transformations mentioned previously, will look like Figure 2-7.

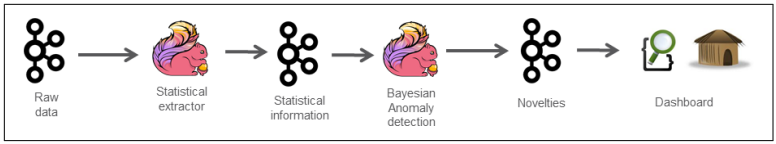

Figure 2-7. Data pipeline with combination of analytics and Bayesian anomaly detector. Credit: Ignacio Mulas Viela and Nicolas Seyvet.

This data pipeline receives the raw data, extracts statistical information (such as log frequencies per machine), applies the Bayesian anomaly detector over the interesting features (statistical and raw), and outputs novelties whenever they are found.

Conclusion

In this blog post, we have presented an approach using the Kappa architecture and a self-training (online) Bayesian model to yield quick, accurate analytics.

The Kappa architecture allows us to develop a new generation of analytics systems. Remember, this architecture has four main principles: data is immutable, everything is a stream, a single stream engine is used, and data can be replayed. It simplifies both the software systems and the development and maintenance of machine-learning models. Those principles can easily be applied to most use cases.

The Bayesian model quickly detects novelties in our cloud. This type of online learning has the advantage of adapting over time to new situations, but one of its main challenges is a lack of ready-to-use algorithms. However, the analytics landscape is evolving quickly, and we are confident that a richer environment can be expected in the near future.

CHAPTER 3

Intelligent Real-Time Applications

To begin the chapter, we include an excerpt from Tyler Akidau's post on streaming engines for processing unbounded data. In this excerpt, Akidau describes the utility of watermarks and triggers to help determine when results are materialized during processing time. Holden Karau then explores how machine-learning algorithms, particularly Naive Bayes, may eventually be implemented on top of Spark's Structured Streaming API. Next, we include highlights from Ben Lorica's discussion with Anodot's cofounder and chief data scientist Ira Cohen. They explored the challenges in building an advanced analytics system that requires scalable, adaptive, and unsupervised machine-learning algorithms. Finally, Uber's Vinoth Chandar tells us about a variety of processing systems for near-real-time data, and how adding incremental processing primitives to existing technologies can solve a lot of problems.

The World Beyond Batch Streaming

By Tyler Akidau

This is an excerpt. You can read the full blog post on oreilly.com here.

Streaming 102

We just observed the execution of a windowed pipeline on a batch engine. But ideally, we'd like to have lower latency for our results, and we'd also like to natively handle unbounded data sources.

Switching to a streaming engine is a step in the right direction; but whereas the batch engine had a known point at which the input for each window was complete (i.e., once all of the data in the bounded input source had been consumed), we currently lack a practical way of determining completeness with an unbounded data source. Enter watermarks.

Watermarks

Watermarks are the first half of the answer to the question: "When in processing time are results materialized?" Watermarks are temporal notions of input completeness in the event-time domain. Worded differently, they are the way the system measures progress and completeness relative to the event times of the records being processed in a stream of events (either bounded or unbounded, though their usefulness is more apparent in the unbounded case).

Recall this diagram from "Streaming 101," slightly modified here, where I described the skew between event time and processing time as an ever-changing function of time for most real-world distributed data processing systems (Figure 3-1).

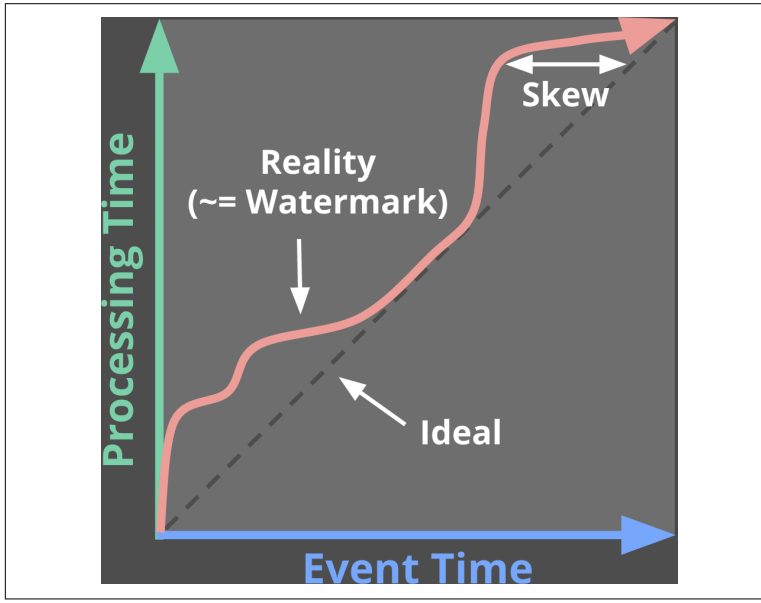

Figure 3-1. Event time progress, skew, and watermarks. Credit: Tyler Akidau.

That meandering red line that I claimed represented reality is essentially the watermark; it captures the progress of event time completeness as processing time progresses. Conceptually, you can think of the watermark as a function, $F(P) \rightarrow E$, which takes a point in processing time and returns a point in event time. (More accurately, the input to the function is really the current state of everything upstream of the point in the pipeline where the watermark is being observed: the input source, buffered data, data actively being processed, etc. But conceptually, it's simpler to think of it as a mapping from processing time to event time.) That point in event time, E, is the point up to which the system believes all inputs with event times less than E have been observed. In other words, it's an assertion that no more data with event times less than E will ever be seen again. Depending upon the type of watermark, perfect or heuristic, that assertion may be a strict guarantee or an educated guess, respectively:

- **Perfect watermarks:** in the case where we have perfect knowledge of all of the input data, it's possible to construct a perfect watermark. In such a case, there is no such thing as late data; all data are early or on time.
- **Heuristic watermarks:** for many distributed input sources, perfect knowledge of the input data is impractical, in which case the next best option is to provide a heuristic watermark. Heuristic watermarks use whatever information is available about the inputs (partitions, ordering within partitions if any, growth rates of files, etc.) to provide an estimate of progress that is as accurate as possible. In many cases, such watermarks can be remarkably accurate in their predictions. Even so, the use of a heuristic watermark means it may sometimes be wrong, which will lead to late data. We'll learn about ways to deal with late data in "The wonderful thing about triggers, is triggers are wonderful things!" on page 46.

Watermarks are a fascinating and complex topic, with far more to talk about than I can reasonably fit here or in the margin, so a further deep dive on them will have to wait for a future post. For now, to get a better sense of the role that watermarks play, as well as some of their shortcomings, let's look at two examples of a streaming engine using watermarks alone to determine when to materialize output while executing the windowed pipeline from Listing 2. The

example in Figure 3-2 uses a perfect watermark; the one in Figure 3-3 uses a heuristic watermark.

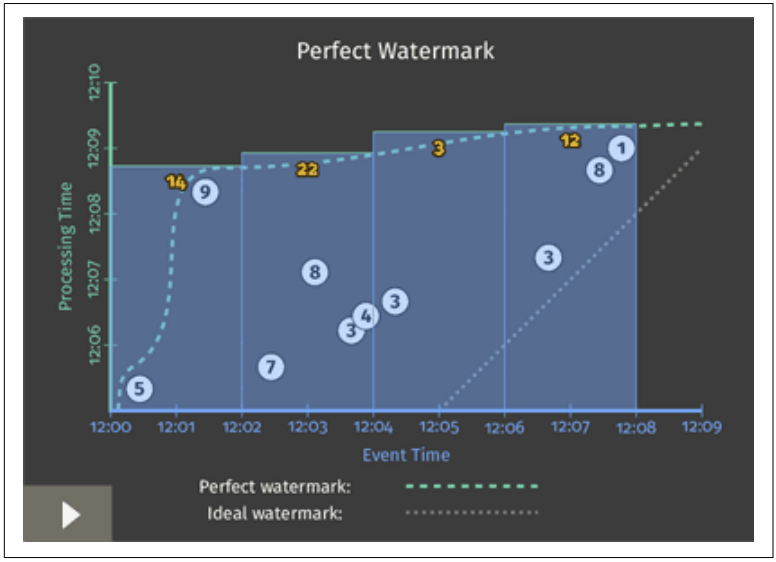

Figure 3-2. Windowed summation on a streaming engine with perfect watermarks. Credit: Tyler Akidau.

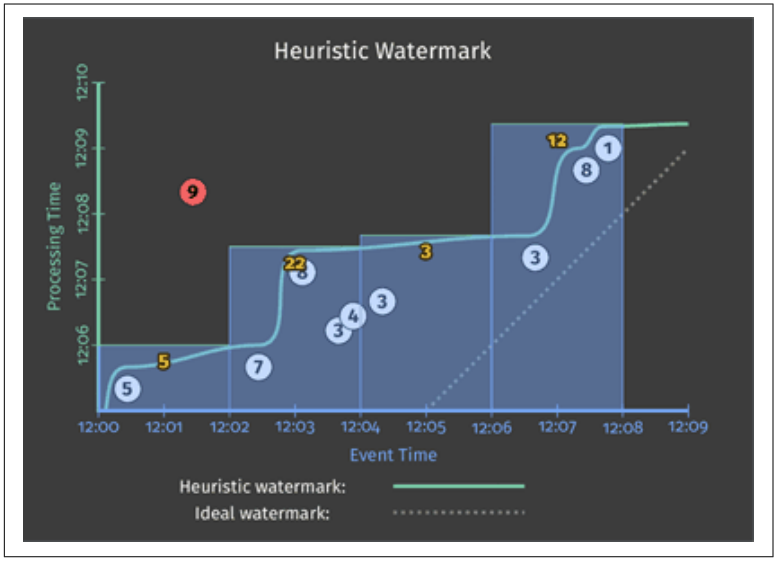

Figure 3-3. Windowed summation on a streaming engine with heuristic watermarks. Credit: Tyler Akidau.

In both cases, windows are materialized as the watermark passes the end of the window. The primary difference between the two executions is that the heuristic algorithm used in watermark calculation on the right fails to take the value of 9 into account, which drastically changes the shape of the watermark. These examples highlight two shortcomings of watermarks (and any other notion of completeness), specifically that they can be:

- **Too slow**: when a watermark of any type is correctly delayed due to known unprocessed data (e.g., slowly growing input logs due to network bandwidth constraints), that translates directly into delays in output if advancement of the watermark is the only thing you depend on for stimulating results.

  This is most obvious in Figure 3-2, where the late-arriving 9 holds back the watermark for all the subsequent windows, even though the input data for those windows become complete earlier. This is particularly apparent for the second window, (12:02, 12:04), where it takes nearly seven minutes from the time the first value in the window occurs until we see any results for the window whatsoever. The heuristic watermark in this example doesn't suffer the same issue quite so badly (five minutes until output), but don't take that to mean heuristic watermarks never suffer from watermark lag; that's really just a consequence of the record I chose to omit from the heuristic watermark in this specific example.

  The important point here is the following: while watermarks provide a very useful notion of completeness, depending upon completeness for producing output is often not ideal from a latency perspective. Imagine a dashboard that contains valuable metrics, windowed by hour or day. It's unlikely that you'd want to wait a full hour or day to begin seeing results for the current window; that's one of the pain points of using classic batch systems to power such systems. Instead, it would be much nicer to see the results for those windows refine over time as the inputs evolve and eventually become complete.
- **Too fast**: when a heuristic watermark is incorrectly advanced earlier than it should be, it's possible for data with event times before the watermark to arrive some time later, creating late data. This is what happened in the example in Figure 3-3: the watermark advanced past the end of the first window before all

the input data for that window had been observed, resulting in an incorrect output value of 5 instead of 14. This shortcoming is strictly a problem with heuristic watermarks; their heuristic nature implies they will sometimes be wrong. As a result, relying on them alone for determining when to materialize output is insufficient if you care about correctness.

In "Streaming 101," I made some rather emphatic statements about notions of completeness being insufficient for robust out-of-order processing of unbounded data streams. These two shortcomings, watermarks being too slow or too fast, are the foundations for those arguments. You simply cannot get both low latency *and* correctness out of a system that relies solely on notions of completeness. Addressing these shortcomings is where triggers come into play.

The wonderful thing about triggers, is triggers are wonderful things!

Triggers are the second half of the answer to the question: "When in processing time are results materialized?" Triggers declare when output for a window should happen in processing time (though the triggers themselves may make those decisions based off of things that happen in other time domains, such as watermarks progressing in the event-time domain). Each specific output for a window is referred to as a *pane* of the window.

Examples of signals used for triggering include:

- **Watermark progress (i.e., event-time progress)**, an implicit version of which we already saw in Figures 3-2 and 3-3, where outputs were materialized when the watermark passed the end of the window.[1] Another use case is triggering garbage collection when the lifetime of a window exceeds some useful horizon, an example of which we'll see a little later on.
- **Processing time progress**, which is useful for providing regular, periodic updates because processing time (unlike event time) always progresses more or less uniformly and without delay.

1 Truth be told, we actually saw such an implicit trigger in use in all of the examples thus far, even the batch ones; in batch processing, the watermark conceptually advances to infinity at the end of the batch, thus triggering all active windows, even global ones spanning all of event time.

- **Element counts**, which are useful for triggering after some finite number of elements have been observed in a window.
- **Punctuations**, or other data-dependent triggers, where some record or feature of a record (e.g., an EOF element or a flush event) indicates that output should be generated.

In addition to simple triggers that fire based off of concrete signals, there are also composite triggers that allow for the creation of more sophisticated triggering logic. Example composite triggers include:

- **Repetitions**, which are particularly useful in conjunction with processing time triggers for providing regular, periodic updates.
- **Conjunctions** (logical AND), which fire only once *all* child triggers have fired (e.g., after the watermark passes the end of the window AND we observe a terminating punctuation record).
- **Disjunctions** (logical OR), which fire after *any* child triggers fire (e.g., after the watermark passes the end of the window OR we observe a terminating punctuation record).
- **Sequences**, which fire a progression of child triggers in a predefined order.

To make the notion of triggers a bit more concrete (and give us something to build upon), let's go ahead and make explicit the implicit default trigger used in 3-2 and 3-3 by adding it to the code from Listing 2.

Example 3-1. Explicit default trigger

```
PCollection<KV<String, Integer>> scores = input
  .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2)))
               .triggering(AtWatermark()))
  .apply(Sum.integersPerKey());
```

With that in mind, and a basic understanding of what triggers have to offer, we can look at tackling the problems of watermarks being too slow or too fast. In both cases, we essentially want to provide some sort of regular, materialized updates for a given window, either before or after the watermark advances past the end of the window (in addition to the update we'll receive at the threshold of the watermark passing the end of the window). So, we'll want some sort of repetition trigger. The question then becomes: what are we repeating?

In the "too slow" case (i.e., providing early, speculative results), we probably should assume that there may be a steady amount of incoming data for any given window because we know (by definition of being in the early stage for the window) that the input we've observed for the window is thus far incomplete. As such, triggering periodically when processing time advances (e.g., once per minute) is probably wise because the number of trigger firings won't be dependent upon the amount of data actually observed for the window; at worst, we'll just get a steady flow of periodic trigger firings.

In the "too fast" case (i.e., providing updated results in response to late data due to a heuristic watermark), let's assume our watermark is based on a relatively accurate heuristic (often a reasonably safe assumption). In that case, we don't expect to see late data very often, but when we do, it would be nice to amend our results quickly. Triggering after observing an element count of 1 will give us quick updates to our results (i.e., immediately any time we see late data), but is not likely to overwhelm the system, given the expected infrequency of late data.

Note that these are just examples: we're free to choose different triggers (or to choose not to trigger at all for one or both of them) if appropriate for the use case at hand.

Lastly, we need to orchestrate the timing of these various triggers: early, on-time, and late. We can do this with a `Sequence` trigger and a special `OrFinally` trigger, which installs a child trigger that terminates the parent trigger when the child fires.

Example 3-2. Manually specified early and late firings

```
PCollection<KV<String, Integer>> scores = input
  .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2)))
               .triggering(Sequence(
                 Repeat(AtPeriod(Duration.standardMinutes(1)))
                   .OrFinally(AtWatermark()),
                 Repeat(AtCount(1))))
  .apply(Sum.integersPerKey());
```

However, that's pretty wordy. And given that the pattern of repeated-early | on-time | repeated-late firings is so common, we provide a custom (but semantically equivalent) API in Dataflow to make specifying such triggers simpler and clearer:

Example 3-3. Early and late firings via the early/late API

```
PCollection<KV<String, Integer>> scores = input
  .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2)))
                .triggering(
                  AtWatermark()
                    .withEarlyFirings(AtPeriod(Duration
                    .standardMinutes(1)))
                    .withLateFirings(AtCount(1))))
  .apply(Sum.integersPerKey());
```

Executing either Listing 4 or 5 on a streaming engine (with both perfect and heuristic watermarks, as before) then yields results that look like Figures 3-4 and 3-5.

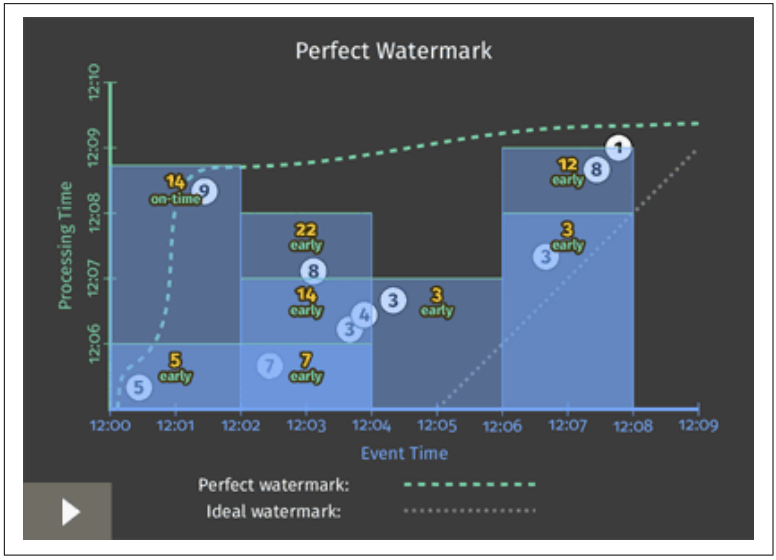

Figure 3-4. Windowed summation on a streaming engine with early and late firings (perfect watermark). Credit: Tyler Akidau.

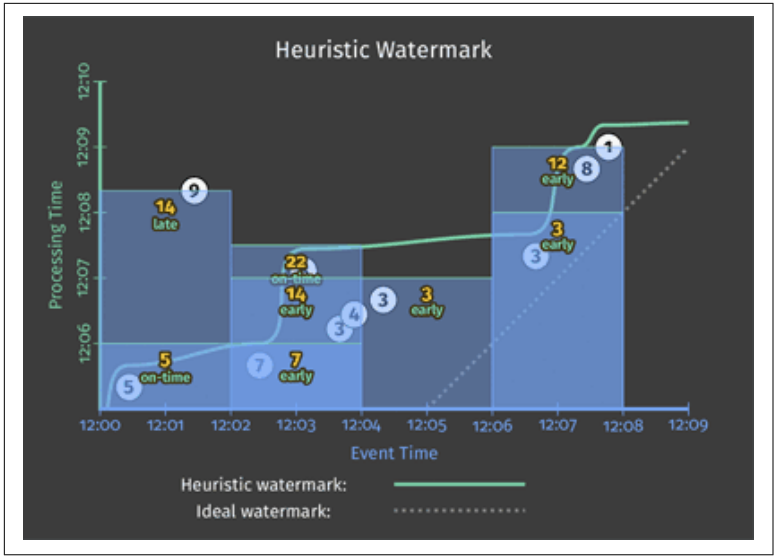

Figure 3-5. Windowed summation on a streaming engine with early and late firings (heuristic watermark). Credit: Tyler Akidau.

This version has two clear improvements over Figures 3-2 and 3-3:

- For the "watermarks too slow" case in the second window, (12:02, 12:04): we now provide periodic early updates once per minute. The difference is most stark in the perfect watermark case, where time-to-first-output is reduced from almost seven minutes down to three and a half; but it's also clearly improved in the heuristic case as well. Both versions now provide steady refinements over time (panes with values 7, 14, then 22), with relatively minimal latency between the input becoming complete and materialization of the final output pane for the window.
- For the "heuristic watermarks too fast" case in the first window, (12:00, 12:02): when the value of 9 shows up late, we immediately incorporate it into a new, corrected pane with a value of 14.

One interesting side effect of these new triggers is that they effectively normalize the output pattern between the perfect and heuristic watermark versions. Whereas the two versions in Figures 3-2 and 3-3 were starkly different, the two versions here look quite similar.

The biggest remaining difference at this point is window lifetime bounds. In the perfect watermark case, we know we'll never see any more data for a window once the watermark has passed the end of it; hence we can drop all of our state for the window at that time. In the heuristic watermark case, we still need to hold on to the state for a window for some amount of time to account for late data. But as of yet, our system doesn't have any good way of knowing just how long state needs to be kept around for each window. That's where allowed lateness comes in.

Extend Structured Streaming for Spark ML

By Holden Karau

You can read this post on oreilly.com here.

Spark's new ALPHA Structured Streaming API has caused a lot of excitement because it brings the Data set/DataFrame/SQL APIs into a streaming context. In this initial version of Structured Streaming, the machine-learning APIs have not yet been integrated. However, this doesn't stop us from having fun exploring how to get machine learning to work with Structured Streaming. (Simply keep in mind that this is exploratory, and things will change in future versions.)

For our "Spark Structured Streaming for machine learning" talk at Strata + Hadoop World New York 2016, we started early proof-of-concept work to integrate Structured Streaming and machine learning available in the spark-structured-streaming-ml repo. If you are interested in following along with the progress toward Spark's ML pipelines supporting Structured Streaming, I encourage you to follow SPARK-16424 and give us your feedback on our early draft design document.

One of the simplest streaming machine-learning algorithms you can implement on top of Structured Streaming is Naive Bayes, since much of the computation can be simplified to grouping and aggregating. The challenge is how to collect the aggregate data in such a way that you can use it to make predictions. The approach taken in the current streaming Naive Bayes won't directly work, as the ForeachSink available in Spark Structured Streaming executes the actions on the workers, so you can't update a local data structure with the latest counts.

Instead, Spark's Structured Streaming has an in-memory table output format you can use to store the aggregate counts:

```
// Compute the counts using a Dataset transformation
 val counts = ds.flatMap{
   case LabeledPoint(label, vec) =>
     vec.toArray.zip(Stream from 1).map(value =>
     LabeledToken(label, value))
 }.groupBy($"label", $"value").agg(count($"value").alias
 ("count"))
   .as[LabeledTokenCounts]
 // Create a table name to store the output in
 val tblName = "qbsnb" + java.util.UUID.randomUUID.toString
 .filter(_ != '-').toString
 // Write out the aggregate result in complete form to the
 // in memory table
 val query = counts.writeStream.outputMode(OutputMode
 .Complete())
   .format("memory").queryName(tblName).start()
 val tbl = ds.sparkSession.table(tblName).as
 [LabeledTokenCounts]
```

The initial approach taken with Naive Bayes is not easily generalizable to other algorithms, which cannot as easily be represented by aggregate operations on a `Dataset`. Looking back at how the early DStream-based Spark Streaming API implemented machine learning can provide some hints on one possible solution. Provided you can come up with an `update` mechanism on how to merge new data into your existing model, the `DStream foreachRDD` solution allows you to access the underlying microbatch view of the data. Sadly, `foreachRDD` doesn't have a direct equivalent in Structured Streaming, but by using a custom sink, you can get similar behavior in Structured Streaming.

The sink API is defined by StreamSinkProvider, which is used to create an instance of the Sink given a SQLContext and settings about the sink, and Sink trait, which is used to process the actual data on a batch basis:

```
abstract class ForeachDatasetSinkProvider extends
StreamSinkProvider {
  def func(df: DataFrame): Unit

  def createSink(
      sqlContext: SQLContext,
      parameters: Map[String, String],
      partitionColumns: Seq[String],
      outputMode: OutputMode): ForeachDatasetSink = {
```

```
      new ForeachDatasetSink(func)
    }
  }

  case class ForeachDatasetSink(func: DataFrame => Unit)
      extends Sink {
    override def addBatch(batchId: Long, data: DataFrame):
    Unit = {
      func(data)
    }
  }
```

To use a third-party sink, you can specify the full class name of the sink, as with writing DataFrames to custom formats. Since you need to specify the full class name of the format, you need to ensure that any instance of the SinkProvider can update the model—and since you can't get access to the sink object that gets constructed—you need to make the model outside of the sink:

```
object SimpleStreamingNaiveBayes {
  val model = new StreamingNaiveBayes()
}

class StreamingNaiveBayesSinkProvider extends
ForeachDatasetSinkProvider {
  override def func(df: DataFrame) {
    val spark = df.sparkSession
    SimpleStreamingNaiveBayes.model.update(df)
  }
}
```

You can use the custom sink shown in the previous example to integrate machine learning into Structured Streaming while you are waiting for Spark ML to be updated with Structured Streaming:

```
  // Train using the model inside SimpleStreamingNaiveBayes
  // object - if called on multiple streams, all streams will
  // update the same model :(
  // or would except if not for the hardcoded query name
  // preventing multiple of the same running.
  def train(ds: Dataset[_]) = {
    ds.writeStream.format(
      "com.highperformancespark.examples.structuredstreaming." +
        "StreamingNaiveBayesSinkProvider")
      .queryName("trainingnaiveBayes")
      .start()
  }
```

If you are willing to throw caution to the wind, you can access some Spark internals to construct a sink that behaves more like the origi-

nal `foreachRDD`. If you are interested in custom sink support, you can follow SPARK-16407 or this PR.

The cool part is, regardless of whether you want to access the internal Spark APIs, you can now handle batch updates in the same way that Spark's earlier streaming machine learning is implemented.

While this certainly isn't ready for production usage, you can see that the Structured Streaming API offers a number of different ways it can be extended to support machine learning.

You can learn more in High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark.

Semi-Supervised, Unsupervised, and Adaptive Algorithms for Large-Scale Time Series

By Ben Lorica

You can read this post on oreilly.com here.

Since my days in quantitative finance, I've had a longstanding interest in time-series analysis. Back then, I used statistical (and data mining) techniques on relatively small volumes of financial time series. Today's applications and use cases involve data volumes and speeds that require a new set of tools for data management, collection, and simple analysis.

On the analytics side, applications are also beginning to require online machine-learning algorithms that are able to scale, adaptive, and free of a rigid dependence on labeled data.

In a recent episode of the *O'Reilly Data Show*, I spoke with Ira Cohen, cofounder and chief data scientist at Anodot (full disclosure: I'm an advisor to Anodot). I talked with Cohen about the challenges in building an advanced analytics system for intelligent applications at extremely large scale.

Here are some highlights from our conversation:

Surfacing Anomalies

> A lot of systems have a concept called dashboarding, where you put your regular things that you look at—the total revenue, the total amount of traffic to my website. … We have a parallel concept that we called Anoboard, which is an anomaly board. An anomaly

board is basically showing you only the things that right now have some strange patterns to them. … So, out of the millions, here are the top 20 things you should be looking at because they have a strange behavior to them.

… The Anoboard is something that gets populated by machine-learning algorithms. … We only highlight the things that you need to look at rather than the subset of things that you're used to looking at, but that might not be relevant for discovering anything that's happening right now.

Adaptive, Online, Ensupervised Algorithms at Scale

We are a generic platform that can take any time series into it, and we'll output anomalies. Like any machine-learning system, we have success criteria. In our case, it's that the number of false positives should be minimal, and the number of true detections should be the highest possible. Given those constraints and given that we are agnostic to the data so we're generic enough, we have to have a set of algorithms that will fit almost any type of metrics, any type of time series signals that get sent to us.

To do that, we had to observe and collect a lot of different types of time-series data from various types of customers. … We have millions of metrics in our system today. … We have over a dozen different algorithms that fit different types of signals. We had to design them and implement them, and obviously because our system is completely unsupervised, we also had to design algorithms that know how to choose the right one for every signal that comes in.

… When you have millions of time series and you're measuring a large ecosystem, there are relationships between the time series, and the relationships and anomalies between different signals do tell a story. … There are a set of learning algorithms behind the scene that do this correlation automatically.

… All of our algorithms are adaptive, so they take in samples and basically adapt themselves over time to fit the samples. Let's say there is a regime change. It might trigger an anomaly, but if it stays in a different regime, it will learn that as the new normal. … All our algorithms are completely online, which means they adapt themselves as new samples come in. This actually addresses the second part of the first question, which was scale. We know we have to be adaptive. We want to track 100% of the metrics, so it's not a case where you can collect a month of data, learn some model, put it in production, and then everything is great and you don't have to do anything. You don't have to relearn anything. … We assume that we have to relearn everything all the time because things change all the time.

Discovering Relationships Among KPIs and Semi-Supervised Learning

> We find relationships between different KPIs and show it to a user; it's often something they are not aware of and are surprised to see. … Then, when they think about it and go back, they realize, 'Oh, yeah. That's true.' That completely changes their way of thinking. … If you're measuring all sorts of business KPIs, nobody knows the relationships between things. They can only conjecture about them, but they don't really know it.
>
> … I came from a world of semi-supervised learning where you have some labels, but most of the data is unlabeled. I think this is the reality for us as well. We get some feedback from users, but it's a fraction of the feedback you need if you want to apply supervised learning methods. Getting that feedback is actually very, very helpful. … Because I'm from the semi-supervised learning world, I always try to see where I can get some inputs from users, or from some oracle, but I never want to rely on it being there.

Related Resources:

- "Building self-service tools to monitor high-volume time-series data" (a previous episode of the *Data Show*)
- *Introduction to Apache Kafka*
- "An Introduction to Time Series with Team Apache"
- "How intelligent data platforms are powering smart cities"

Uber's Case for Incremental Processing on Hadoop

By Vinoth Chandar

You can read this post on oreilly.com here.

Uber's mission is to provide "transportation as reliable as running water, everywhere, for everyone." To fulfill this promise, Uber relies on making data-driven decisions at every level, and most of these decisions could benefit from faster data processing such as using data to understand areas for growth or accessing fresh data by the city operations team to debug each city. Needless to say, the choice of data processing systems and the necessary SLAs are the topics of daily conversations between the data team and the users at Uber.

In this post, I would like to discuss the choices of data processing systems for near-real-time use cases, based on experiences building data infrastructure at Uber as well as drawing from previous experiences. In this post, I argue that by adding new incremental processing primitives to existing Hadoop technologies, we will be able to solve a lot more problems, at reduced cost and in a unified manner. At Uber, we are building our systems to tackle the problems outlined here, and are open to collaborating with like-minded organizations interested in this space.

Near-Real-Time Use Cases

First, let's establish the kinds of use cases we are talking about: cases in which up to one-hour latency is tolerable are well understood and mostly can be executed using traditional batch processing via MapReduce/Spark, coupled with incremental ingestion of data into Hadoop/S3. On the other extreme, cases needing less than one to two seconds of latency typically involve pumping your data into a scale-out key value store (having worked on one at scale) and querying that. Stream processing systems like Storm, Spark Streaming, and Flink have carved out a niche of operating really well at practical latencies of around one to five minutes, and are needed for things like fraud detection, anomaly detection, or system monitoring—basically, those decisions made by machines with quick turnaround or humans staring at computer screens as their day job.

That leaves us with a wide chasm of five-minute to one-hour end-to-end processing latency, which I refer to in this post as *near-real-time* (see Figure 3-6). Most such cases are either powering business dashboards and/or aiding some human decision making. Here are some examples where near-real-time could be applicable:

- Observing whether something was anomalous across the board in the last *x* minutes
- Gauging how well the experiments running on the website performed in the last *x* minutes
- Rolling up business metrics at *x*-minute intervals
- Extracting features for a machine-learning pipeline in the last *x* minutes

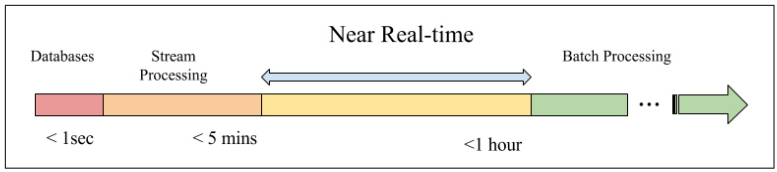

Figure 3-6. Different shades of processing latency with the typical technologies used therein. Credit: Vinoth Chandar.

Incremental Processing via "Mini" Batches

The choices to tackle near-real-time use cases are pretty open-ended. Stream processing can provide low latency, with budding SQL capabilities, but it requires the queries to be predefined in order to work well. Proprietary warehouses have a lot of features (e.g., transactions, indexes, etc.) and can support ad hoc and predefined queries, but such proprietary warehouses are typically limited in scale and are expensive. Batch processing can tackle massive scale and provides mature SQL support via Spark SQL/Hive, but the processing styles typically involve larger latency. With such fragmentation, users often end up making their choices based on available hardware and operational support within their organizations. We will circle back to these challenges at the conclusion of this post.

For now, I'd like to outline some technical benefits to tackling near-real-time use cases via "mini" batch jobs run every *x* minutes, using Spark/MR as opposed to running stream-processing jobs. Analogous to "micro" batches in Spark Streaming (operating at second-by-second granularity), "mini" batches operate at minute-by-minute granularity. Throughout the post, I use the term *incremental processing* collectively to refer to this style of processing.

Increased efficiency

Incrementally processing new data in "mini" batches could be a much more efficient use of resources for the organization. Let's take a concrete example, where we have a stream of Kafka events coming in at 10K/sec and we want to count the number of messages in the last 15 minutes across some dimensions. Most stream-processing pipelines use an external result store (e.g., Cassandra, ElasticSearch) to keep aggregating the count, and keep the YARN/Mesos containers running the whole time. This makes sense in the less-than-five-minute latency windows such pipelines operate on. In practice, typical YARN container start-up costs tend to be around a minute.

In addition, to scale the writes to the result stores, we often end up buffering and batching updates, and this protocol needs the containers to be long-running.

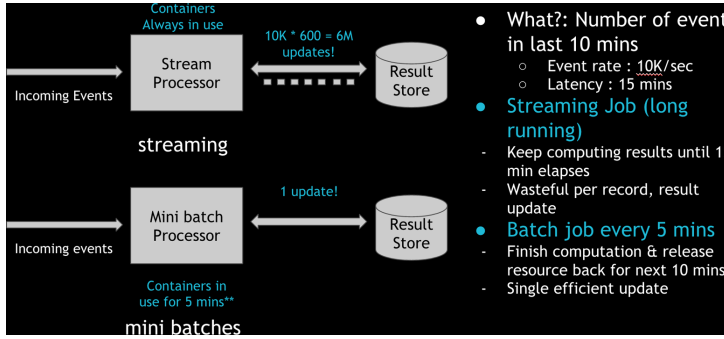

Figure 3-7. Comparison of processing via stream processing engines versus incremental "mini" batch jobs. Credit: Vinoth Chandar.

However, in the near-real-time context, these decisions may not be the best ones. To achieve the same effect, you can use short-lived containers and improve the overall resource utilization. In Figure 3-7, the stream processor performs six million updates over 15 minutes to the result store. But in the incremental processing model, we perform in-memory merge once and only one update to the result store, while using the containers for only five minutes. The incremental processing model is three times more CPU-efficient, and several magnitudes more efficient on updating of the result store. Basically, instead of waiting for work and eating up CPU and memory, the processing wakes up often enough to finish up pending work, grabbing resources on demand.

Built on top of existing SQL engines

Over time, a slew of SQL engines have evolved in the Hadoop/big data space (e.g., Hive, Presto, SparkSQL) that provide better expressibility for complex questions against large volumes of data. These systems have been deployed at massive scale and have hardened over time in terms of query planning, execution, and so forth. On the other hand, SQL on stream processing is still in early stages. By performing incremental processing using existing, much more mature SQL engines in the Hadoop ecosystem, we can leverage the solid foundations that have gone into building them.

For example, joins are very tricky in stream processing, in terms of aligning the streams across windows. In the incremental processing model, the problem naturally becomes simpler due to relatively longer windows, allowing more room for the streams to align across a processing window. On the other hand, if correctness is more important, SQL provides an easier way to expand the join window selectively and reprocess.

Another important advancement in such SQL engines is the support for columnar file formats like ORC/Parquet, which have significant advantages for analytical workloads. For example, joining two Kafka topics with Avro records would be much more expensive than joining two Hive/Spark tables backed by ORC/Parquet file formats. This is because with Avro records, you would end up deserializing the entire record, whereas columnar file formats only read the columns in the record that are needed by the query. For example, if we are simply projecting out 10 fields out of a total 1,000 in a Kafka Avro encoded event, we still end up paying the CPU and IO cost for all fields. Columnar file formats can typically be smart about pushing the projection down to the storage layer (Figure 3-8).

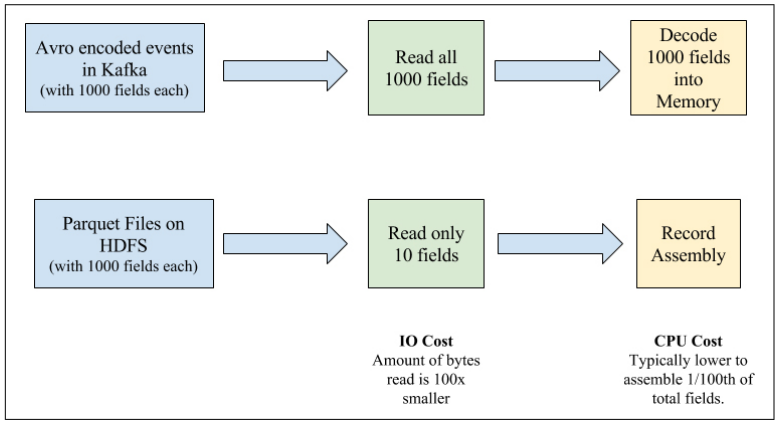

Figure 3-8. Comparison of CPU/IO cost of projecting 10 fields out of 1,000 total, as Kafka events versus columnar files on HDFS. Credit: Vinoth Chandar.

Fewer moving parts

The famed Lambda architecture that is broadly implemented today has two components: speed and batch layers, usually managed by two separate implementations (from code to infrastructure). For

example, Storm is a popular choice for the speed layer, and MapReduce could serve as the batch layer. In practice, people often rely on the speed layer to provide fresher (and potentially inaccurate) results, whereas the batch layer corrects the results of the speed layer at a later time, once the data is deemed complete. With incremental processing, we have an opportunity to implement the Lambda architecture in a unified way at the code level as well as the infrastructure level.

The idea illustrated in Figure 3-9 is fairly simple. You can use SQL, as discussed, or the same batch-processing framework as Spark to implement your processing logic uniformly. The resulting table gets incrementally built by way of executing the SQL on the "new data," just like stream processing, to produce a "fast" view of the results. The same SQL can be run periodically on *all* of the data to correct any inaccuracies (remember, joins are tricky!) to produce a more "complete" view of the results. In both cases, we will be using the same Hadoop infrastructure for executing computations, which can bring down overall operational cost and complexity.

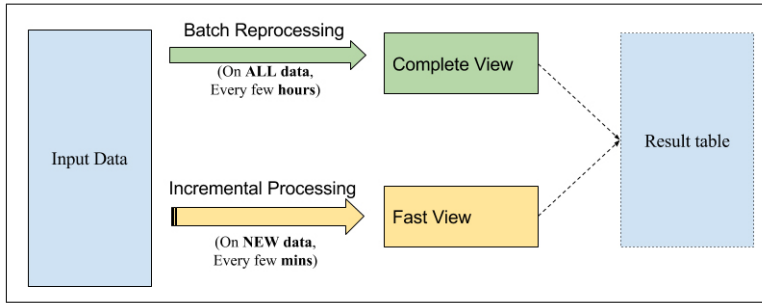

Figure 3-9. Computation of a result table, backed by a fast view via incremental processing and a more complete view via traditional batch processing. Credit: Vinoth Chandar.

Challenges of Incremental Processing

Having laid out the advantages of an architecture for incremental processing, let's explore the challenges we face today in implementing this in the Hadoop ecosystem.

Trade-off: completeness versus latency

In computing, as we traverse the line between stream processing, incremental processing, and batch processing, we are faced with the

same fundamental trade-off. Some applications need all the data and produce more complete/accurate results, whereas some just need data at lower latency to produce acceptable results. Let's look at a few examples.

Figure 3-10 depicts a few sample applications, placing them according to their tolerance for latency and (in)completeness. Business dashboards can display metrics at different granularities because they often have the flexibility to show more incomplete data at lower latencies for recent times while over time getting complete (which also made them the marquee use case for Lambda architecture). For data science/machine-learning use cases, the process of extracting the features from the incoming data typically happens at lower latencies, and the model training itself happens at a higher latency with more complete data. Detecting fraud, on the one hand, requires low-latency processing on the data available thus far. An experimentation platform, on the other hand, needs a fair amount of data, at relatively lower latencies, to keep results of experiments up to date.

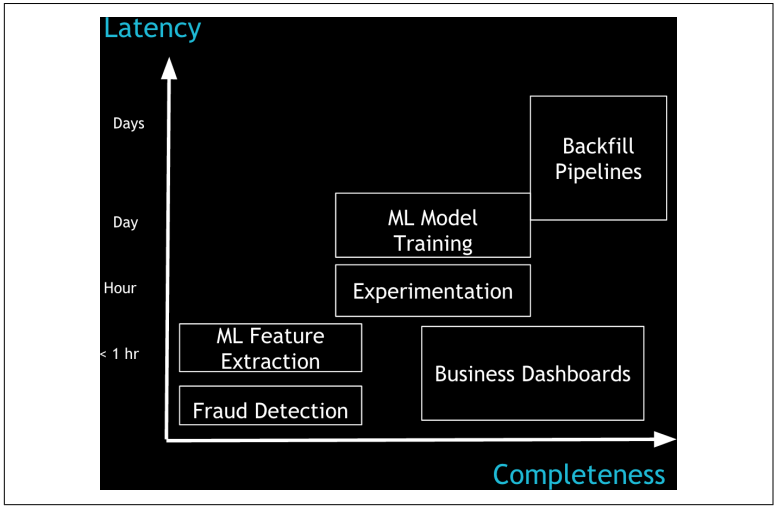

Figure 3-10. Figure showing different Hadoop applications and their tolerance for latency and completeness. Credit: Vinoth Chandar.

The most common cause for lack of completeness is late-arriving data (as explained in detail in this Google Cloud Dataflow deck). In the wild, late data can manifest in infrastructure-level issues, such as data center connectivity flaking out for 15 minutes, or user-level issues, such as a mobile app sending late events due to spotty con-

nectivity during a flight. At Uber, we face very similar challenges, as we presented at Strata + Hadoop World in March 2016.

To effectively support such a diverse set of applications, the programming model needs to treat late-arrival data as a first-class citizen. However, Hadoop processing has typically been batch-oriented on "complete" data (e.g., partitions in Hive), with the responsibility of ensuring completeness also resting solely with the data producer. This is simply too much responsibility for individual producers to take on in today's complex data ecosystems. Most producers end up using stream processing on a storage system like Kafka to achieve lower latencies while relying on Hadoop storage for more "complete" (re)processing. We will expand on this in the next section.

Lack of primitives for incremental processing

As detailed in this article on stream processing, the notions of event time versus arrival time and handling of late data are important aspects of computing with lower latencies. Late data forces recomputation of the time windows (typically, Hive partitions in Hadoop), over which results might have been computed already and even communicated to the end user. Typically, such recomputations in the stream processing world happen incrementally at the record/event level by use of scalable key-value stores, which are optimized for point lookups and updates. However, in Hadoop, recomputing typically just means rewriting the entire (immutable) Hive partition (or a folder inside HDFS for simplicity) and recomputing all jobs that consumed that Hive partition.

Both of these operations are expensive in terms of latency as well as resource utilization. This cost typically cascades across the entire data flow inside Hadoop, ultimately adding hours of latency at the end. Thus, incremental processing needs to make these two operations much faster so that we can efficiently incorporate changes into existing Hive partitions as well as provide a way for the downstream consumer of the table to obtain only the new changes.

Effectively supporting incremental processing boils down to the following primitives:

Upserts
: Conceptually, rewriting the entire partition can be viewed as a highly inefficient upsert operation, which ends up writing way more than the amount of incoming changes. Thus, first-class

support for (batch) upserts becomes an important tool to possess. In fact, recent trends like Kudu and Hive Transactions do point in this direction. Google's "Mesa" paper also talks about several techniques that can be applied in the context of ingesting quickly.

Incremental consumption
: Although upserts can solve the problem of publishing new data to a partition quickly, downstream consumers do not know what data has changed since a point in the past. Typically, consumers learn this by scanning the entire partition/table and recomputing everything, which can take a lot of time and resources. Thus, we also need a mechanism to more efficiently obtain the records that have changed since the last time the partition was consumed.

With the two primitives above, you can support a lot of common use cases by upserting one data set and then incrementally consuming from it to build another data set incrementally. Projections are the most simple to understand, as depicted in Figure 3-11.

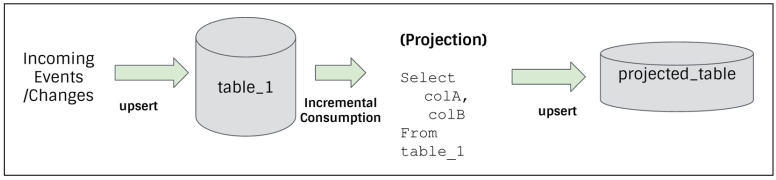

Figure 3-11. Simple example of building of table_1 by upserting new changes, and building a simple projected_table via incremental consumption. Credit: Vinoth Chandar.

Borrowing terminology from Spark Streaming, we can perform simple projections and stream-data set joins much more efficiently at lower latency. Even stream-stream joins can be computed incrementally, with some extra logic to align windows (Figure 3-12).

This is actually one of the rare scenarios where we could save money with hardware while also cutting down the latencies dramatically.

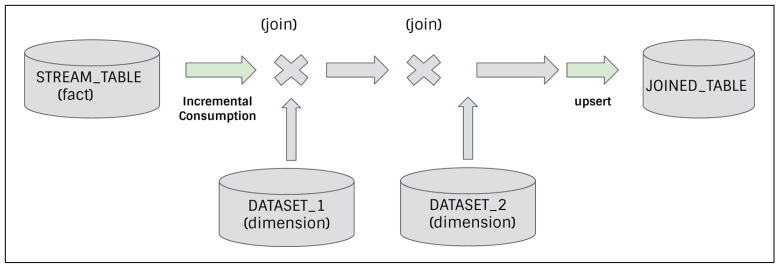

Figure 3-12. More complex example that joins a fact table against multiple dimension tables, to produce a joined table. Credit: Vinoth Chandar.

Shift in mindset

The final challenge is not strictly technical. Organizational dynamics play a central role in which technologies are chosen for different use cases. In many organizations, teams pick templated solutions that are prevalent in the industry, and teams get used to operating these systems in a certain way. For example, typical warehousing latency needs are on the order of hours. Thus, even though the underlying technology could solve a good chunk of use cases at lower latencies, a lot of effort needs to be put into minimizing downtimes or avoiding service disruptions during maintenance. If you are building toward lower latency SLAs, these operational characteristics are essential. On the one hand, teams that solve low-latency problems are extremely good at operating those systems with strict SLAs, and invariably the organization ends up creating silos for batch and stream processing, which impedes realization of the aforementioned benefits to incremental processing on a system like Hadoop.

This is in no way an attempt to generalize the challenges of organizational dynamics, but is merely my own observation as someone who has spanned the online services powering LinkedIn as well as the data ecosystem powering Uber.

Takeaways

I would like to leave you with the following takeaways:

1. Getting really specific about your actual latency needs can save you tons of money.
2. Hadoop can solve a lot more problems by employing primitives to support incremental processing.

3. Unified architectures (code + infrastructure) are the way of the future.

At Uber, we have very direct and measurable business goals/incentives in solving these problems, and we are working on a system that addresses these requirements. Please feel free to reach out if you are interested in collaborating on the project.

CHAPTER 4
Cloud Infrastructure

In this chapter, Rich Morrow outlines the differentiators between the major cloud service providers—Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP)—as a guide for choosing where to launch a managed or unmanaged Hadoop cluster. Then Michael Li and Ariel M'ndange-Pfupfu compare AWS and GCP in terms of cost, performance, and runtime of a typical Spark workload. Finally, Arti Garg and Parviz Deyhim explore how tools like AWS Auto Scaling enable customers to automatically provision resources to meet real-time demand (i.e., scale up or scale down), leading to significant cost savings.

Where Should You Manage a Cloud-Based Hadoop Cluster?

By Rich Morrow

You can read this post on oreilly.com here.

It's no secret that Hadoop and public cloud play very nicely with each other. Rather than having to provision and maintain a set number of servers and expensive networking equipment in house, Hadoop clusters can be spun up in the cloud as a managed service, letting users pay only for what they use, only when they use it.

The scalability and per-workload customizability of public cloud is also unmatched. Rather than having one predefined set of servers (with a set amount of RAM, CPU, and network capability) in-house,

public cloud offers the ability to stand up workload-specific clusters with varying amounts of those resources tailored for each workload. The access to "infinite" amounts of hardware that public cloud offers is also a natural fit for Hadoop, as running 100 nodes for 10 hours is the same cost and complexity level as running 1,000 nodes for one hour.

But among cloud providers, the similarities largely end there. Although Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) each has its own offerings for both managed and VM-based clusters, there are many differentiators that may drive you to one provider over another.

High-Level Differentiators

When comparing the "Big 3" providers in the context of Hadoop operations, several important factors come into play. The high-level ones being:

- **Network isolation:** this refers to the ability to create "private" networks and control routing, IP address spaces, subnetting, and additional security. In this area, AWS, Azure, and GCP each provides roughly equal offerings in the way of VPC, Azure Virtual Networks, and Google Subnetworks, respectively.
- **Type and number of underlying VMs:** for workload customizability, the more VM types, the better. Although all providers have "general," "high CPU," and "high RAM" instance types, AWS takes the ball further with "high storage," GPU, and "high IO" instance types. AWS also has the largest raw number of instance types (currently 55), while both GCP and Azure offer only 18 each.
- **Cost granularity:** for short-term workloads (those completing in just a few hours), costs can vary greatly, with Azure offering the most granular model (minute-by-minute granularity), GCP offering the next best model (pre-billing the first 10 minutes of usage, then billing for each minute), and AWS offering the least flexibility (each full hour billed ahead).
- **Cost flexibility:** how you pay for your compute nodes makes an even bigger difference with regard to cost. AWS wins here with multiple models like Spot Instances & Reservations, which can save up to 90% of the cost of the "on-demand" models, which all three support. Azure and GCP both offer cost-saving mecha-

nisms, with Azure using reservations (but only up to 12 months) and GCP using "sustained-use discounts," which are automatically applied for heavily utilized instances. AWS's reservations can go up to three years, and therefore offer deeper discounts.
- **Hadoop support:** each provider offers a managed, hosted version of Hadoop. AWS's is called Elastic MapReduce or EMR, Azure's is called HDInsight, and GCP's is called DataProc. EMR and DataProc both use core Apache Hadoop (EMR also supports MapR distributions), while Azure uses Hortonworks. Outside of the managed product, each provider also offers the ability to use raw instance capacity to build Hadoop clusters, removing the convenience of the managed service but allowing for much more customizability, including the ability to choose alternate distributions like Cloudera.

Cloud Ecosystem Integration

In addition to the high-level differentiators, one of the public cloud's biggest impacts for Hadoop operations is the integration to other cloud-based services like object stores, archival systems, and the like. Each provider is roughly equivalent with regard to integration and support of:

- **Object storage and data archival:** each provider has near parity here for both cost and functionality, with their respective object stores (S3 for AWS, Blob Storage for Azure, and Google Cloud Storage for GCP) being capable of acting as a data sink or source.
- **NoSQL integrations:** each provider has different, but comparable, managed NoSQL offerings (DynamoDB for AWS, DocumentDB and Managed MongoDB for Azure, and BigTable and BigQuery for GCP), which again can act as data sinks or sources for Hadoop.
- **Dedicated point-to-point fiber interconnects:** each provider offers comparable capability to stretch dedicated, secured fiber connections between on-premise data centers and their respective clouds. AWS's is DirectConnect, Azure's is ExpressRoute, and GCP's is Google Cloud Interconnect.
- **High-speed networking:** AWS and Azure each offer the ability to launch clusters in physically grouped hardware (ideally all

machines in the same rack if possible), allowing the often bandwidth-hungry Hadoop clusters to take advantage of 10 Gbps network interconnects. AWS offers Placement Groups, and Azure offers Affinity Groups. DataProc offers no such capability, but GCP's cloud network is already well known as the most performant of the three.

Big Data Is More Than Just Hadoop

Although the immediate Hadoop-related ecosystems discussed above have few differentiators, the access to the provider's other services and features can give Hadoop administrators many other tools to either perform analytics elsewhere (off the physical cluster) or make Hadoop operations easier to perform.

AWS really shines here with a richer service offering than any of the three. Some big services that come into play for larger systems are Kinesis (which provides near-real-time analytics and stream ingestion), Lambda (for event-driven analytics architectures), Import/Export Snowball (for secure, large-scale data import/export), and AWS IoT (for ingestion and processing of IoT device data)—all services either completely absent at Azure or GCP, or much less mature and not as rich in features.

Key Takeaways

While one could argue any number of additions or edits to the comparisons just described, it represents a good checklist to use when comparing where to launch a managed or unmanaged cloud-based Hadoop cluster. One of the great things about using Hadoop in the cloud is that it's nearly the exact same regardless of distribution or cloud provider. Each of the big three has a mature offering with regards to Hadoop, so whichever partner you choose, you can bet that your cluster will work well, provide cost-saving options, strong security features, and all the flexibility that public cloud provides.

This post was a collaboration between O'Reilly and Pepperdata. See our statement of editorial independence.

Spark Comparison: AWS Versus GCP

By Michael Li and Ariel M'ndange-Pfupfu

You can read this post on oreilly.com here.

There's little doubt that cloud computing will play an important role in data science for the foreseeable future. The flexible, scalable, on-demand computing power available is an important resource, and as a result, there's a lot of competition between the providers of this service. Two of the biggest players in the space are Amazon Web Services (AWS) and Google Cloud Platform (GCP).

This article includes a short comparison of distributed Spark workloads in AWS and GCP—both in terms of setup time and operating cost. We ran this experiment with our students at The Data Incubator, a big data training organization that helps companies hire top-notch data scientists and train their employees on the latest data science skills. Even with the efficiencies built into Spark, the cost and time of distributed workloads can be substantial, and we are always looking for the most efficient technologies so our students are learning the best and fastest tools.

Submitting Spark Jobs to the Cloud

Spark is a popular distributed computation engine that incorporates MapReduce-like aggregations into a more flexible, abstract framework. There are APIs for Python and Java, but writing applications in Spark's native Scala is preferable. That makes job submission simple, as you can package your application and all its dependencies into one JAR file.

It's common to use Spark in conjunction with HDFS for distributed data storage, and YARN for cluster management; this makes Spark a perfect fit for AWS's Elastic MapReduce (EMR) clusters and GCP's Dataproc clusters. Both EMR and Dataproc clusters have HDFS and YARN preconfigured, with no extra work required.

Configuring Cloud Services

Managing data, clusters, and jobs from the command line is more scalable than using the web interface. For AWS, this means installing and using the command-line interface (CLI). You'll have to set up your credentials beforehand as well as make a separate keypair for the EC2 instances that are used under the hood. You'll also need to set up roles—basically permissions—for both users (making sure

they have sufficient rights) and EMR itself (usually, running `aws emr create-default-roles` in the CLI is good enough to get started).

For GCP, the process is more straightforward. If you install the Google Cloud SDK and sign in with your Google account, you should be able to do most things right off the bat. The thing to remember here is to enable the relevant APIs in the API Manager: Compute Engine, Dataproc, and Cloud Storage JSON.

Once you have things set up to your liking, the fun part begins! Using commands like `aws s3 cp` or `gsutil cp`, you can copy your data into the cloud. Once you have buckets set up for your inputs, outputs, and anything else you might need, running your app is as easy as starting up a cluster and submitting the JAR file. Make sure you know where the logs are kept—it can be tricky to track down problems or bugs in a cloud environment.

You Get What You Pay For

When it comes to cost, Google's service is more affordable in several ways. First, the raw cost of purchasing computing power is cheaper. Running a Google Compute Engine machine with four vCPUs and 15 GB of RAM will run you $0.20 every hour, or $0.24 with Dataproc. An identically-specced AWS instance will cost you $0.336 per hour running EMR.

The second factor to consider is the granularity of the billing. AWS charges by the hour, so you pay the full rate even if your job takes 15 minutes. GCP charges by the minute, with a 10-minute minimum charge. This ends up being a huge difference in cost in a lot of use cases.

Both services have various other discounts. You can effectively bid on spare cloud capacity with AWS's spot instances or GCP's preemptible instances. These will be cheaper than dedicated, on-demand instances, but they're not guaranteed to be available. Discounted rates are available on GCP if your instances live for long periods of time (25% to 100% of the month). On AWS, paying some of the costs upfront or buying in bulk can save you some money. The bottom line is, if you're a power user and you use cloud computing on a regular or even constant basis, you'll need to delve deeper and perform your own calculations.

Lastly, the costs for new users wanting to try out these services are lower for GCP. They offer a 60-day free trial with $300 in credit to use however you want. AWS only offers a free tier where certain services are free to a certain point or discounted, so you will end up paying to run Spark jobs. This means that if you want to test out Spark for the first time, you'll have more freedom to do what you want on GCP without worrying about price.

Performance Comparison

We set up a trial to compare the performance and cost of a typical Spark workload. The trial used clusters with one master and five core instances of AWS's `m3.xlarge` and GCP's `n1-standard-4`. They differ slightly in specification, but the number of virtual cores and amount of memory is the same. In fact, they behaved almost identically when it came to job execution time.

The job itself involved parsing, filtering, joining, and aggregating data from the publicly available Stack Exchange Data Dump. We ran the same JAR on a ~50M subset of the data (Cross Validated) and then on the full ~9.5G data set (Figures 4-1 and 4-2).

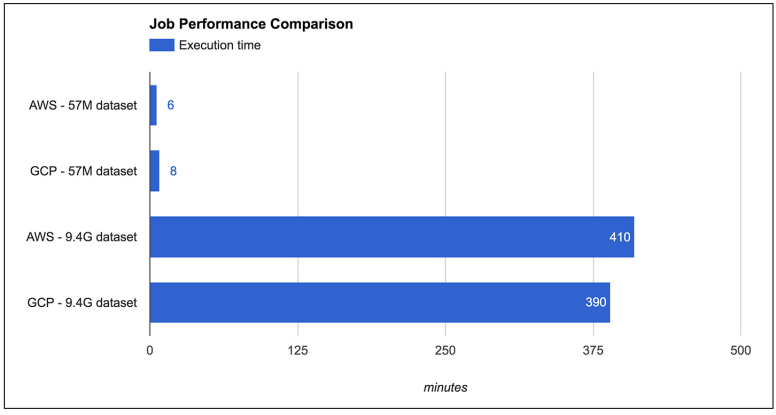

Figure 4-1. Job performance comparison. Credit: Michael Li and Ariel M'ndange-Pfupfu.

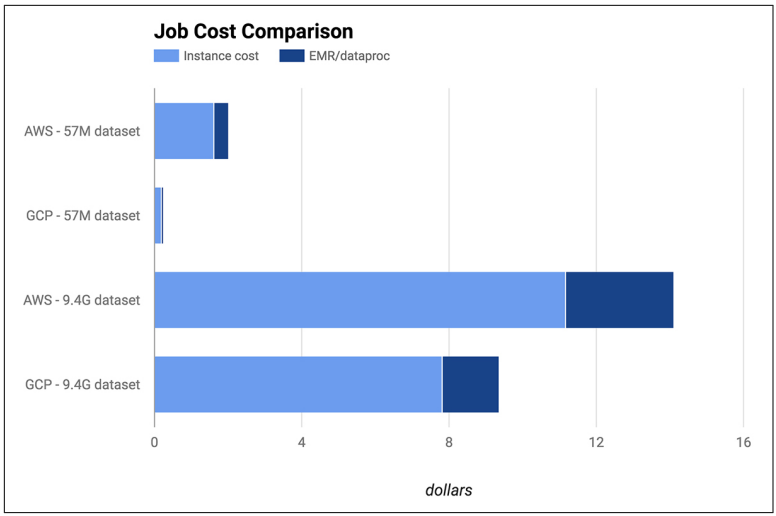

Figure 4-2. Job cost comparison. Credit: Michael Li and Ariel M'ndange-Pfupfu.

The short job clearly benefited from GCP's by-the-minute billing, being charged only for 10 minutes of cluster time, whereas AWS charged for a full hour. But even the longer job was cheaper on GPS both because of fractional-hour billing and a lower per-unit time cost for comparable performance. It's also worth noting that storage costs weren't included in this comparison.

Conclusion

AWS was the first mover in the space, and this shows in the API. Its ecosystem is vast, but its permissions model is a little dated, and its configuration is a little arcane. By contrast, Google is the shiny new entrant in this space and has polished off some of the rough edges. It is missing some features on our wishlist, like an easy way to auto-terminate clusters and detailed billing information broken down by job. Also, for managing tasks programmatically in Python, the API client library isn't as full-featured as AWS's Boto.

If you're new to cloud computing, GCP is easier to get up and running, and the credits make it a tempting platform. Even if you are already used to AWS, you may still find the cost savings make switching worth it, although the switching costs may not make moving to GCP worth it.

Ultimately, it's difficult to make sweeping statements about these services because they're not just one entity; they're entire ecosystems of integrated parts, and both have pros and cons. The real winners are the users. As an example, at The Data Incubator, our PhD data science fellows really appreciate the cost reduction as they learn about distributed workloads. And while our big data corporate training clients may be less price-sensitive, they appreciate being able to crunch enterprise data faster while holding price constant. Data scientists can now enjoy the multitude of options available and the benefits of having a competitive cloud computing market.

Time-Series Analysis on Cloud Infrastructure Metrics

By Arti Garg and Parviz Deyhim

You can read this post on oreilly.com here.

Many businesses are choosing to migrate to, or natively build their infrastructure in the cloud; doing so helps them realize a myriad of benefits. Among these benefits is the ability to lower costs by "right-sizing" infrastructure to adequately meet demand without under- or over-provisioning. For businesses with time-varying resource needs, the ability to "spin-up" and "spin-down" resources based on real-time demand can lead to significant cost savings.

Major cloud-hosting providers like Amazon Web Services (AWS) offer management tools to enable customers to scale their infrastructure to current demand. However, fully embracing such capabilities, such as AWS Auto Scaling, typically requires:

1. Optimized Auto Scaling configuration that can match customers' application resource demands
2. Potential cost saving and business ROI

Attempting to understand potential savings from the use of dynamic infrastructure sizing is not a trivial task. AWS's Auto Scaling capability offers a myriad of options, including resource scheduling and usage-based changes in infrastructure. Businesses must undertake detailed analyses of their applications to understand how best to utilize Auto Scaling, and further analysis to estimate cost savings.

In this article, we will discuss the approach we use at Datapipe to help customers customize Auto Scaling, including the analyses we've

done, and to estimate potential savings. In addition to that, we aim to demonstrate the benefits of applying data science skills to the infrastructure operational metrics. We believe what we're demonstrating here can also be applied to other operational metrics, and hope our readers can apply the same approach to their own infrastructure data.

Infrastructure Usage Data

We approach Auto Scaling configuration optimization by considering a recent client project, where we helped our client realize potential cost savings by finding the most optimized configuration. When we initially engaged with the client, their existing web-application infrastructure consisted of a static and fixed number of AWS instances running at all times. However, after analyzing their historical resource usage patterns, we observed that the application had time-varying CPU usage where, at times, the AWS instances were barely utilized. In this article, we will analyze simulated data that closely matches the customers' usage patterns, but preserves their privacy.

In Figure 4-3, we show two weeks' worth of usage data, similar to that available from Amazon's CloudWatch reporting/monitoring service, which allows you to collect infrastructure-related metrics.

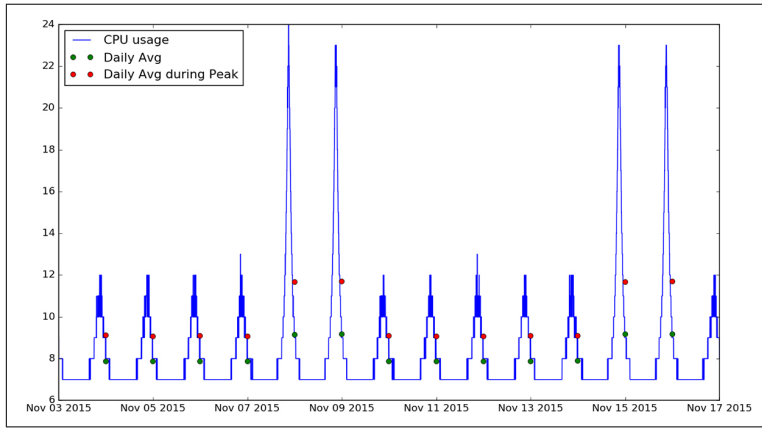

Figure 4-3. Usage collected from the Amazon CloudWatch reporting/monitoring service. Credit: Arti Garg.

A quick visual inspection reveals two key findings:

- Demand for the application is significantly higher during late evenings and nights. During other parts of the day, it remains constant.
- There is a substantial increase in demand over the weekend.

A bit more analysis will allow us to better understand these findings. Let's look at the weekend usage (Saturday–Sunday) and the weekday usage (Monday–Friday), independently. To get a better sense of the uniformity of the daily cycle within each of these two groups, we can aggregate the data to compare the pattern on each day. To do so, we binned the data into regular five-minute intervals throughout the 24-hour day (e.g., 0:00, 0:05, etc.) and determined the minimum, maximum, and average for each of these intervals.

Note that for this example, since the peak period extends slightly past midnight, we defined a "day" as spanning from noon to noon across calendar dates. The difference between the weekday group (red) and the weekend group (blue) is seen quite starkly in the plot below. The dotted lines show the minimum and maximum usage envelopes around the average, which is shown with the solid line in Figure 4-4.

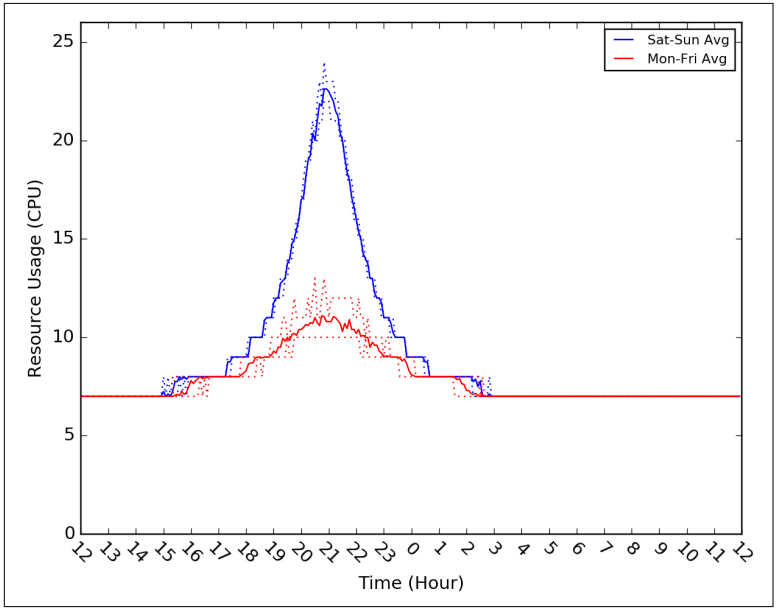

Figure 4-4. Difference between the weekday group (red) and the weekend group (blue). Credit: Arti Garg.

In this example, it is also visually apparent that the minimum and maximum envelopes hew very closely to the average usage cycles for both the weekend and weekday groups—indicating that, over this two-week period, the daily cycles are very consistent. If the envelope were wider, we could examine additional metrics, such as standard deviation, 1st and 3rd quartiles, or other percentiles (e.g., 10th and 90th or 1st and 99th), to get a sense for how consistent the usage cycle is from day to day.

Although not evident in this example, another frequent consideration when examining infrastructure usage is assessing whether there is an overall increase or decrease in usage over time. For a web-based software application, such changes could indicate growth or contraction of its user base, or reveal issues with the software implementation, such as memory leaks. The lack of such trends in this data is apparent upon visual inspection, but there are some simple quantitative techniques we can use to verify this theory.

One approach is to find the average usage for each day in the data set and determine whether there is a trend for these values within either the weekday or the weekend groupings. These daily averages are plotted in green in Figure 4-3. In this example, it is obvious to the eye that there is no trend; however, this can also be verified by fitting a line to the values in each set. We find that for both groupings, the slope is consistent with zero, indicating no change in the average daily usage over this two-week period. However, because of the cyclical nature of the usage pattern, we may be concerned that the long periods of low, constant usage might overwhelm any trends during the peak periods. To test this, we can calculate the average daily usage only during the peak periods, shown in Figure 4-3, in red. Once again, we find the slopes for each of the groupings to be consistent with zero, suggesting no obvious trend over this two-week period.

In a real-world scenario, we would urge some caution in interpreting these results. Two weeks represents a relatively short period over which to observe trends, particularly those associated with growth or contraction of a user base. Growth on the order of months or annual usage cycles, such as those that may be associated with e-commerce applications, may not be detectable in this short of a time span. To fully assess whether a business's CPU usage demonstrates long-term trends, we recommend continuing to collect a longer usage history. For this data, however, our analyses indicate that the

relevant patterns are (1) a distinct peak in usage during the late-night period and (2) differing usage patterns on weekends versus weekdays.

Scheduled Auto Scaling

Based on the two findings about this business' usage, we can immediately determine that there may be cost-savings achieved by scheduling resources to coincide with demand. Let's assume that the business wants to have sufficient resources available so that at any given time, its usage does not exceed 60% of available capacity (i.e., CPU). Let's further assume that this customer does not want fewer than two instances available at any time to provide high availability when there are unforeseen instance failures.

Over this two-week period, this business's maximum CPU usage tops out at 24 cores. If the business does not use any of Auto Scaling's scheduling capabilities, it would have to run 20 t2.medium instances, each having two CPU/instance, on AWS at all times to ensure it will not exceed its 60% threshold. Priced as an hourly on-demand resource in Northern California, this would lead to a weekly cost of about $230. With the use of Auto Scaling, however, we can potentially reduce the cost signficantly.

First, let's consider our finding that usage experienced a high peak at nighttime. Because of the reliably cyclical nature of the usage pattern, we can create a schedule wherein the business can toggle between a "high" and "low" usage setting of 20 and 6 instances, respectively, where the "low" setting is determined by the number of CPUs necessary to not exceed the 60% threshold during the constant daytime periods. By determining the typical start and end times for the peak, we created a single daily schedule that indicates whether to use either the "high" or the "low" setting for each hour of the day. We found that by implementing such a schedule, the business could achieve a weekly cost of around $150—*a savings of more than a third*. A schedule with even more settings could potentially achieve even further savings.

In the previous example, we use the same schedule for each day of the week. As we noted however, this business has significantly different usage patterns on weekdays than on weekends. By creating two different schedules (weekend versus weekday) the business can realize even further savings by utilizing fewer resources during the

slower weekday periods. For this particular usage pattern, the "low" setting would be the same for both groupings, while the "high" setting for the weekday grouping is 10 instances—half that of the weekend grouping.

Figure 4-5 illustrates how this setting would be implemented. The red line shows the binary schedule, including the difference between the weekday and weekend schedules. The blue line shows a more granular, multilevel schedule. The black line shows the actual usage.

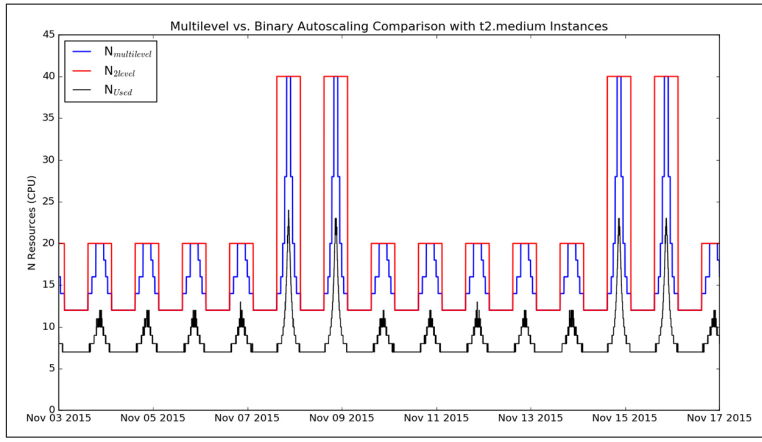

Figure 4-5. Comparison of binary and multilevel schedules, and actual usage. Credit: Arti Garg.

It may be tempting to create even more detailed schedules, perhaps one for each day of the week, but we emphasize caution before proceeding. As discussed above, these analyses are based on only two weeks of usage data, and we lack sufficient information to assess whether these patterns are unique to this particular time of year. However, if we can determine that the observed pattern is consistent with what might be expected from the company's business model, we can feel more confident in basing resource decisions upon it. The table below summarizes weekly cost estimates for a variety of schedules and instance types. It also includes pricing using dynamic Auto Scaling, which we'll explore next.

Dynamic Auto Scaling

As we can see, using AWS's Auto Scaling feature to schedule resources can lead to significant savings. At the same time, by using a mul-

tilevel schedule that hews closely to the observed usage pattern, a business also runs the risk that out-of-normal traffic can exceed the scheduled capacity. To avoid this, AWS offers a dynamic Auto Scaling capability that automatically adds or subtracts resources based upon predefined rules. For this usage pattern, we will consider a single scaling rule, though we note that AWS allows for multiple rules.

Let's consider a rule where at any given time, if the usage exceeds 70% of available capacity, AWS should add 10% of the existing capacity. As usage falls off, AWS should subtract 20% of existing capacity when current usage falls below 55%. When setting this scaling rule, we must also account for the finite amount of time needed for a new instance to "warm up" before becoming operational.

For this scenario, we use AWS's default setting of five minutes. Using this rule, we can step through our historical usage data to determine how many instances would be in use, launched, or deleted at any given time. Based on that output, we find that the average weekly cost for the period would be about $82, similar to the multilevel weekend + weekday schedule. This is not surprising when looking at historical data; our multilevel approach, which is optimized to the actual usage pattern, should produce similar results as dynamic, rules-based scaling.

This can be seen in Figure 4-6, which shows the number of CPUs that would be made available from a multilevel schedule (blue line) and from dynamic Auto Scaling (green line). Notably, the highest resource level launched by dynamic Auto Scaling is lower than what is made available by the multilevel schedule, but the cost impact is not significant since the peak lasts for only a short duration. The main advantage of dynamic Auto Scaling compared to the multilevel is that resources will still be added as needed even if the usage patterns deviate from historical behavior. For this usage pattern, this single rule is sufficient to provide substantial savings, though the optimal dynamic Auto Scaling setting will vary by each application's web traffic. For more complex usage patterns, we could consider and analyze a more complex sets of rules.

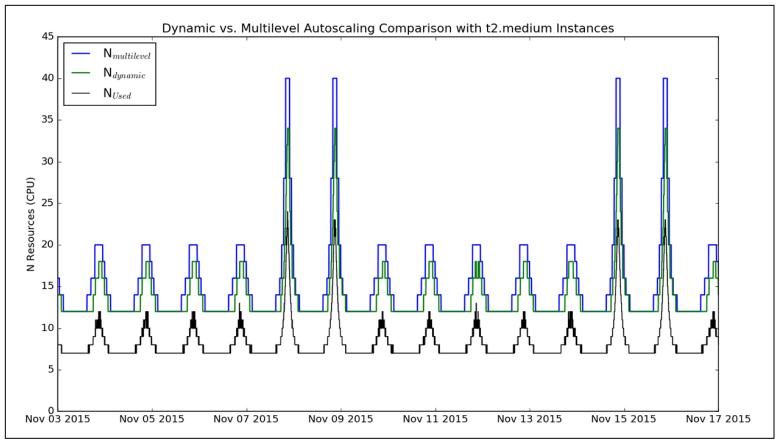

Figure 4-6. Comparison of dynamic and multilevel autoscaling. Credit: Arti Garg.

Assess Cost Savings First

Third-party-hosted, cloud-based infrastructure can offer businesses unprecedented advantages over private, on-site infrastructure. Cloud-based infrastructure can be deployed very quickly—saving months of procurement and set-up time. The use of dynamic resource scheduling, such as the capability enabled by AWS's Auto Scaling tool, can also help significantly reduce costs by right-sizing infrastructure. As we have seen, however, determining the optimal settings for realizing the most savings requires detailed analyses of historical infrastructure usage patterns. Since re-engineering is often required to make applications work with changing numbers of resources, it is important that businesses assess potential cost savings prior to implementing Auto Scaling.

Note: this example was put together by Datapipe's Data and Analytics Team.

CHAPTER 5

Machine Learning: Models and Training

In this chapter, Mikio Braun looks at how data-driven recommendations are computed, how they are brought into production, and how they can add real business value. He goes on to explore broader questions such as what the interface between data science and engineering looks like. Michelle Casbon then discusses the technology stack used to perform natural language processing at startup Idibon, and some of the challenges they've tackled, such as combining Spark functionality with their unique NLP-specific code. Next, Ben Lorica offers techniques to address overfitting, hyperparameter tuning, and model interpretability. Finally, Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin introduce local interpretable model-agnostic explanations (LIME), a technique to explain the predictions of any machine-learning classifier.

What Is Hardcore Data Science—in Practice?

By Mikio Braun

You can read this post on oreilly.com here.

During the past few years, data science has become widely accepted across a broad range of industries. Originally more of a research topic, data science has early roots in scientists' efforts to understand human intelligence and to create artificial intelligence; it has since also proven that it can add real business value.

As an example, we can look at the company I work for—Zalando, one of Europe's biggest fashion retailers—where data science is heavily used to provide data-driven recommendations, among other things. Recommendations are provided as a backend service in many places, including product pages, catalogue pages, newsletters, and for retargeting (Figure 5-1).

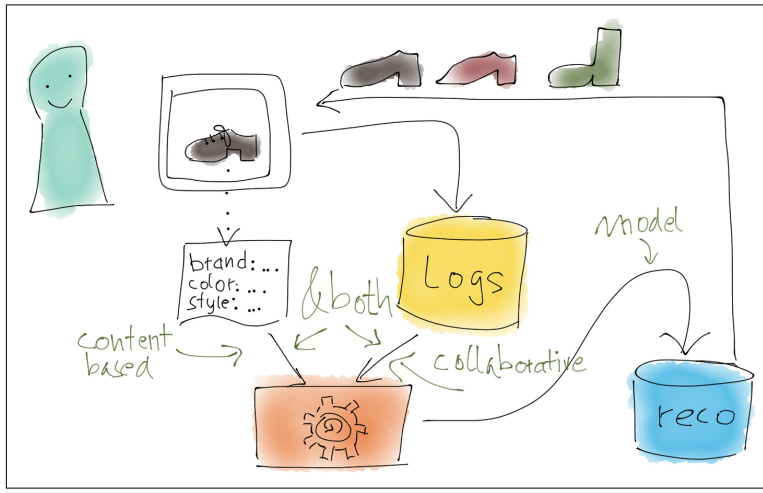

Figure 5-1. Data-driven recommendations. Credit: Mikio Braun.

Computing Recommendations

Naturally, there are many ways to compute data-driven recommendations. For so-called collaborative filtering, user actions like product views, actions on a wish list, and purchases are collected over the whole user base and then crunched to determine which items have similar user patterns. The beauty of this approach lies in the fact that the computer does not have to understand the items at all; the downside is that one has to have a lot of traffic to accumulate enough information about the items. Another approach only looks at the attributes of the items; for example, recommending other items from the same brand, or with similar colors. And of course, there are many ways to extend or combine these approaches.

Simpler methods consist of little more than counting to compute recommendations, but of course, there is practically no limit to the complexity of such methods. For example, for personalized recommendations, we have been working with *learning to rank* methods that learn individual rankings over item sets. Figure 5-2 shows the

cost function to optimize here, mostly to illustrate the level of complexity data science sometimes brings with it. The function itself uses a pairwise weighted ranking metric, with regularization terms. While being very mathematically precise, it is also very abstract. This approach can be used not only for recommendations in an ecommerce setting, but for all kinds of ranking problems, provided one has reasonable features.

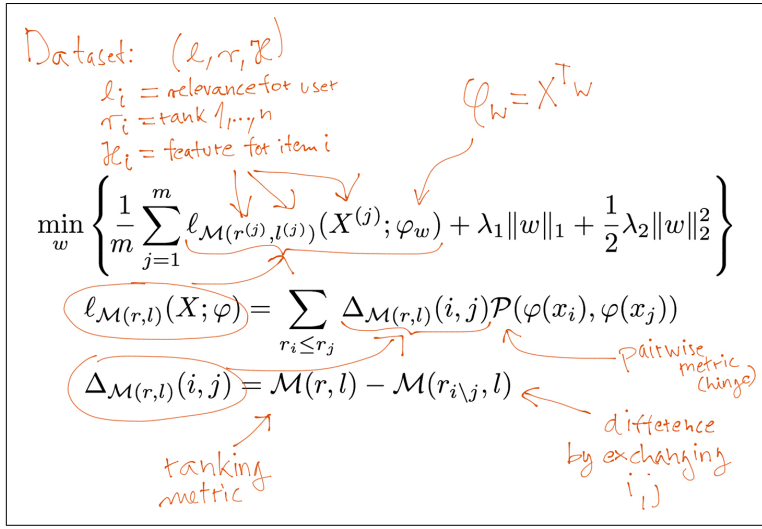

Figure 5-2. Data complexity. Credit: Antonio Freno, from "One-Pass Ranking Models for Low-Latency Product Recommendations," KDD 2015. Used with permission.

Bringing Mathematical Approaches into Industry

So, what does it take to bring a quite formal and mathematical approach like what we've described into production? And what does the interface between data science and engineering look like? What kind of organizational and team structures are best suited for this approach? These are all very relevant and reasonable questions, because they decide whether the investment in a data scientist or a whole team of data scientists will ultimately pay off.

In the remainder of this article, I will discuss a few of these aspects, based on my personal experience of having worked as a machine-learning researcher as well as having led teams of data scientists and engineers at Zalando.

Understanding Data Science Versus Production

Let's start by having a look at data science and backend production systems, and see what it takes to integrate these two systems (Figure 5-3).

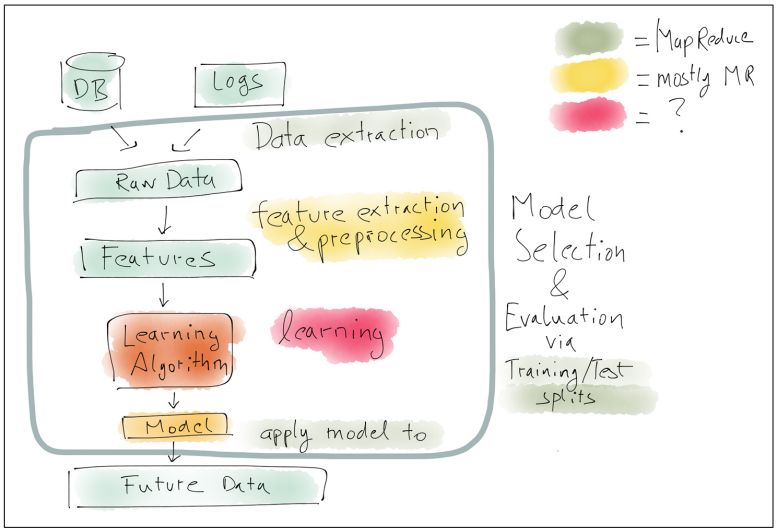

Figure 5-3. Data science and backend production systems. Credit: Mikio Braun.

The typical data science workflow looks like this: the first step is always identifying the problem and then gathering some data, which might come from a database or production logs. Depending on the data-readiness of your organization, this might already prove very difficult because you might have to first figure out who can give you access to the data, and then figure out who can give you the green light to actually get the data. Once the data is available, it's preprocessed to extract features, which are hopefully informative for the task to be solved. These features are fed to the learning algorithm, and the resulting model is evaluated on test data to get an estimate of how well it will work on future data.

This pipeline is usually done in a one-off fashion, often with the data scientist manually going through the individual steps using a programming language like Python, which comes with many libraries for data analysis and visualization. Depending on the size of the data, one may also use systems like Spark or Hadoop, but often the data scientist will start with a subset of the data first.

Why Start Small?

The main reason for starting small is that this is a process that is not done just once but will in fact be iterated *many times*. Data science projects are intrinsically exploratory, and to some amount, open-ended. The goal might be clear, but what data is available, or whether the available data is fit for the task at hand, is often unclear from the beginning. After all, choosing machine learning as an approach already means that one cannot simply write a program to solve the problem. Instead, one resorts to a data-driven approach.

This means that this pipeline is iterated and improved many times, trying out different features, different forms of preprocessing, different learning methods, or maybe even going back to the source and trying to add more data sources.

The whole process is inherently iterative, and often highly exploratory. Once the performance looks good, one is ready to try the method on real data. This brings us to production systems (Figure 5-4).

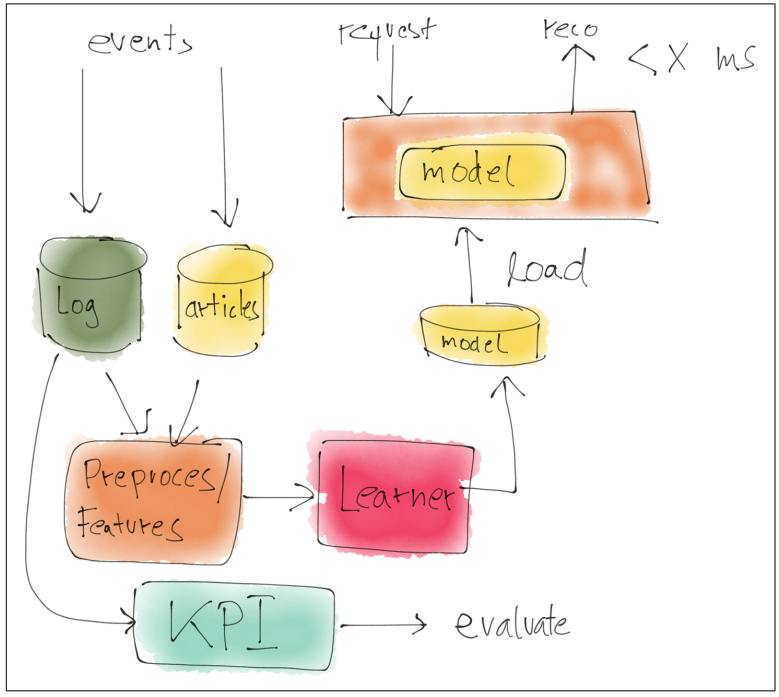

Figure 5-4. Production systems. Credit: Mikio Braun.

Distinguishing a Production System from Data Science

Probably the main difference between production systems and data science systems is that production systems are real-time systems that are continuously running. Data must be processed and models must be updated. The incoming events are also usually used for computing key performance indicators like click-through rates. The models are often retrained on available data every few hours and then loaded into the production system that serve the data via a REST interface, for example.

These systems are often written in programming languages like Java for performance and stability reasons.

If we put these two systems side by side, we get a picture like the Figure 5-5. On the top right, there is the data science side, characterized by using languages like Python or systems like Spark, but often with one-shot, manually triggered computations and iterations to optimize the system. The outcome of that is a model, which is essentially a bunch of numbers that describe the learned model. This model is then loaded by the production system. The production system is a more classical enterprise system, written in a language like Java, which is continually running.

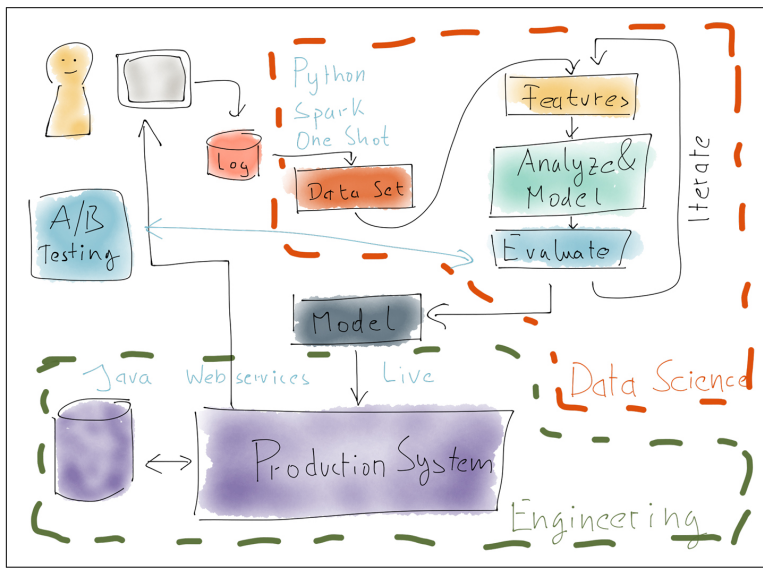

Figure 5-5. Production systems and data science systems. Credit: Mikio Braun.

The picture is a bit simplistic, of course. In reality, models have to be retrained, so some version of the processing pipeline must also be put into place on the production side to update the model every now and then.

Note that the A/B testing, which happens in the live system, mirrors the evaluation in the data science side. These are often not exactly comparable because it is hard to simulate the effect of a recommendation—for example, offline—without actually showing it to customers, but there should be a link in performance increase.

Finally, it's important to note that this whole system is not "done" once it is set up. Just as one first needs to iterate and refine the data analysis pipeline on the data science side, the whole live system also needs to be iterated as data distributions change and new possibilities for data analysis open up. To me, this "outer iteration" is the biggest challenge to get right—and also the most important one, because it will determine whether you can continually improve the system and secure your initial investment in data science.

Data Scientists and Developers: Modes of Collaboration

So far, we have focused on how systems typically look in production. There are variations in how far you want to go to make the production system really robust and efficient. Sometimes, it may suffice to directly deploy a model in Python, but the separation between the exploratory part and production part is usually there.

One of the big challenges you will face is how to organize the collaboration between data scientists and developers. "Data scientist" is still a somewhat new role, but the work they do differs enough from that of typical developers that you should expect some misunderstandings and difficulties in communication.

The work of data scientists is usually highly exploratory. Data science projects often start with a vague goal and some ideas of what kind of data is available and the methods that could be used, but very often, you have to try out ideas and get insights into your data. Data scientists write a lot of code, but much of this code is there to test out ideas and is not expected to be part of the final solution (Figure 5-6).

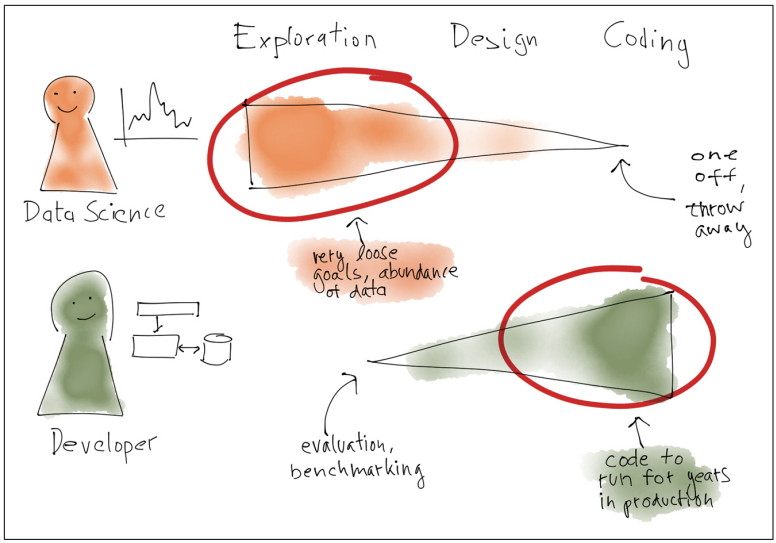

Figure 5-6. Data scientists and developers. Credit: Mikio Braun.

Developers, on the other hand, naturally have a much higher focus on coding. It is their goal to write a system and to build a program that has the required functionality. Developers sometimes also work in an exploratory fashion—building prototypes, proof of concepts, or performing benchmarks—but the main goal of their work is to write code.

These differences are also very apparent in the way the code evolves over time. Developers usually try to stick to a clearly defined process that involves creating branches for independent work streams, then having those reviewed and merged back into the main branch. People can work in parallel but need to incorporate approved merges into the main branch back into their branch, and so on. It is a whole process around making sure that the main branch evolves in an orderly fashion (Figure 5-7).

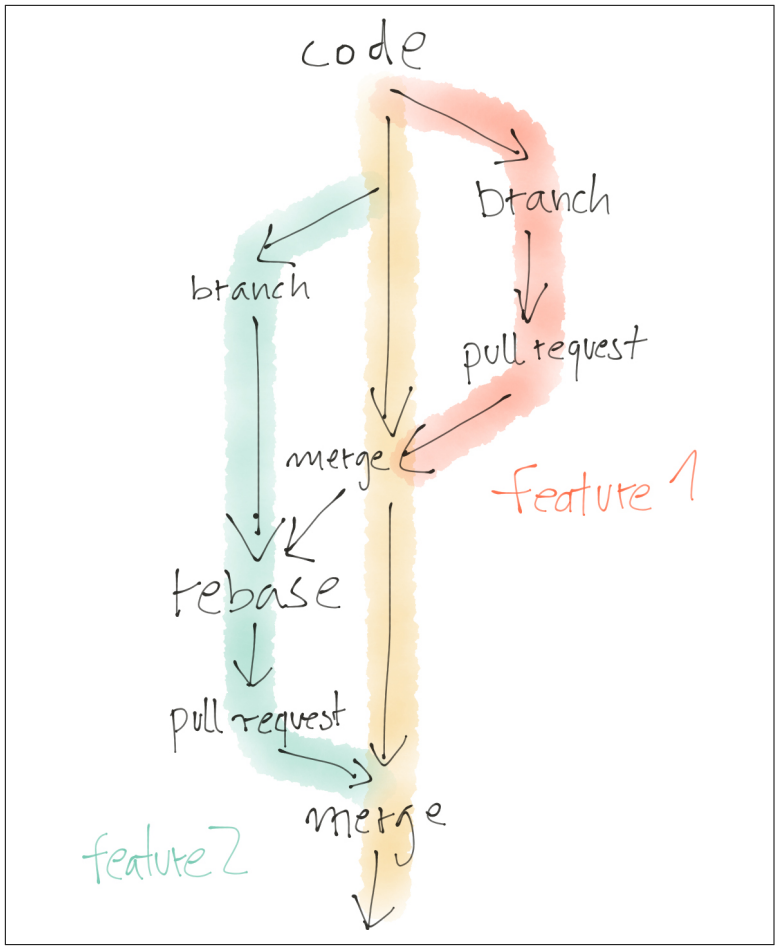

Figure 5-7. Branches for independent work streams. Credit: Mikio Braun.

While data scientists also write a lot of code, as I mentioned, it often serves to explore and try out ideas. So, you might come up with a version 1, which didn't quite do what you expected; then you have a version 2 that leads to versions 2.1 and 2.2 before you stop working on this approach, and go to versions 3 and 3.1. At this point you realize that if you take some ideas from 2.1 and 3.1, you can actually get a better solution, leading to versions 3.3 and 3.4, which is your final solution (Figure 5-8).

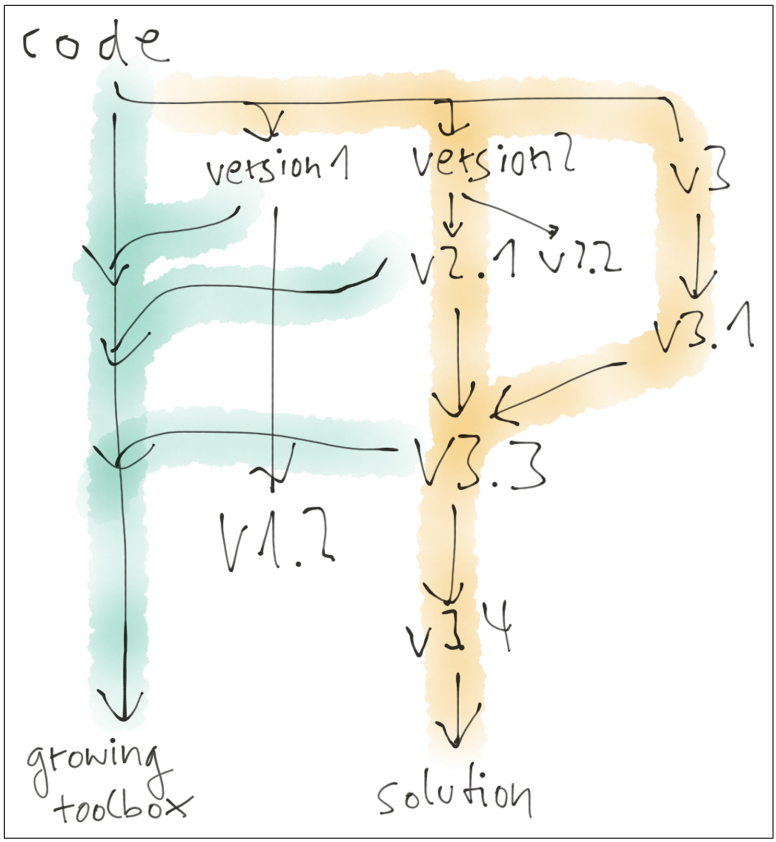

Figure 5-8. Data scientist process. Credit: Mikio Braun.

The interesting thing is that you would actually want to keep all those dead ends because you might need them at some later point. You might also put some of the things that worked well back into a growing toolbox—something like your own private machine-learning library—over time. While developers are interested in removing "dead code" (also because they know that you can always retrieve that later on, and they know how to do that quickly), data scientists often like to keep code, just in case.

These differences mean, in practice, that developers and data scientists often have problems working together. Standard software engineering practices don't really work out for data scientist's exploratory work mode because the goals are different. Introducing code reviews and an orderly branch, review, and merge-back workflow would just not work for data scientists and would slow them

down. Likewise, applying this exploratory mode to production systems also won't work.

So, how can we structure the collaboration to be most productive for both sides? A first reaction might be to keep the teams separate—for example, by completely separating the codebases and having data scientists work independently, producing a specification document as outcome that then needs to be implemented by the developers. This approach works, but it is also very slow and error-prone because reimplementing may introduce errors, especially if the developers are not familiar with data analysis algorithms, and performing the outer iterations to improve the overall system depends on developers having enough capacity to implement the data scientists' specifications (Figure 5-9).

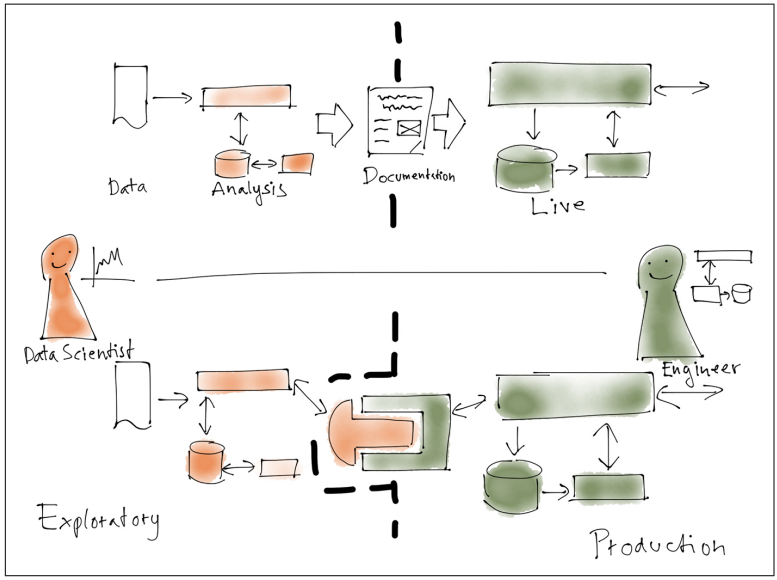

Figure 5-9. Keep the teams separate. Credit: Mikio Braun.

Luckily, many data scientists are actually interested in becoming better software engineers, and the other way round, so we have started to experiment with modes of collaboration that are a bit more direct and help to speed up the process.

For example, data science and developer code bases could still be separate, but there is a part of the production system that has a clearly identified interface into which the data scientists can hook

their methods. The code that communicates with the production system obviously needs to follow stricter software development practices, but would still be in the responsibility of the data scientists. That way, they can quickly iterate internally, but also with the production system (Figure 5-10).

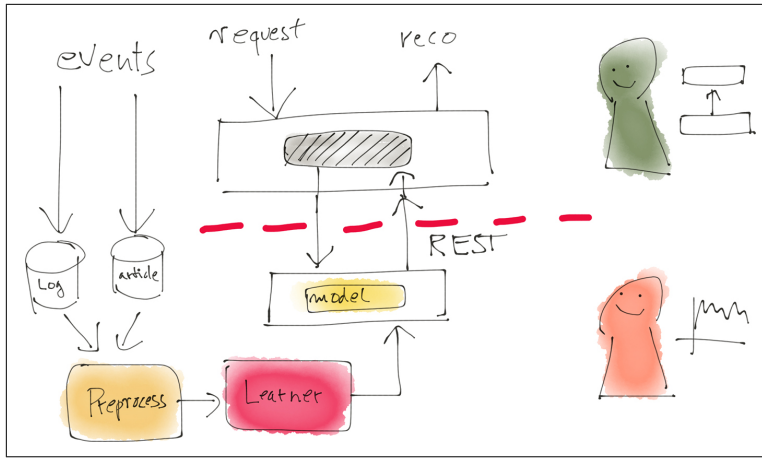

Figure 5-10. Experiment with modes of collaboration. Credit: Mikio Braun.

One concrete realization of that architecture pattern is to take a microservice approach and have the ability in the production system to query a microservice owned by the data scientists for recommendations. That way, the whole pipeline used in the data scientist's offline analysis can be repurposed to also perform A/B tests or even go in production without developers having to reimplement everything. This also puts more emphasis on the software engineering skills of the data scientists, but we are increasingly seeing more people with that skill set. In fact, we have lately changed the title of data scientists at Zalando to "research engineer (data science)" to reflect the fact.

With an approach like this, data scientists can move fast, iterate on offline data, and iterate in a production setting—and the whole team can migrate stable data analysis solutions into the production system over time.

Constantly Adapt and Improve

So, I've outlined the typical anatomy of an architecture to bring data science into production. The key concept to understand is that such a system needs to constantly adapt and improve (as almost all data-driven projects working with live data). Being able to iterate quickly, trying out new methods and testing the results on live data in A/B-tests, is most important.

In my experience, this cannot be achieved by keeping data scientists and developers separate. At the same time, it's important to acknowledge that their working modes are different because they follow different goals—data scientists are more exploratory and developers are more focused on building software and systems. By allowing both sides to work in a fashion that best suits these goals and defining a clear interface between them, it is possible to integrate the two sides so that new methods can be quickly tried out. This requires more software engineering skills from data scientists, or at least support by engineers who are able to bridge both worlds.

Training and Serving NLP Models Using Spark MLlib

By Michelle Casbon

You can read this post on oreilly.com here.

Identifying critical information amidst a sea of unstructured data and customizing real-time human interaction are a couple of examples of how clients utilize our technology at Idibon, a San Francisco startup focusing on natural language processing (NLP). The machine-learning libraries in Spark ML and MLlib have enabled us to create an adaptive machine intelligence environment that analyzes text in any language, at a scale far surpassing the number of words per second in the Twitter firehose.

Our engineering team has built a platform that trains and serves thousands of NLP models, which function in a distributed environment. This allows us to scale out quickly and provide thousands of predictions per second for many clients simultaneously. In this post, we'll explore the types of problems we're working to resolve, the processes we follow, and the technology stack we use. This should be

helpful for anyone looking to build out or improve their own NLP pipelines.

Constructing Predictive Models with Spark

Our clients are companies that need to automatically classify documents or extract information from them. This can take many diverse forms, including social media analytics, message categorization and routing of customer communications, newswire monitoring, risk scoring, and automating inefficient data entry processes. All of these tasks share a commonality: *the construction of predictive models, trained on features extracted from raw text*. This process of creating NLP models represents a unique and challenging use case for the tools provided by Spark (Figure 5-11).

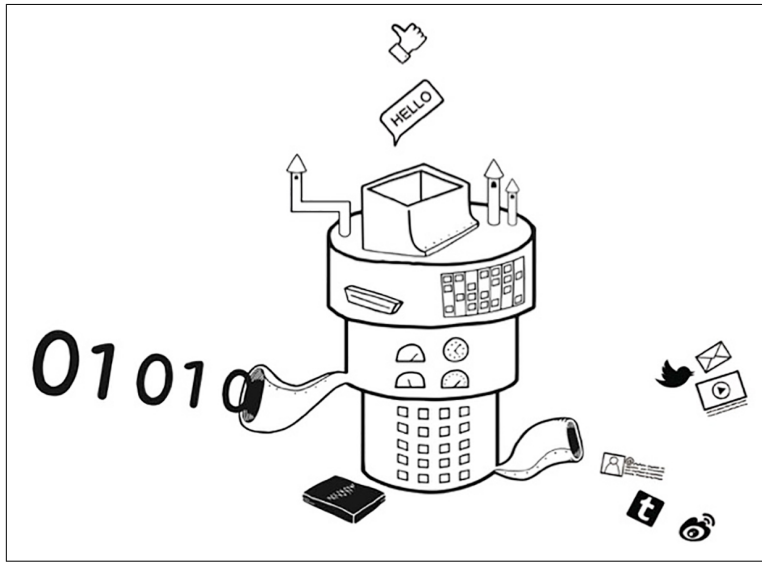

Figure 5-11. Creating NLP models. Credit: Idibon.

The Process of Building a Machine-Learning Product

A machine-learning product can be broken down into three conceptual pieces: *the prediction* itself, *the models* that provide the prediction, and *the data set* used to train the models (Figure 5-12).

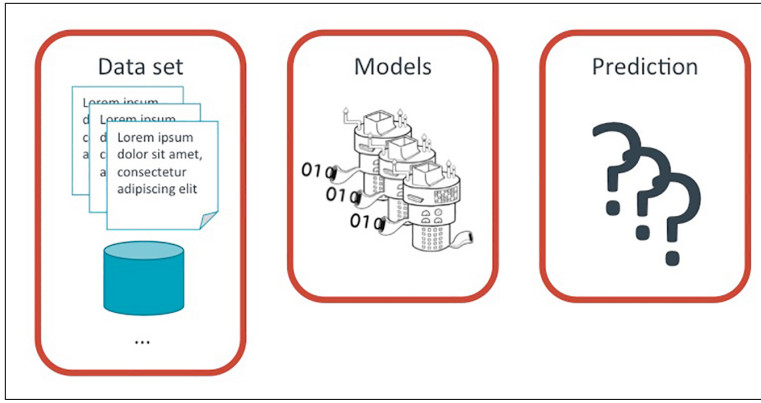

Figure 5-12. Building a machine-learning product. Credit: Michelle Casbon.

Prediction

In our experience, it's best to begin with business questions and use them to drive the selection of data sets, rather than having data sets themselves drive project goals. If you do begin with a data set, it's important to connect data exploration with critical business needs as quickly as possible. With the right questions in place, it becomes straightforward to choose useful classifications, which is what a prediction ultimately provides.

Data set

Once the predictions are defined, it becomes fairly clear which data sets would be most useful. It is important to verify that the data you have access to can support the questions you are trying to answer.

Model training

Having established the task at hand and the data to be used, it's time to worry about the models. In order to generate models that are accurate, we need training data, which is often generated by humans. These humans may be experts within a company or consulting firm; or in many cases, they may be part of a network of analysts.

Additionally, many tasks can be done efficiently and inexpensively by using a crowdsourcing platform like CrowdFlower. We like the platform because it categorizes workers based on specific areas of

expertise, which is particularly useful for working with languages other than English.

All of these types of workers submit annotations for specific portions of the data set in order to generate training data. The training data is what you'll use to make predictions on new or remaining parts of the data set. Based on these predictions, you can make decisions about the *next* set of data to send to annotators. The point here is to make the best models with the fewest human judgments. You continue iterating between model training, evaluation, and annotation—getting higher accuracy with each iteration. We refer to this process as *adaptive learning*, which is a quick and cost-effective means of producing highly accurate predictions.

Operationalization

To support the adaptive learning process, we built a platform that *automates* as much as possible. Having components that *auto-scale without our intervention* is key to supporting a real-time API with fluctuating client requests. A few of the tougher scalability challenges we've addressed include:

- Document storage
- Serving up thousands of *individual* predictions per second
- Support for continuous training, which involves automatically generating updated models whenever the set of training data or model parameters change
- Hyperparameter optimization for generating the most performant models

We do this by using a combination of components within the AWS stack, such as Elastic Load Balancing, Auto Scaling Groups, RDS, and ElastiCache. There are also a number of metrics that we monitor within New Relic and Datadog, which alert us before things go terribly awry.

Figure 5-13 a high-level diagram of the main tools in our infrastructure.

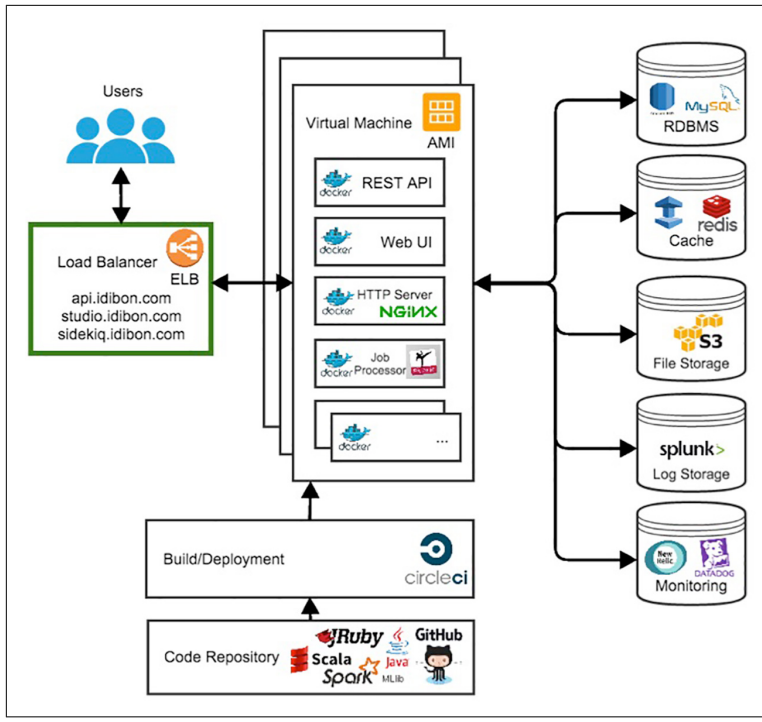

Figure 5-13. Main tools. Credit: Michelle Casbon.

Spark's Role

A core component of our machine-learning capabilities is the optimization functionality within Spark ML and MLlib. Making use of these for NLP purposes involves the addition of a *persistence layer* that we refer to as IdiML. This allows us to utilize Spark for individual predictions, rather than its most common usage as a platform for processing large amounts of data all at once.

What are we using Spark for?

At a more detailed level, there are three main components of an NLP pipeline (Figure 5-14):

1. **Feature extraction**, in which text is converted into a numerical format appropriate for statistical modeling.
2. **Training**, in which models are generated based on the classifications provided for each feature vector.

3. **Prediction**, in which the trained models are used to generate a classification for new, unseen text.

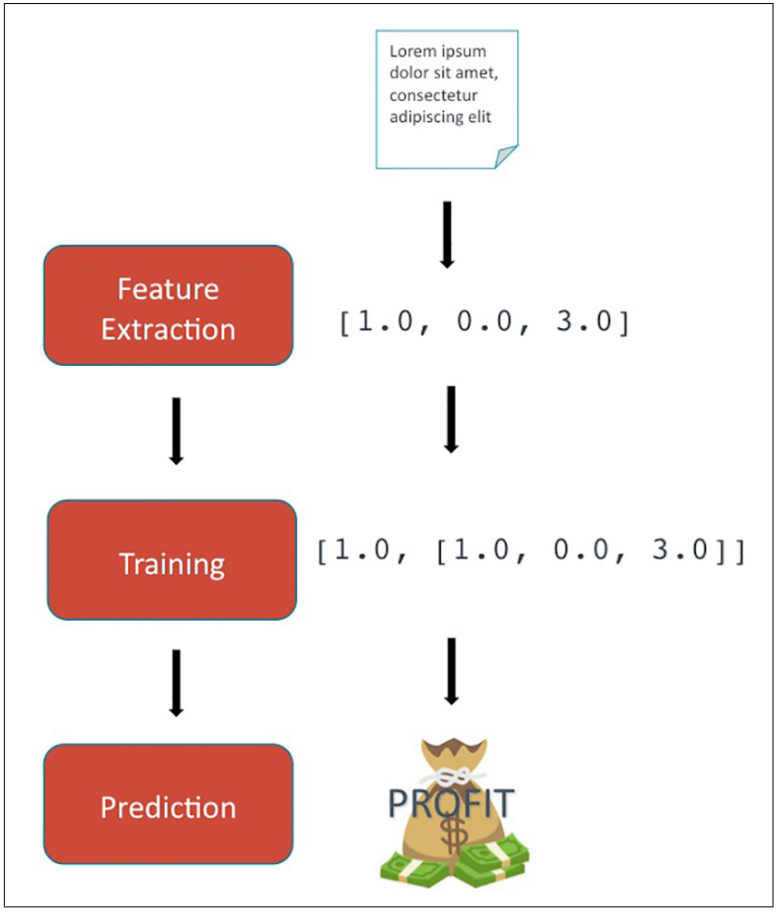

Figure 5-14. Core components of our machine learning capabilities. Credit: Michelle Casbon.

A simple example of each component is described next:

Feature extraction

In the feature extraction phase, text-based data is transformed into numerical data in the form of a *feature vector*. This vector represents the unique characteristics of the text and can be generated by any sequence of mathematical transformations. Our system was built to easily accommodate additional feature types such as features derived

from deep learning, but for simplicity's sake, we'll consider a basic feature pipeline example (Figure 5-15):

1. Input: a single document, consisting of content and perhaps metadata.
2. Content extraction: isolates the portion of the input that we're interested in, which is usually the content itself.
3. Tokenization: separates the text into individual words. In English, a token is more or less a string of characters with whitespace or punctuation around them, but in other languages like Chinese or Japanese, you need to probabilistically determine what a "word" is.
4. Ngrams: generates sets of word sequences of length *n*. Bigrams and trigrams are frequently used.
5. Feature lookup: assigns an arbitrary numerical index value to each unique feature, resulting in a vector of integers. This feature index is stored for later use during prediction.
6. Output: a numerical feature vector in the form of Spark MLlib's `Vector` data type (`org.apache.spark.mllib.linalg.Vector`).

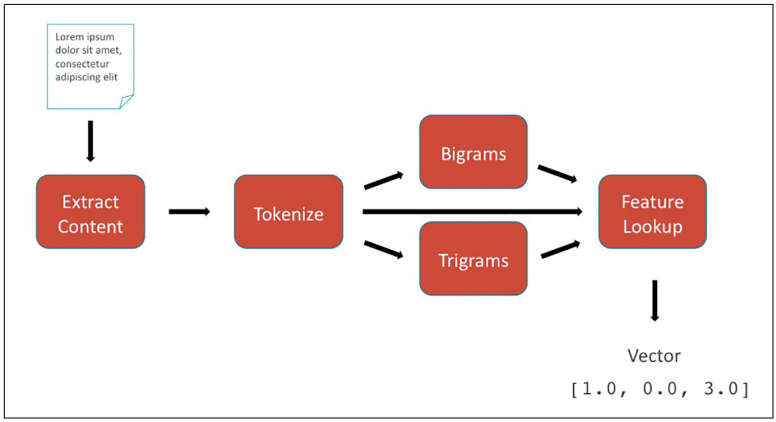

Figure 5-15. Feature extraction pipeline. Credit: Michelle Casbon.

Training

During the training phase, a classification is appended to the feature vector. In Spark, this is represented by the `LabeledPoint` data type. In a binary classifier, the classification is either true or false (1.0 or 0.0) (Figure 5-16):

1. Input: numerical feature `Vectors`.

2. A `LabeledPoint` is created, consisting of the feature vector and its classification. This classification was generated by a human earlier in the project lifecycle.
3. The set of `LabeledPoints` representing the full set of training data is sent to the `LogisticRegressionWithLBFGS` function in MLlib, which fits a model based on the given feature vectors and associated classifications.
4. Output: a `LogisticRegressionModel`.

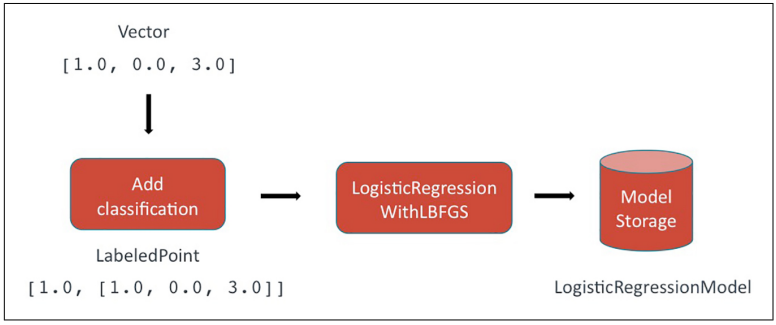

Figure 5-16. Training phase. Credit: Michelle Casbon.

Prediction

At prediction time, the models generated during training are used to provide a classification for the new piece of text. A *confidence interval* of 0–1 indicates the strength of the model's confidence in the prediction. The higher the confidence, the more certain the model is. The following components encompass the prediction process (Figure 5-17):

1. Input: unseen document in the same domain as the data used for training.
2. The same featurization pipeline is applied to the unseen text. The feature index generated during training is used here as a lookup. This results in a feature vector in the same feature space as the data used for training.
3. The trained model is retrieved.
4. The feature `Vector` is sent to the model, and a classification is returned as a prediction.
5. The classification is interpreted in the context of the specific model used, which is then returned to the user.

6. Output: a predicted classification for the unseen data and a corresponding confidence interval.

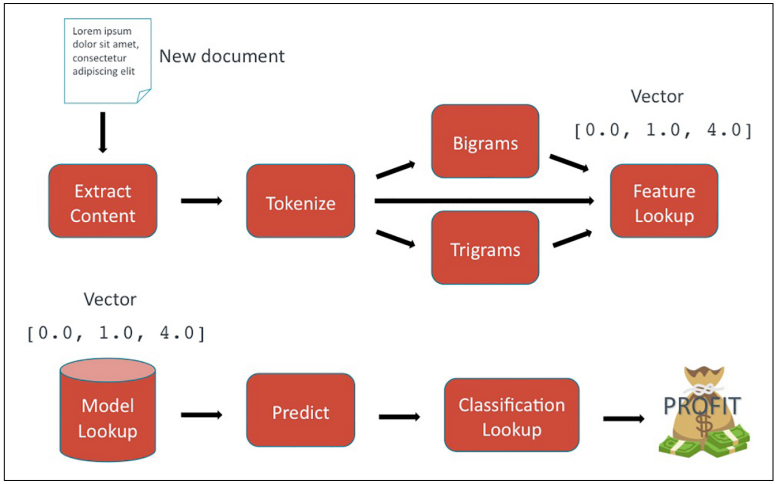

Figure 5-17. The prediction process. Credit: Michelle Casbon.

Prediction data types

In typical Spark ML applications, predictions are mainly generated using `RDDs` and `DataFrames`: the application loads document data into one column, and MLlib places the results of its prediction in another. Like all Spark applications, these prediction jobs may be distributed across a cluster of servers to efficiently process petabytes of data. However, our most demanding use case is exactly the opposite of big data: often, we must analyze a single, short piece of text and return results as quickly as possible, ideally within a millisecond.

Unsurprisingly, `DataFrames` are not optimized for this use case, and our initial `DataFrame`-based prototypes fell short of this requirement.

Fortunately for us, MLlib is implemented using an efficient linear algebra library, and all of the algorithms we planned to use included internal methods that generated predictions using single `Vector` objects without any added overhead. These methods looked perfect for our use case, so we designed IdiML to be extremely efficient at converting single documents to single `Vectors` so that we could use Spark MLlib's internal `Vector`-based prediction methods.

For a single prediction, we observed speed improvements of up to two orders of magnitude by working with Spark MLlib's `Vector` type as opposed to `RDDs`. The speed differences between the two data types are most pronounced among smaller batch sizes. This makes sense considering that `RDDs` were designed for processing large amounts of data. In a real-time web server context such as ours, small batch sizes are by far the most common scenario. Since distributed processing is already built into our web server and load balancer, the distributed components of core Spark are unnecessary for the small-data context of individual predictions. As we learned during the development of IdiML and have shown in Figure 5-18, Spark MLlib is an incredibly useful and performant machine-learning library for low-latency and real-time applications. Even the worst-case IdiML performance is capable of performing sentiment analysis on every Tweet written, in real time, from a mid-range consumer laptop (Figure 5-19).

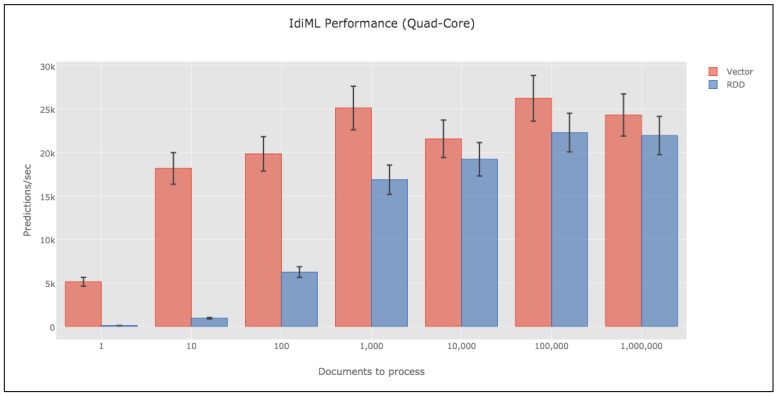

Figure 5-18. Measurements performed on a mid-2014 15-inch MacBook Pro Retina. The large disparity in single-document performance is due to the inability of the test to take advantage of the multiple cores. Credit: Michelle Casbon.

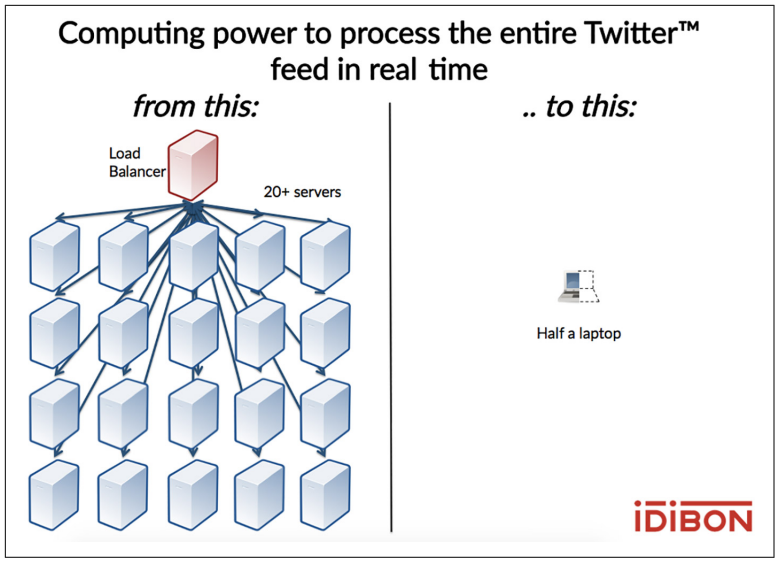

Figure 5-19. Processing power of IdiML. Credit: Rob Munro. Used with permission.

Fitting It Into Our Existing Platform with IdiML

In order to provide the most accurate models possible, we want to be able to support different types of machine-learning libraries. Spark has a unique way of doing things, so we want to insulate our main code base from any idiosyncrasies. This is referred to as a *persistence layer* (IdiML), which allows us to combine Spark functionality with NLP-specific code that we've written ourselves. For example, during hyperparameter tuning we can train models by combining components from both Spark and our own libraries. This allows us to automatically choose the implementation that performs best for each model, rather than having to decide on just one for all models (Figure 5-20).

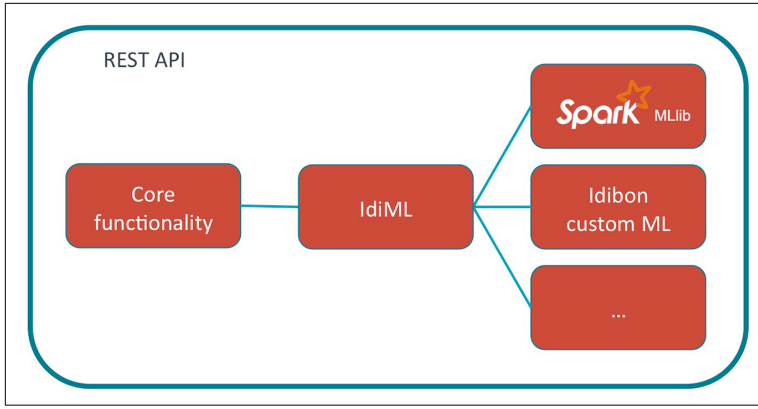

Figure 5-20. Persistence layer. Credit: Michelle Casbon.

Why a persistence layer?

The use of a persistence layer allows us to operationalize the training and serving of many thousands of models. Here's what IdiML provides us with:

- **A means of storing the parameters used during training.** This is necessary in order to return the corresponding prediction.
- **The ability to version control every part of the pipeline.** This enables us to support backward compatibility after making updates to the code base. Versioning also refers to the ability to recall and support previous iterations of models during a project's lifecycle.
- **The ability to automatically choose the best algorithm for each model.** During hyperparameter tuning, implementations from different machine-learning libraries are used in various combinations and the results evaluated.
- **The ability to rapidly incorporate new NLP features** by standardizing the developer-facing components. This provides an insulation layer that makes it unnecessary for our feature engineers and data scientists to learn how to interact with a new tool.
- **The ability to deploy in any environment.** We are currently using Docker containers on EC2 instances, but our architecture means that we can also take advantage of the *burst capabilities* that services such as Amazon Lambda provide.

- **A single save and load framework** based on generic `InputStreams` & `OutputStreams`, which frees us from the requirement of reading and writing to and from disk.
- **A logging abstraction in the form of slf4j**, which insulates us from being tied to any particular framework.

Faster, Flexible Performant Systems

NLP differs from other forms of machine learning because it operates directly on human-generated data. This is often messier than machine-generated data because language is inherently ambiguous, which results in highly variable interpretability—even among humans. Our goal is to automate as much of the NLP pipeline as possible so that resources are used more efficiently: machines help humans, help machines, help humans. To accomplish this across language barriers, we're using tools such as Spark to build performant systems that are faster and more flexible than ever before.

Three Ideas to Add to Your Data Science Toolkit

By Ben Lorica

You can read this post on oreilly.com here.

I'm always on the lookout for ideas that can improve how I tackle data analysis projects. I particularly favor approaches that translate to tools I can use *repeatedly*. Most of the time, I find these tools on my own—by trial and error—or by consulting other practitioners. I also have an affinity for academics and academic research, and I often tweet about research papers that I come across and am intrigued by. Often, academic research results don't immediately translate to what I do, but I recently came across ideas from several research projects that are worth sharing with a wider audience.

The collection of ideas I've presented in this post address problems that come up frequently. In my mind, these ideas also reinforce the notion of data science as comprising data pipelines, not just machine-learning algorithms. These ideas also have implications for engineers trying to build artificial intelligence (AI) applications.

Use a Reusable Holdout Method to Avoid Overfitting During Interactive Data Analysis

Overfitting is a well-known problem in statistics and machine learning. Techniques like the holdout method, bootstrap, and cross-validation are used to avoid overfitting in the context of static data analysis. The widely used holdout method involves splitting an underlying data set into two separate sets. But practitioners (and I'm including myself here) often forget something important when applying the classic holdout method: in theory, the corresponding holdout set is accessible only once (as illustrated in Figure 5-21).

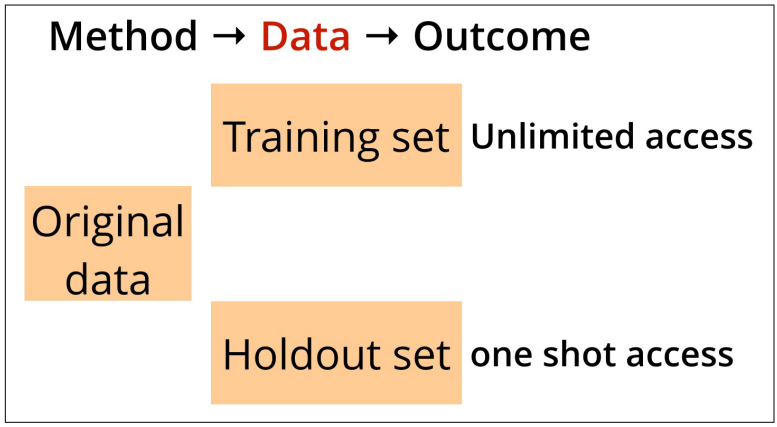

Figure 5-21. Static data analysis. Credit: Ben Lorica.

In reality, most data science projects are *interactive* in nature. Data scientists iterate many times and revise their methods or algorithms based on previous results. This frequently leads to overfitting because in many situations, the same holdout set is used multiple times (as illustrated in Figure 5-22).

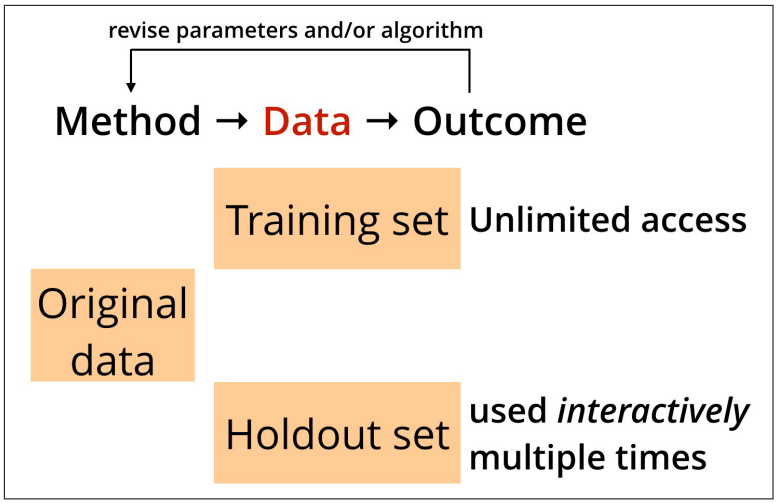

Figure 5-22. Interactive data analysis. Credit: Ben Lorica.

To address this problem, a team of researchers devised *reusable holdout* methods by drawing from ideas in differential privacy. By addressing overfitting, their methods can increase the reliability of data products, particularly as more intelligent applications get deployed in critical situations. The good news is that the solutions they came up with are accessible to data scientists and do not require an understanding of differential privacy. In a presentation at Hardcore Data Science in San Jose, Moritz Hardt of Google (one of the researchers) described their proposed Thresholdout method using the following Python code:

```
from numpy import *
def Thresholdout(sample, holdout, q):
    # function q is what you're "testing" - e.g., model loss
    sample_mean = mean([q(x)  for x in sample])
    holdout_mean = mean([q(x)  for x in holdout])
    sigma = 1.0 / sqrt(len(sample))
    threshold = 3.0*sigma
    if (abs(sample_mean - holdout_mean)
           < random.normal(threshold, sigma) ):
        # q does not overfit: your "training estimate" is good
        return sample_mean
    else:
        # q overfits (you may have overfit using your
        # training data)
        return holdout_mean + random.normal(0, sigma)
```

Details of their Thresholdout and other methods can be found in this paper and Hardt's blog posts here and here. I also recommend a recent paper on blind analysis—a related data-perturbation method used in physics that may soon find its way into other disciplines.

Use Random Search for Black-Box Parameter Tuning

Most data science projects involve pipelines that involve "knobs" (or hyperparameters) that need to be tuned appropriately, usually on a trial-and-error basis. These hyperparameters typically come with a particular machine-learning method (network depth and architecture, window size, etc.), but they can also involve aspects that affect data preparation and other steps in a data pipeline.

With the growing number of applications of machine-learning pipelines, hyperparameter tuning has become a subject of many research papers (and even commercial products). Many of the results are based on Bayesian optimization and related techniques.

Practicing data scientists need not rush to learn Bayesian optimization. Recent blog posts (here and here) by Ben Recht of UC Berkeley highlighted research that indicates when it comes to black-box parameter tuning, simple random search is actually quite competitive with more advanced methods. And efforts are underway to accelerate random search for particular workloads.

Explain Your Black-Box Models Using Local Approximations

In certain domains (including health, consumer finance, and security), model interpretability is often a requirement. This comes at a time when black-box models—including deep learning and other algorithms, and even ensembles of models—are all the rage. With the current interest in AI, it is important to point out that black-box techniques will only be deployed in certain application domains if tools to make them more interpretable are developed.

A recent paper by Marco Tulio Ribeiro and colleagues hints at a method that can make such models easier to explain. The idea proposed in this paper is to use a series of interpretable, *locally faithful approximations*. These are interpretable, local models that approximate how the original model behaves in the vicinity of the instance being predicted. The researchers observed that although a model

may be too complex to explain globally, providing an explanation that is locally faithful is often sufficient.

A recent presentation illustrated the utility of the researchers' approach. One of the co-authors of the paper, Carlos Guestrin, demonstrated an implementation of a related method that helped debug a deep neural network used in a computer vision application.

Related Resources

- "Introduction to Local Interpretable Model-Agnostic Explanations (LIME)"
- "6 reasons why I like KeystoneML"—a conversation with Ben Recht
- "The evolution of GraphLab"—a conversation with Carlos Guestrin
- The Deep Learning Video Collection: 2016—a collection of talks at Strata + Hadoop World

Introduction to Local Interpretable Model-Agnostic Explanations (LIME)

By Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin

You can read this post on oreilly.com here.

Machine learning is at the core of many recent advances in science and technology. With computers beating professionals in games like Go, many people have started asking if machines would also make for better drivers or even better doctors.

In many applications of machine learning, users are asked to trust a model to help them make decisions. A doctor will certainly not operate on a patient simply because "the model said so." Even in lower-stakes situations, such as when choosing a movie to watch from Netflix, a certain measure of trust is required before we surrender hours of our time based on a model. Despite the fact that many machine-learning models are black boxes, understanding the rationale behind the model's predictions would certainly help users decide when to trust or not to trust their predictions. An example is shown in Figure 5-23, in which a model predicts that a certain patient has the flu. The prediction is then explained by an "explainer" that high-

lights the symptoms that are most important to the model. With this information about the rationale behind the model, the doctor is now empowered to trust the model—or not.

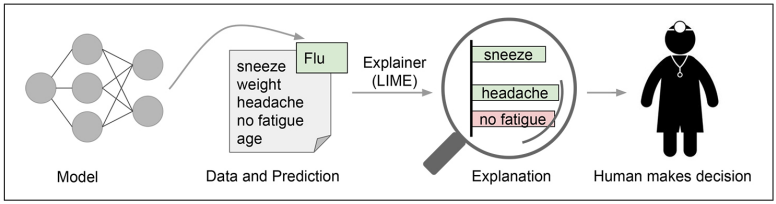

Figure 5-23. Explaining individual predictions to a human decision-maker. Credit: Marco Tulio Ribeiro.

In a sense, every time an engineer uploads a machine-learning model to production, the engineer is implicitly trusting that the model will make sensible predictions. Such assessment is usually done by looking at held-out accuracy or some other aggregate measure. However, as anyone who has ever used machine learning in a real application can attest, such metrics can be very misleading. Sometimes data that shouldn't be available accidentally leaks into the training and into the held-out data (e.g., looking into the future). Sometimes the model makes mistakes that are too embarrassing to be acceptable. These and many other tricky problems indicate that understanding the model's predictions can be an additional useful tool when deciding if a model is trustworthy or not, because humans often have good intuition and business intelligence that is hard to capture in evaluation metrics. Assuming a "pick step" in which certain representative predictions are selected to be explained to the human would make the process similar to the one illustrated in Figure 5-24.

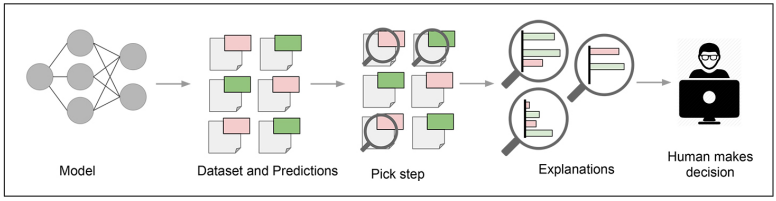

Figure 5-24. Explaining a model to a human decision-maker. Credit: Marco Tulio Ribeiro.

In "'Why Should I Trust You?' Explaining the Predictions of Any Classifier," a joint work by Marco Tulio Ribeiro, Sameer Singh, and

Carlos Guestrin (which appeared at ACM's Conference on Knowledge Discovery and Data Mining -- KDD2016), we explore precisely the question of trust and explanations. We propose local interpretable model-agnostic explanations (LIME), a technique to explain the predictions of *any* machine-learning classifier, and evaluate its usefulness in various tasks related to trust.

Intuition Behind LIME

Because we want to be model-agnostic, what we can do to learn the behavior of the underlying model is to perturb the input and see how the predictions change. This turns out to be a benefit in terms of interpretability, because we can perturb the input by changing components that make sense to humans (e.g., words or parts of an image), even if the model is using much more complicated components as features (e.g., word embeddings).

We generate an explanation by approximating the underlying model by an interpretable one (such as a linear model with only a few nonzero coefficients), learned on perturbations of the original instance (e.g., removing words or hiding parts of the image). The key intuition behind LIME is that it is much easier to approximate a black-box model by a simple model *locally* (in the neighborhood of the prediction we want to explain), as opposed to trying to approximate a model globally. This is done by weighting the perturbed images by their similarity to the instance we want to explain. Going back to our example of a flu prediction, the three highlighted symptoms may be a faithful approximation of the black-box model for patients who look like the one being inspected, but they probably do not represent how the model behaves for all patients.

See Figure 5-25 for an example of how LIME works for image classification. Imagine we want to explain a classifier that predicts how likely it is for the image to contain a tree frog. We take the image on the left and divide it into interpretable components (contiguous superpixels).

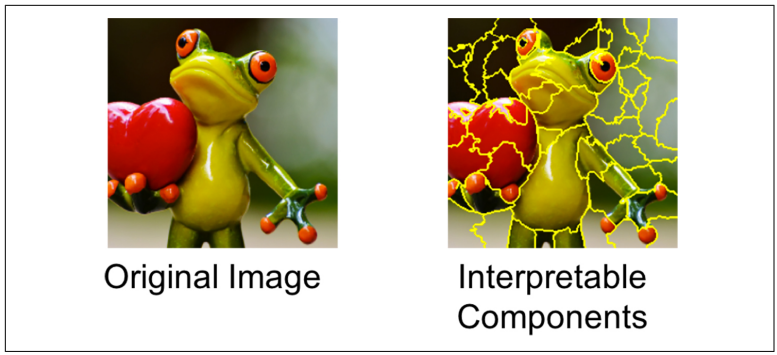

Figure 5-25. Transforming an image into interpretable components. Credit: Marco Tulio Ribeiro, Pixabay.

As illustrated in Figure 5-26, we then generate a data set of perturbed instances by turning some of the interpretable components "off" (in this case, making them gray). For each perturbed instance, we get the probability that a tree frog is in the image according to the model. We then learn a simple (linear) model on this data set, which is locally weighted—that is, we care more about making mistakes in perturbed instances that are more similar to the original image. In the end, we present the superpixels with highest positive weights as an explanation, graying out everything else.

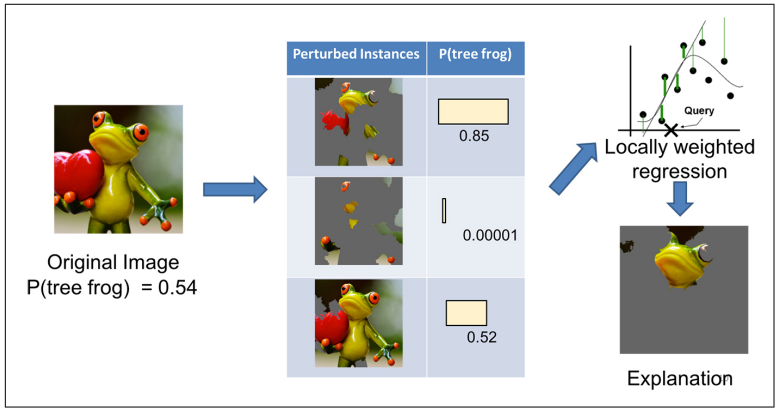

Figure 5-26. Explaining a prediction with LIME. Credit: Marco Tulio Ribeiro, Pixabay.

Examples

We used LIME to explain a myriad of classifiers (such as random forests, support vector machines (SVM), and neural networks) in the text and image domains. Here are a few examples of the generated explanations.

First, an example from text classification. The famous 20 newsgroups data set is a benchmark in the field, and has been used to compare different models in several papers. We take two classes that are hard to distinguish because they share many words: Christianity and atheism. Training a random forest with 500 trees, we get a test set accuracy of 92.4%, which is surprisingly high. If accuracy was our only measure of trust, we would definitely trust this classifier. However, let's look at an explanation in Figure 5-27 for an arbitrary instance in the test set (a one-liner in Python with our open source package):

```
exp = explainer.explain_instance(test_example, classifier
.predict_proba, num_features=6)
```

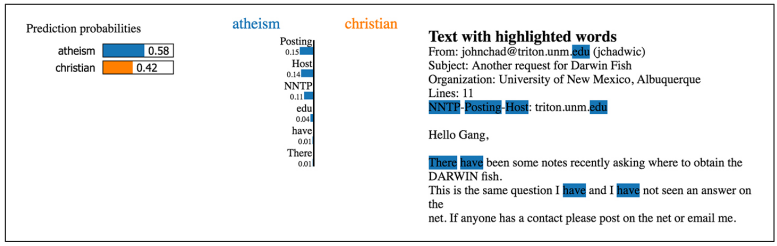

Figure 5-27. Explanation for a prediction in the 20 newsgroups data set. Credit: Marco Tulio Ribeiro.

This is a case in which the classifier predicts the instance correctly, but for the wrong reasons. Additional exploration shows us that the word "posting" (part of the email header) appears in 21.6% of the examples in the training set but only two times in the class "Christianity." This is also the case in the test set, where the word appears in almost 20% of the examples but only twice in "Christianity." This kind of artifact in the data set makes the problem much easier than it is in the real world, where we wouldn't expect such patterns to occur. These insights become easy once you understand what the models are actually doing, which in turn leads to models that generalize much better.

As a second example, we explain Google's Inception neural network on arbitrary images. In this case, illustrated in Figure 5-28, the classifier predicts "tree frog" as the most likely class, followed by "pool table" and "balloon" with lower probabilities. The explanation reveals that the classifier primarily focuses on the frog's face as an explanation for the predicted class. It also sheds light on why "pool table" has nonzero probability: the frog's hands and eyes bear a resemblance to billiard balls, especially on a green background. Similarly, the heart bears a resemblance to a red balloon.

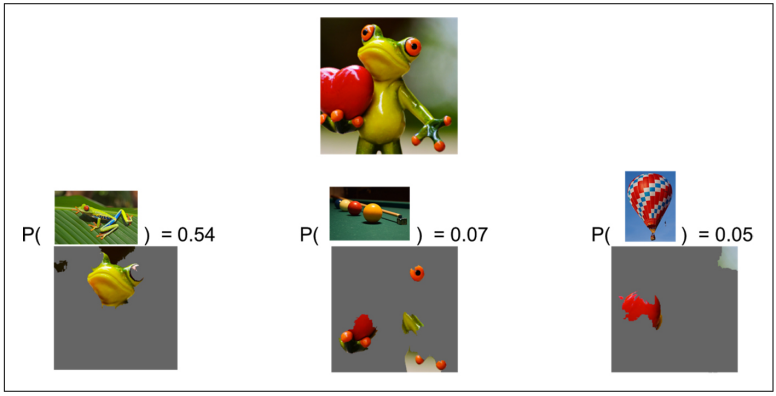

Figure 5-28. Explanation for a prediction from Inception. The top three predicted classes are "tree frog," "pool table," and "balloon." Credit: Marco Tulio Ribeiro, Pixabay (frog, billiards, hot air balloon).

In the experiments in our research paper, we demonstrate that both machine-learning experts and lay users greatly benefit from explanations similar to Figures 5-27 and 5-28, and are able to choose which models generalize better, improve models by changing them, and get crucial insights into the models' behavior.

Conclusion

Trust is crucial for effective human interaction with machine-learning systems, and we think explaining individual predictions is an effective way of assessing trust. LIME is an efficient tool to facilitate such trust for machine-learning practitioners and a good choice to add to their tool belts (did we mention we have an open source project?), but there is still plenty of work to be done to better explain machine-learning models. We're excited to see where this research direction will lead us. More details are available in our paper.

CHAPTER 6

Deep Learning and AI

We begin this chapter with an overview of the machine intelligence landscape from Shivon Zilis and James Cham. They document the emergence of a clear machine intelligence stack, weigh in on the "great chatbot explosion of 2016," and argue that implementing machine learning requires companies to make deep organizational and process changes. Then, Aaron Schumacher introduces TensorFlow, the open source software library for machine learning developed by the Google Brain Team, and explains how to build and train TensorFlow graphs. Finally, Song Han explains how deep compression greatly reduces the computation and storage required by neural networks, and also introduces a novel training method—dense-sparse-dense (DSD)—that aims to improve prediction accuracy.

The Current State of Machine Intelligence 3.0

By Shivon Zilis and James Cham

You can read this post on oreilly.com here.

Almost a year ago, we published our now-annual landscape of machine intelligence companies, and goodness have we seen a lot of activity since then. This year's landscape (see Figure 6-1) has *a third more companies* than our first one did two years ago, and it feels even more futile to try to be comprehensive, since this just scratches the surface of all of the activity out there.

As has been the case for the last couple of years, our fund still obsesses over "problem first" machine intelligence—we've invested in 35

machine-intelligence companies solving 35 meaningful problems in areas from security to recruiting to software development. (Our fund focuses on the future of work, so there are some machine-intelligence domains where we invest more than others.)

At the same time, the hype around machine-intelligence methods continues to grow: the words "deep learning" now equally represent a series of meaningful breakthroughs (wonderful) but also a hyped phrase like "big data" (not so good!). We care about whether a founder uses the right method to solve a problem, not the fanciest one. We favor those who apply technology thoughtfully.

What's the biggest change in the last year? We are getting inbound inquiries from a different mix of people. For v1.0, we heard almost exclusively from founders and academics. Then came a healthy mix of investors, both private and public. Now overwhelmingly we have heard from existing companies trying to figure out how to transform their businesses using machine intelligence.

For the first time, *a "one stop shop" of the machine intelligence stack is coming into view*—even if it's a year or two off from being neatly formalized. The maturing of that stack might explain why more established companies are more focused on building legitimate machine-intelligence capabilities. Anyone who has their wits about them is still going to be making initial build-and-buy decisions, so we figured an early attempt at laying out these technologies is better than no attempt.

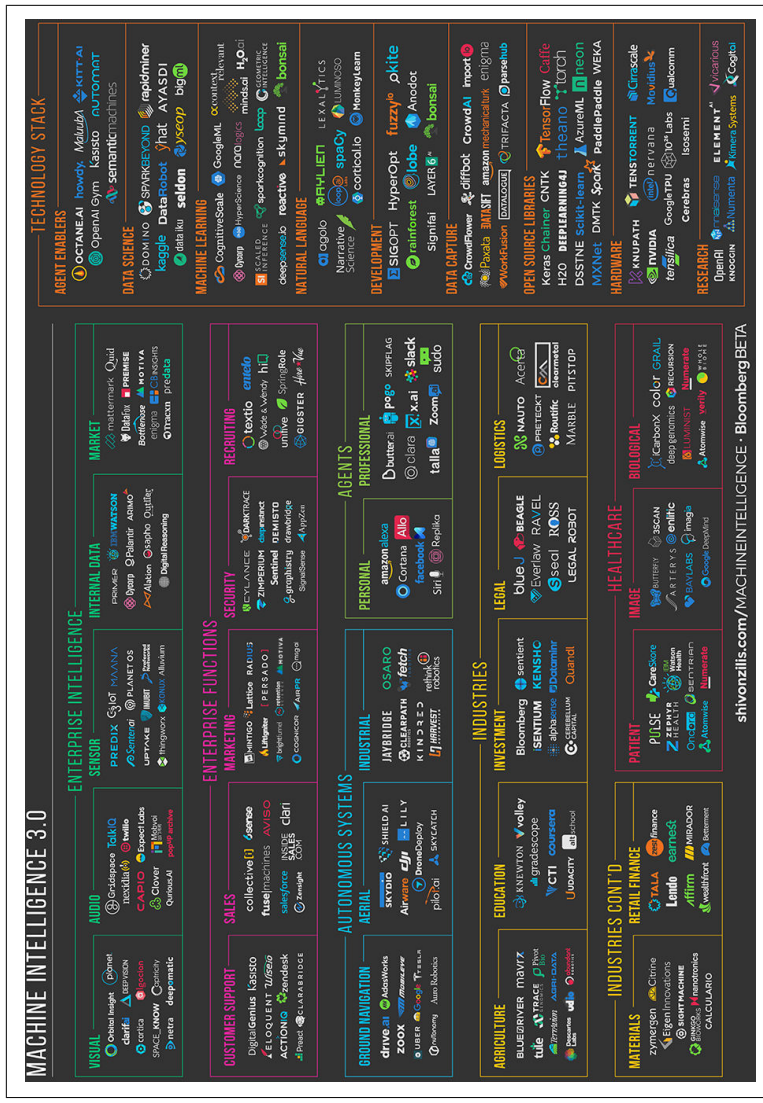

Figure 6-1. Machine intelligence landscape. Credit: Shivon Zilis and James Cham, designed by Heidi Skinner. (A larger version can be found on Shivon Zilis' website.)

Ready Player World

Many of the most impressive-looking feats we've seen have been in the gaming world, from DeepMind beating Atari classics and the

world's best at Go, to the OpenAI gym, which allows anyone to train intelligent agents across an array of gaming environments.

The gaming world offers a perfect place to start machine intelligence work (e.g., constrained environments, explicit rewards, easy-to-compare results, looks impressive)—especially for reinforcement learning. And it is much easier to have a self-driving car agent go a trillion miles in a simulated environment than on actual roads. Now we're seeing the techniques used to conquer the gaming world moving to the real world. A newsworthy example of game-tested technology entering the real world was when DeepMind used neural networks to make Google's data centers more efficient. This begs questions: What else in the world looks like a game? Or *what else in the world can we reconfigure to make it look more like a game*?

Early attempts are intriguing. Developers are dodging meter maids (brilliant—a modern day Paper Boy), categorizing cucumbers, sorting trash, and recreating the memories of loved ones as conversational bots. Otto's self-driving trucks delivering beer on their first commercial ride even seems like a bonus level from Grand Theft Auto. We're excited to see what new creative applications come in the next year.

Why Even Bot-Her?

Ah, the great chatbot explosion of 2016, for better or worse—we liken it to the mobile app explosion we saw with the launch of iOS and Android. The dominant platforms (in the machine-intelligence case, Facebook, Slack, Kik) race to get developers to build on their platforms. That means we'll get some excellent bots but also many terrible ones—the joys of public experimentation.

The danger here, unlike the mobile app explosion (where we lacked expectations for what these widgets could actually do), is that *we assume anything with a conversation interface will converse with us at near-human level. Most do not.* This is going to lead to disillusionment over the course of the next year, but it will clean itself up fairly quickly thereafter.

When our fund looks at this emerging field, we divide each technology into two components: the conversational interface itself and the "agent" behind the scenes that's learning from data and transacting on a user's behalf. While you certainly can't drop the ball on the interface, we spend almost all our time thinking about that behind-

the-scenes agent and whether it is actually solving a meaningful problem.

We get a lot of questions about whether there will be "one bot to rule them all." To be honest, as with many areas at our fund, we disagree on this. We certainly believe there will not be one *agent* to rule them all, even if there is one interface to rule them all. For the time being, bots will be idiot savants: stellar for very specific applications.

We've written a bit about this, and the framework we use to think about how agents will evolve is a CEO and her support staff. Many Fortune 500 CEOs employ a scheduler, handler, a research team, a copy editor, a speechwriter, a personal shopper, a driver, and a professional coach. Each of these people performs a dramatically different function and has access to very different data to do their job. The bot/agent ecosystem will have a similar separation of responsibilities with very clear winners, and they will divide fairly cleanly along these lines. (Note that some CEOs have a chief of staff who coordinates among all these functions, so perhaps we will see examples of "one interface to rule them all.")

You can also see, in our landscape, some of the corporate functions machine intelligence will reinvent (most often in interfaces other than conversational bots).

On to 11111000001

Successful use of machine intelligence at a large organization is surprisingly binary, like flipping a stubborn light switch. It's hard to do, but once machine intelligence is enabled, an organization sees everything through the lens of its potential. Organizations like Google, Facebook, Apple, Microsoft, Amazon, Uber, and Bloomberg (our sole investor) bet heavily on machine intelligence and its capabilities are pervasive throughout all of their products.

Other companies are struggling to figure out what to do, as many boardrooms did on "what to do about the Internet" in 1997. Why is this so difficult for companies to wrap their heads around? *Machine intelligence is different from traditional software.* Unlike with big data, where you could buy a new capability, machine intelligence depends on deeper organizational and process changes. Companies need to decide whether they will trust machine-intelligence analysis for one-off decisions or if they will embed often-inscrutable machine-intelligence models in core processes. Teams need to figure

out how to test newfound capabilities, and applications need to change so they offer more than a system of record; they also need to coach employees and learn from the data they enter.

Unlike traditional hardcoded software, machine intelligence gives only probabilistic outputs. We want to ask machine intelligence to make subjective decisions based on imperfect information (eerily like what we trust our colleagues to do?). As a result, this new machine-intelligence software will make mistakes, just like we do, and we'll need to be thoughtful about when to trust it and when not to.

The idea of this new machine trust is daunting and makes machine intelligence harder to adopt than traditional software. We've had a few people tell us that the biggest predictor of whether a company will successfully adopt machine intelligence is whether it has a C-suite executive with an advanced math degree. These executives understand it isn't magic—it is just (hard) math.

Machine-intelligence business models are going to be different from licensed and subscription software, but we don't know how. Unlike traditional software, we still lack frameworks for management to decide where to deploy machine intelligence. Economists like Ajay Agrawal, Joshua Gans, and Avi Goldfarb have taken the first steps toward helping managers understand the economics of machine intelligence and predict where it will be most effective. But there is still a lot of work to be done.

In the next few years, the danger here isn't what we see in dystopian sci-fi movies. The real danger of machine intelligence is that *executives will make bad decisions about what machine-intelligence capabilities to build.*

Peter Pan's Never-Never Land

We've been wondering about the path to grow into a large machine-intelligence company. Unsurprisingly, there have been many machine-intelligence acquisitions (Nervana by Intel, Magic Pony by Twitter, Turi by Apple, Metamind by Salesforce, Otto by Uber, Cruise by GM, SalesPredict by Ebay, Viv by Samsung). Many of these happened fairly early in a company's life and at quite a high price. Why is that?

Established companies struggle to understand machine-intelligence technology, so it's painful to sell to them, and the market for buyers who can use this technology in a self-service way is small. Then, if you do understand how this technology can supercharge your organization, you realize it's so valuable that you want to hoard it. Businesses are saying to machine-intelligence companies, "Forget you selling this technology to others; I'm going to buy the whole thing."

This absence of a market today makes it difficult for a machine-intelligence startup, especially horizontal technology providers, to "grow up"—hence the Peter Pans. *Companies we see successfully entering a long-term trajectory can package their technology as a new problem-specific application for enterprise or simply transform an industry themselves as a new entrant* (love this). In this year's landscape, we flagged a few of the industry categories where we believe startups might "go the distance."

Inspirational Machine Intelligence

Once we do figure it out, machine-intelligence can solve much more interesting problems than traditional software. We're thrilled to see so many smart people applying machine intelligence for good.

Established players like Conservation Metrics and Vulcan Conservation have been using deep learning to protect endangered animal species; the ever-inspiring team at Thorn is constantly coming up with creative algorithmic techniques to protect our children from online exploitation. The philanthropic arms of the tech titans joined in, enabling nonprofits with free storage, compute, and even developer time. Google partnered with nonprofits to found Global Fishing Watch to detect illegal fishing activity using satellite data in near real time, satellite intelligence startup Orbital Insight (in which we are investors) partnered with Global Forest Watch to detect illegal logging and other causes of global forest degradation. Startups are getting into the action, too. The Creative Destruction Lab machine intelligence accelerator (with whom we work closely) has companies working on problems like earlier disease detection and injury prevention. One area where we have seen some activity but would love to see more is machine intelligence to assist the elderly.

In talking to many people using machine intelligence for good, they all cite the critical role of open source technologies. In the last year, we've seen the launch of OpenAI, which offers everyone access to

world-class research and environments, and better and better releases of TensorFlow and Keras. Nonprofits are always trying to do more with less, and machine intelligence has allowed them to extend the scope of their missions without extending budget. Algorithms allow nonprofits to inexpensively scale what would not be affordable to do with people.

We also saw growth in universities and corporate think tanks, where new centers like USC's Center for AI in Society, Berkeley's Center for Human Compatible AI, and the multiple-corporation Partnership on AI study the ways in which machine intelligence can help humanity. The White House even got into the act: after a series of workshops around the US, it published a 48-page report outlining its recommendations for applying machine intelligence to safely and fairly address broad social problems.

On a lighter note, we've also heard whispers of more artisanal versions of machine intelligence. Folks are doing things like using computer vision algorithms to help them choose the best cocoa beans for high-grade chocolate, write poetry, cook steaks, and generate musicals.

Curious minds want to know. If you're working on a unique or important application of machine intelligence, we'd love to hear from you.

Looking Forward

We see all this activity only continuing to accelerate. The world will give us more open sourced and commercially available machine-intelligence building blocks, there will be more data, there will be more people interested in learning these methods, and there will always be problems worth solving. We still need ways of explaining the difference between machine intelligence and traditional software, and we're working on that. The value of code is different from data, but what about the value of the model that code improves based on that data?

Once we understand machine intelligence deeply, we might look back on the era of traditional software and think it was just a pro-

logue to what's happening now. We look forward to seeing what the next year brings.

Thank Yous

A massive thank you to the Bloomberg Beta team, David Klein, Adam Gibson, Ajay Agrawal, Alexandra Suich, Angela Tranyens, Anthony Goldblum, Avi Goldfarb, Beau Cronin, Ben Lorica, Chris Nicholson, Doug Fulop, Dror Berman, Dylan Tweney, Gary Kazantsev, Gideon Mann, Gordon Ritter, Jack Clark, John Lilly, Jon Lehr, Joshua Gans, Matt Turck, Matthew Granade, Mickey Graham, Nick Adams, Roger Magoulas, Sean Gourley, Shruti Gandhi, Steve Jurvetson, Vijay Sundaram, Zavain Dar, and for the help and fascinating conversations that led to this year's report!

Landscape designed by Heidi Skinner.

Disclosure: Bloomberg Beta is an investor in Alation, Arimo, Aviso, Brightfunnel, Context Relevant, Deep Genomics, Diffbot, Digital Genius, Domino Data Labs, Drawbridge, Gigster, Gradescope, Graphistry, Gridspace, Howdy, Kaggle, Kindred.ai, Mavrx, Motiva, PopUpArchive, Primer, Sapho, Shield.AI, Textio, and Tule.

Hello, TensorFlow!

By Aaron Schumacher

You can read this post on oreilly.com here.

The TensorFlow project is bigger than you might realize. The fact that it's a library for deep learning and its connection to Google, have helped TensorFlow attract a lot of attention. But beyond the hype, there are unique elements to the project that are worthy of closer inspection:

- The core library is suited to a broad family of machine-learning techniques, not "just" deep learning.
- Linear algebra and other internals are prominently exposed.
- In addition to the core machine-learning functionality, TensorFlow also includes its own logging system, its own interactive log visualizer, and even its own heavily engineered serving architecture.

- The execution model for TensorFlow differs from Python's scikit-learn, and most tools in R.

Cool stuff, but—especially for someone hoping to explore machine learning for the first time—TensorFlow can be a lot to take in.

How does TensorFlow work? Let's break it down so we can see and understand every moving part. We'll explore the data flow graph that defines the computations your data will undergo, how to train models with gradient descent using TensorFlow, and how TensorBoard can visualize your TensorFlow work. The examples here won't solve industrial machine-learning problems, but they'll help you understand the components underlying everything built with TensorFlow, including whatever you build next!

Names and Execution in Python and TensorFlow

The way TensorFlow manages computation is not totally different from the way Python usually does. With both, it's important to remember, to paraphrase Hadley Wickham, that an object has no name (see Figure 6-2). In order to see the similarities (and differences) between how Python and TensorFlow work, let's look at how they refer to objects and handle evaluation.

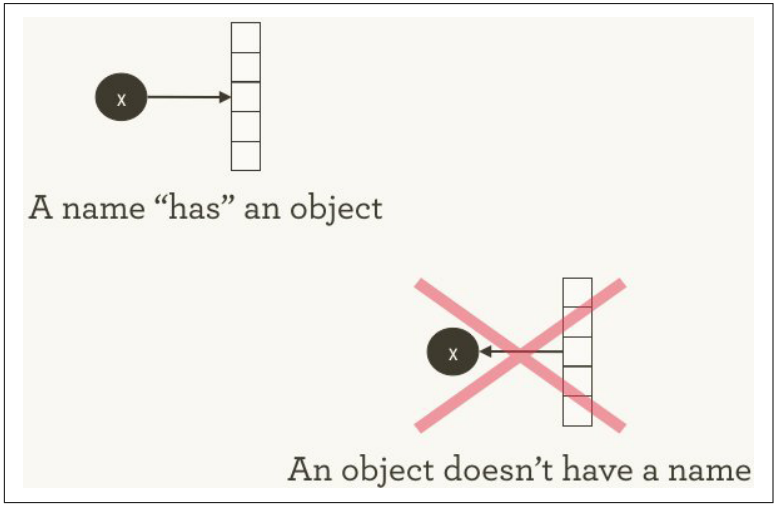

Figure 6-2. Names "have" objects, rather than the reverse. Credit: Hadley Wickham. Used with permission.

The variable names in Python code aren't what they represent; they're just pointing at objects. So, when you say in Python that `foo = []` and `bar = foo`, it isn't just that `foo` equals `bar`; `foo` *is* `bar`, in the sense that they both point at the same list object:

```
>>> foo = []
>>> bar = foo
>>> foo == bar
## True
>>> foo is bar
## True
```

You can also see that `id(foo)` and `id(bar)` are the same. This identity, especially with mutable data structures like lists, can lead to surprising bugs when it's misunderstood.

Internally, Python manages all your objects and keeps track of your variable names and which objects they refer to. The TensorFlow graph represents another layer of this kind of management; as we'll see, Python names will refer to objects that connect to more granular and managed TensorFlow graph operations.

When you enter a Python expression, for example at an interactive interpreter or Read-Evaluate-Print-Loop (REPL), whatever is read is almost always evaluated right away. Python is eager to do what you tell it. So, if I tell Python to `foo.append(bar)`, it appends right away, even if I never use `foo` again.

A lazier alternative would be to just remember that I said `foo.append(bar)`, and if I ever evaluate `foo` at some point in the future, Python could do the append then. This would be closer to how TensorFlow behaves, where defining relationships is entirely separate from evaluating what the results are.

TensorFlow separates the definition of computations from their execution even further by having them happen in separate places: a graph defines the operations, but the operations only happen within a session. Graphs and sessions are created independently. A graph is like a blueprint, and a session is like a construction site.

Back to our plain Python example, recall that `foo` and `bar` refer to the same list. By appending `bar` into `foo`, we've put a list inside itself. You could think of this structure as a graph with one node, pointing to itself. Nesting lists is one way to represent a graph structure like a TensorFlow computation graph:

```
>>> foo.append(bar)
>>> foo
## [[...]]
```

Real TensorFlow graphs will be more interesting than this!

The Simplest TensorFlow Graph

To start getting our hands dirty, let's create the simplest TensorFlow graph we can, from the ground up. TensorFlow is admirably easier to install than some other frameworks. The examples here work with either Python 2.7 or 3.3+, and the TensorFlow version used is 0.8:

```
>>> import tensorflow as tf
```

At this point, TensorFlow has already started managing a lot of state for us. There's already an implicit default graph, for example. Internally, the default graph lives in the `_default_graph_stack`, but we don't have access to that directly. We use `tf.get_default_graph()`:

```
>>> graph = tf.get_default_graph()
```

The nodes of the TensorFlow graph are called "operations," or "ops." We can see what operations are in the graph with `graph.get_operations()`:

```
>>> graph.get_operations()
## []
```

Currently, there isn't anything in the graph. We'll need to put everything we want TensorFlow to compute into that graph. Let's start with a simple constant input value of 1:

```
>>> input_value = tf.constant(1.0)
```

That constant now lives as a node, an operation, in the graph. The Python variable name `input_value` refers indirectly to that operation, but we can also find the operation in the default graph:

```
>>> operations = graph.get_operations()
>>> operations
## [<tensorflow.python.framework.ops.Operation at 0x1185005d0>]
>>> operations[0].node_def
## name: "Const"
## op: "Const"
## attr {
##   key: "dtype"
##   value {
##     type: DT_FLOAT
```

```
##   }
## }
## attr {
##   key: "value"
##   value {
##     tensor {
##       dtype: DT_FLOAT
##       tensor_shape {
##       }
##       float_val: 1.0
##     }
##   }
## }
```

TensorFlow uses protocol buffers internally. (Protocol buffers are sort of like a Google-strength JSON.) Printing the `node_def` for the constant operation in the preceding code block shows what's in TensorFlow's protocol buffer representation for the number 1.

People new to TensorFlow sometimes wonder why there's all this fuss about making "TensorFlow versions" of things. Why can't we just use a normal Python variable without also defining a TensorFlow object? One of the TensorFlow tutorials has an explanation:

> To do efficient numerical computing in Python, we typically use libraries like NumPy that do expensive operations such as matrix multiplication outside Python, using highly efficient code implemented in another language. Unfortunately, there can still be a lot of overhead from switching back to Python every operation. This overhead is especially bad if you want to run computations on GPUs or in a distributed manner, where there can be a high cost to transferring data.
>
> TensorFlow also does its heavy lifting outside Python, but it takes things a step further to avoid this overhead. Instead of running a single expensive operation independently from Python, TensorFlow lets us describe a graph of interacting operations that run entirely outside Python. This approach is similar to that used in Theano or Torch.

TensorFlow can do a lot of great things, but it can only work with what's been explicitly given to it. This is true even for a single constant.

If we inspect our `input_value`, we see it is a constant 32-bit float tensor of no dimension: just one number:

```
>>> input_value
## <tf.Tensor 'Const:0' shape=() dtype=float32>
```

Note that this *doesn't* tell us what that number *is*. To evaluate `input_value` and get a numerical value out, we need to create a "session" where graph operations can be evaluated and then explicitly ask to evaluate or "run" `input_value`. (The session picks up the default graph by default.)

```
>>> sess = tf.Session()
>>> sess.run(input_value)
## 1.0
```

It may feel a little strange to "run" a constant. But it isn't so different from evaluating an expression as usual in Python; it's just that TensorFlow is managing its own space of things—the computational graph—and it has its own method of evaluation.

The Simplest TensorFlow Neuron

Now that we have a session with a simple graph, let's build a neuron with just one parameter, or weight. Often, even simple neurons also have a bias term and a nonidentity activation function, but we'll leave these out.

The neuron's weight isn't going to be constant; we expect it to change in order to learn based on the "true" input and output we use for training. The weight will be a TensorFlow variable. We'll give that variable a starting value of 0.8:

```
>>> weight = tf.Variable(0.8)
```

You might expect that adding a variable would add one operation to the graph, but in fact that one line adds four operations. We can check all the operation names:

```
>>> for op in graph.get_operations(): print(op.name)
## Const
## Variable/initial_value
## Variable
## Variable/Assign
## Variable/read
```

We won't want to follow every operation individually for long, but it will be nice to see at least one that feels like a real computation:

```
>>> output_value = weight * input_value
```

Now there are six operations in the graph, and the last one is that multiplication:

```
>>> op = graph.get_operations()[-1]
>>> op.name
## 'mul'
>>> for op_input in op.inputs: print(op_input)
## Tensor("Variable/read:0", shape=(), dtype=float32)
## Tensor("Const:0", shape=(), dtype=float32)
```

This shows how the multiplication operation tracks where its inputs come from: they come from other operations in the graph. To understand a whole graph, following references this way quickly becomes tedious for humans. TensorBoard graph visualization is designed to help.

How do we find out what the product is? We have to "run" the `output_value` operation. But that operation depends on a variable: `weight`. We told TensorFlow that the initial value of `weight` should be 0.8, but the value hasn't yet been set in the current session. The `tf.initialize_all_variables()` function generates an operation which will initialize all our variables (in this case just one), and then we can run that operation:

```
>>> init = tf.initialize_all_variables()
>>> sess.run(init)
```

The result of `tf.initialize_all_variables()` will include initializers for all the variables *currently in the graph,* so if you add more variables you'll want to use `tf.initialize_all_variables()` again; a stale `init` wouldn't include the new variables.

Now we're ready to run the `output_value` operation:

```
>>> sess.run(output_value)
## 0.80000001
```

Recall that it is 0.8 * 1.0 with 32-bit floats, and 32-bit floats have a hard time with 0.8; 0.80000001 is as close as they can get.

See Your Graph in TensorBoard

Up to this point, the graph has been simple, but it would already be nice to see it represented in a diagram. We'll use TensorBoard to generate that diagram. TensorBoard reads the name field that is stored inside each operation (quite distinct from Python variable names). We can use these TensorFlow names and switch to more conventional Python variable names. Using `tf.mul` here is equivalent to our earlier use of just `*` for multiplication, but it lets us set the name for the operation:

```
>>> x = tf.constant(1.0, name='input')
>>> w = tf.Variable(0.8, name='weight')
>>> y = tf.mul(w, x, name='output')
```

TensorBoard works by looking at a directory of output created from TensorFlow sessions. We can write this output with a `SummaryWriter`, and if we do nothing aside from creating one with a graph, it will just write out that graph.

The first argument when creating the `SummaryWriter` is an output directory name, which will be created if it doesn't exist:

```
>>> summary_writer = tf.train.SummaryWriter('log_simple_graph',
sess.graph)
```

Now, at the command line, we can start up TensorBoard:

```
$ tensorboard --logdir=log_simple_graph
```

TensorBoard runs as a local web app, on port 6006. ("6006" is "goog" upside-down.) If you go in a browser to `localhost:6006/#graphs`, you should see a diagram of the graph you created in TensorFlow, which looks something like Figure 6-3.

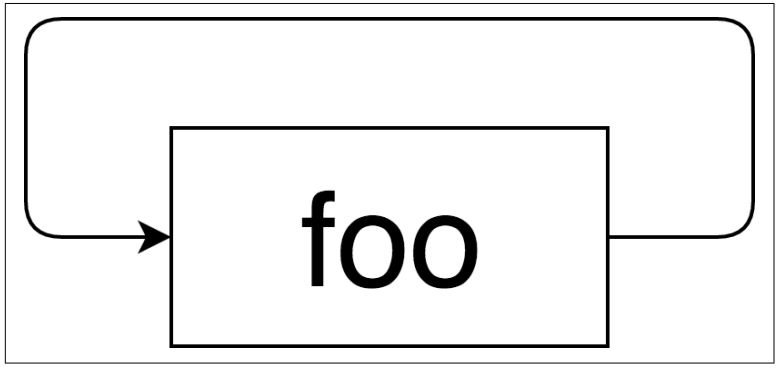

Figure 6-3. A TensorBoard visualization of the simplest TensorFlow neuron.

Making the Neuron Learn

Now that we've built our neuron, how does it learn? We set up an input value of 1.0. Let's say the correct output value is zero. That is, we have a very simple "training set" of just one example with one feature, which has the value 1, and one label, which is zero. We want the neuron to learn the function taking 1 to 0.

Currently, the system takes the input 1 and returns 0.8, which is not correct. We need a way to measure how wrong the system is. We'll call that measure of wrongness the "loss" and give our system the goal of minimizing the loss. If the loss can be negative, then minimizing it could be silly, so let's make the loss the square of the difference between the current output and the desired output:

```
>>> y_ = tf.constant(0.0)
>>> loss = (y - y_)**2
```

So far, nothing in the graph does any learning. For that, we need an optimizer. We'll use a gradient descent optimizer so that we can update the weight based on the derivative of the loss. The optimizer takes a learning rate to moderate the size of the updates, which we'll set at 0.025:

```
>>> optim = tf.train.GradientDescentOptimizer
(learning_rate=0.025)
```

The optimizer is remarkably clever. It can automatically work out and apply the appropriate gradients through a whole network, carrying out the backward step for learning.

Let's see what the gradient looks like for our simple example:

```
>>> grads_and_vars = optim.compute_gradients(loss)
>>> sess.run(tf.initialize_all_variables())
>>> sess.run(grads_and_vars[1][0])
## 1.6
```

Why is the value of the gradient 1.6? Our loss is error squared, and the derivative of that is two times the error. Currently the system says 0.8 instead of 0, so the error is 0.8, and two times 0.8 is 1.6. It's working!

For more complex systems, it will be very nice indeed that TensorFlow calculates and then applies these gradients for us automatically.

Let's apply the gradient, finishing the backpropagation:

```
>>> sess.run(optim.apply_gradients(grads_and_vars))
>>> sess.run(w)
## 0.75999999  # about 0.76
```

The weight decreased by 0.04 because the optimizer subtracted the gradient times the learning rate, 1.6 * 0.025, pushing the weight in the right direction.

Instead of hand-holding the optimizer like this, we can make one operation that calculates and applies the gradients: the `train_step`:

```
>>> train_step = tf.train.GradientDescentOptimizer(0.025)
.minimize(loss)
>>> for i in range(100):
>>>     sess.run(train_step)
>>>
>>> sess.run(y)
## 0.0044996012
```

Running the training step many times, the weight and the output value are now very close to zero. The neuron has learned!

Training diagnostics in TensorBoard

We may be interested in what's happening during training. Say we want to follow what our system is predicting at every training step. We could print from inside the training loop:

```
>>> sess.run(tf.initialize_all_variables())
>>> for i in range(100):
>>>     print('before step {}, y is {}'.format(i, sess.run(y)))
>>>     sess.run(train_step)
>>>
## before step 0, y is 0.800000011921
## before step 1, y is 0.759999990463
## ...
## before step 98, y is 0.00524811353534
## before step 99, y is 0.00498570781201
```

This works, but there are some problems. It's hard to understand a list of numbers. A plot would be better. And even with only one value to monitor, there's too much output to read. We're likely to want to monitor many things. It would be nice to record everything in some organized way.

Luckily, the same system that we used earlier to visualize the graph also has just the mechanisms we need.

We instrument the computation graph by adding operations that summarize its state. Here, we'll create an operation that reports the current value of y, the neuron's current output:

```
>>> summary_y = tf.scalar_summary('output', y)
```

When you run a summary operation, it returns a string of protocol buffer text that can be written to a log directory with a `SummaryWriter`:

```
>>> summary_writer = tf.train.SummaryWriter('log_simple_stats')
>>> sess.run(tf.initialize_all_variables())
>>> for i in range(100):
```

```
>>>     summary_str = sess.run(summary_y)
>>>     summary_writer.add_summary(summary_str, i)
>>>     sess.run(train_step)
>>>
```

Now after running `tensorboard --logdir=log_simple_stats`, you get an interactive plot at `localhost:6006/#events` (Figure 6-4).

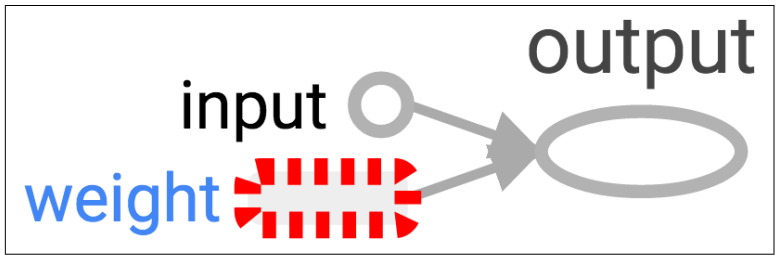

Figure 6-4. A TensorBoard visualization of a neuron's output against training iteration number.

Flowing Onward

Here's a final version of the code. It's fairly minimal, with every part showing useful (and understandable) TensorFlow functionality:

```
import tensorflow as tf

x = tf.constant(1.0, name='input')
w = tf.Variable(0.8, name='weight')
y = tf.mul(w, x, name='output')
y_ = tf.constant(0.0, name='correct_value')
loss = tf.pow(y - y_, 2, name='loss')
train_step = tf.train.GradientDescentOptimizer(0.025)
.minimize(loss)

for value in [x, w, y, y_, loss]:
    tf.scalar_summary(value.op.name, value)

summaries = tf.merge_all_summaries()

sess = tf.Session()
summary_writer = tf.train.SummaryWriter('log_simple_stats',
sess.graph)

sess.run(tf.initialize_all_variables())
for i in range(100):
    summary_writer.add_summary(sess.run(summaries), i)
    sess.run(train_step)
```

The example we just ran through is even simpler than the ones that inspired it in Michael Nielsen's *Neural Networks and Deep Learning*. For myself, seeing details like these helps with understanding and building more complex systems that use and extend from simple building blocks. Part of the beauty of TensorFlow is how flexibly you can build complex systems from simpler components.

If you want to continue experimenting with TensorFlow, it might be fun to start making more interesting neurons, perhaps with different activation functions. You could train with more interesting data. You could add more neurons. You could add more layers. You could dive into more complex prebuilt models, or spend more time with TensorFlow's own tutorials and how-to guides. Go for it!

Compressing and Regularizing Deep Neural Networks

By Song Han

You can read this post on oreilly.com here.

Deep neural networks have evolved to be the state-of-the-art technique for machine-learning tasks ranging from computer vision and speech recognition to natural language processing. However, deep-learning algorithms are both computationally intensive and memory intensive, making them difficult to deploy on embedded systems with limited hardware resources.

To address this limitation, *deep compression* significantly reduces the computation and storage required by neural networks. For example, for a convolutional neural network with fully connected layers, such as Alexnet and VGGnet, it can reduce the model size by 35x–49x. Even for fully convolutional neural networks such as GoogleNet and SqueezeNet, deep compression can still reduce the model size by 10x. *Both scenarios results in no loss of prediction accuracy.*

Current Training Methods Are Inadequate

Compression without losing accuracy means there's significant redundancy in the trained model, which shows the inadequacy of current training methods. To address this, I've worked with Jeff Pool, of NVIDIA, Sharan Narang of Baidu, and Peter Vajda of Facebook to develop the dense-sparse-dense (DSD) training, a novel

training method that first regularizes the model through sparsity-constrained optimization, and improves the prediction accuracy by recovering and retraining on pruned weights. At test time, the final model produced by DSD training still has the same architecture and dimension as the original dense model, and DSD training doesn't incur any inference overhead. We experimented with DSD training on mainstream CNN/RNN/LSTMs for image classification, image caption, and speech recognition and found substantial performance improvements.

In this article, we first introduce deep compression, and then introduce dense-sparse-dense training.

Deep Compression

The first step of deep compression is *synaptic pruning*. The human brain inherently has the process of pruning. 5x synapses are pruned away from infant age to adulthood.

Does a similar process occur in artificial neural networks? The answer is yes. In early work, network pruning proved to be a valid way to reduce the network complexity and overfitting. This method works on modern neural networks as well. We start by learning the connectivity via normal network training. Next, we prune the small-weight connections: all connections with weights below a threshold are removed from the network. Finally, we retrain the network to learn the final weights for the remaining sparse connections. Pruning reduced the number of parameters by 9x and 13x for AlexNet and the VGG-16 model, respectively.

The next step of deep compression is *weight sharing*. We found neural networks have a really high tolerance for low precision: aggressive approximation of the weight values does not hurt the prediction accuracy. As shown in Figure 6-6, the blue weights are originally 2.09, 2.12, 1.92 and 1.87; by letting four of them share the same value, which is 2.00, the accuracy of the network can still be recovered. Thus we can save very few weights, call it "codebook," and let many other weights share the same weight, storing only the index to the codebook.

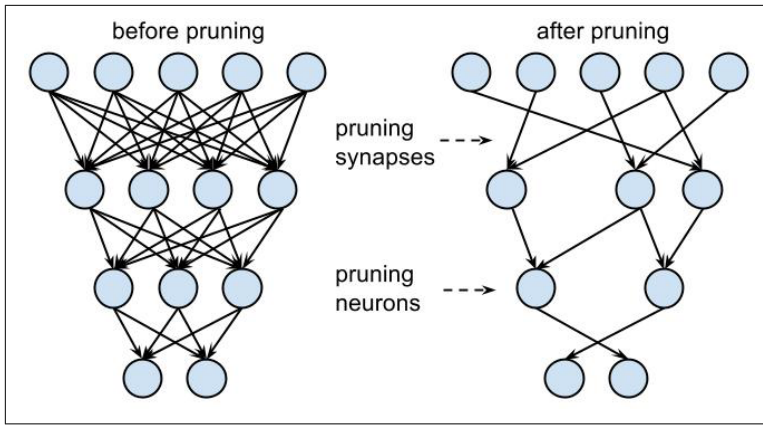

Figure 6-5. Pruning a neural network. Credit: Song Han.

The index could be represented with very few bits; for example, in Figure 6-6, there are four colors; thus only two bits are needed to represent a weight as opposed to 32 bits originally. The codebook, on the other side, occupies negligible storage. Our experiments found this kind of weight-sharing technique is better than linear quantization, with respect to the compression ratio and accuracy trade-off.

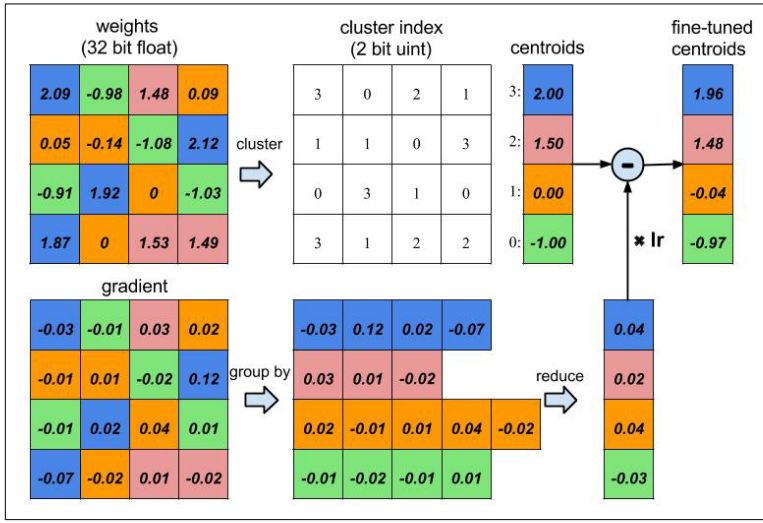

Figure 6-6. Training a weight-sharing neural network.

Figure 6-7 shows the overall result of deep compression. Lenet-300-100 and Lenet-5 are evaluated on a MNIST data set, while AlexNet, VGGNet, GoogleNet, and SqueezeNet are evaluated on an ImageNet data set. The compression ratio ranges from 10x to 49x—even for those fully convolutional neural networks like GoogleNet and SqueezeNet, deep compression can still compress it by an order of magnitude. We highlight SqueezeNet, which has 50x fewer parameters than AlexNet but has the same accuracy, and can still be compressed by 10x, making it only 470 KB. This makes it easy to fit in on-chip SRAM, which is both faster and more energy-efficient to access than DRAM.

We have tried other compression methods such as low-rank approximation-based methods, but the compression ratio isn't as high. A complete discussion can be found in the "Deep Compression" paper.

Network	Original Size	Compressed Size	Compression Ratio	Original Accuracy	Compressed Accuracy
Lenet-300-100	1070KB	27KB	**40x**	98.36%	98.42%
Lenet-5	1720KB	44Kb	**39x**	99.20%	99.26%
AlexNet	240MB	6.9MB	**35x**	80.27%	80.30%
VGGNet	550MB	11.3MB	**49x**	88.68%	89.09%
GoogleNet	28MB	2.8MB	**10x**	88.90%	88.92%
SqueezeNet	4.8MB	0.47MB	**10x**	80.32%	80.35%

Figure 6-7. Results of deep compression.

DSD Training

The fact that deep neural networks can be aggressively pruned and compressed means that our current training method has some limitation: it can not fully exploit the full capacity of the dense model to find the best local minima; yet a pruned, sparse model that has much fewer synapses can achieve the same accuracy. This raises a question: can we achieve better accuracy by recovering those weights, and learn them again?

Let's make an analogy to training for track racing in the Olympics. The coach will first train a runner on high-altitude mountains, where there are a lot of constraints: low oxygen, cold weather, etc. The result is that when the runner returns to the plateau area again, his/her speed is increased. Similar for neural networks, given the heavily constrained sparse training, the network performs as well as

the dense model; once you release the constraint, the model can work better.

Theoretically, the following factors contribute to the effectiveness of DSD training:

1. **Escape saddle point:** one of the most profound difficulties of optimizing deep networks is the proliferation of saddle points. DSD training overcomes saddle points by a pruning and re-densing framework. Pruning the converged model perturbs the learning dynamics and allows the network to jump away from saddle points, which gives the network a chance to converge at a better local or global minimum. This idea is also similar to simulated annealing. While simulated annealing randomly jumps with decreasing probability on the search graph, DSD deterministically deviates from the converged solution achieved in the first dense training phase by removing the small weights and enforcing a sparsity support.
2. **Regularized and sparse training:** the sparsity regularization in the sparse training step moves the optimization to a lower-dimensional space where the loss surface is smoother and tends to be more robust to noise. More numerical experiments verified that both sparse training and the final DSD reduce the variance and lead to lower error.
3. **Robust reinitialization:** weight initialization plays a big role in deep learning. Conventional training has only one chance of initialization. DSD gives the optimization a second (or more) chance during the training process to reinitialize from more robust sparse training solutions. We re-dense the network from the sparse solution, which can be seen as a zero initialization for pruned weights. Other initialization methods are also worth trying.
4. **Break symmetry:** The permutation symmetry of the hidden units makes the weights symmetrical, thus prone to co-adaptation in training. In DSD, pruning the weights breaks the symmetry of the hidden units associated with the weights, and the weights are asymmetrical in the final dense phase.

We examined several mainstream CNN/RNN/LSTM architectures on image classification, image caption, and speech recognition data sets, and found that this dense-sparse-dense training flow gives significant accuracy improvement. Our DSD training employs a three-

step process: dense, sparse, dense; each step is illustrated in Figure 6-8:

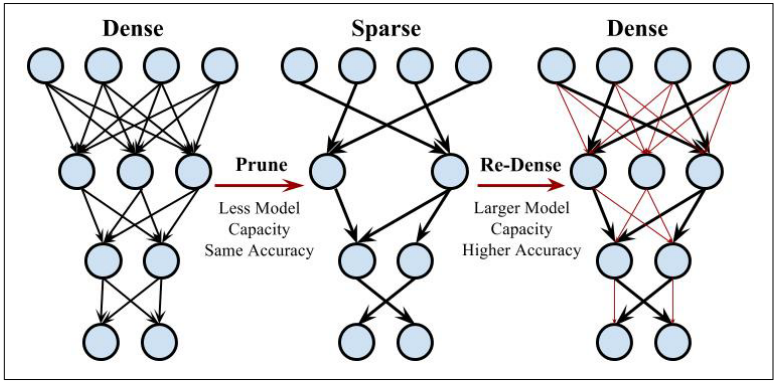

Figure 6-8. Dense-sparse-dense training flow.

1. **Initial dense training**: the first D-step learns the connectivity via normal network training on the dense network. Unlike conventional training, however, the goal of this D step is not to learn the final values of the weights; rather, we are learning which connections are important.
2. **Sparse training**: the S-step prunes the low-weight connections and retrains the sparse network. We applied the same sparsity to all the layers in our experiments; thus there's a *single* hyperparameter: the sparsity. For each layer, we sort the parameters, and the smallest N*sparsity parameters are removed from the network, converting a dense network into a sparse network. We found that a sparsity ratio of 50%–70% works very well. Then, we retrain the sparse network, which can fully recover the model accuracy under the sparsity constraint.
3. **Final dense training**: the final D step recovers the pruned connections, making the network dense again. These previously pruned connections are initialized to zero and retrained. Restoring the pruned connections increases the dimensionality of the network, and more parameters make it easier for the network to slide down the saddle point to arrive at a better local minima.

We applied DSD training to different kinds of neural networks on data sets from different domains. We found that DSD training improved the accuracy for all these networks compared to neural

networks that were not trained with DSD. The neural networks are chosen from CNN, RNN, and LSTMs; the data sets are chosen from image classification, speech recognition, and caption generation. The results are shown in Figure 6-9. DSD models are available to download at DSD Model Zoo.

Baseline	Top-1 error	Top-5 error	DSD	Top-1 error	Top-5 error
AlexNet	42.78%	19.73%	AlexNet_DSD	41.48%	18.71%
VGG16	31.50%	11.32%	VGG16_DSD	27.19%	8.67%
GoogleNet	31.14%	10.96%	GoogleNet_DSD	30.02%	10.34%
SqueezeNet	42.39%	19.32%	SqueezeNet_DSD	38.24%	16.53%
ResNet18	30.43%	10.76%	ResNet18_DSD	29.17%	10.13%
ResNet50	24.01%	7.02%	ResNet50_DSD	22.89%	6.47%

Figure 6-9. DSD training improves the prediction accuracy.

Generating Image Descriptions

We visualized the effect of DSD training on an image caption task (see Figure 6-10). We applied DSD to NeuralTalk, an LSTM for generating image descriptions. The baseline model fails to describe images 1, 4, and 5. For example, in the first image, the baseline model mistakes the girl for a boy, and mistakes the girl's hair for a rock wall; the sparse model can tell that it's a girl in the image, and the DSD model can further identify the swing.

In the the second image, DSD training can tell that the player is trying to make a shot, rather than the baseline, which just says he's playing with a ball. It's interesting to notice that the sparse model sometimes works better than the DSD model. In the last image, the sparse model correctly captured the mud puddle, while the DSD model only captured the forest from the background. The good performance of DSD training generalizes beyond these examples, and more image caption results generated by DSD training are provided in the appendix of this paper.

Figure 6-10. Visualization of DSD training improves the performance of image captioning.

Advantages of Sparsity

Deep compression for compressing deep neural networks for smaller model size and DSD training for regularizing neural networks are techniques that utilize sparsity and achieve a smaller size or higher prediction accuracy. Apart from model size and prediction accuracy, we looked at two other dimensions that take advantage of sparsity: speed and energy efficiency, which is beyond the scope of this article. Readers can refer to our paper "EIE: Efficient Inference Engine on Compressed Deep Neural Network" for further references.