O'REILLY®

Introducing MLOps

How to Scale Machine Learning in the Enterprise

Early Release
RAW & UNEDITED

Compliments of
dataiku

Clément Stenac
and the Dataiku team

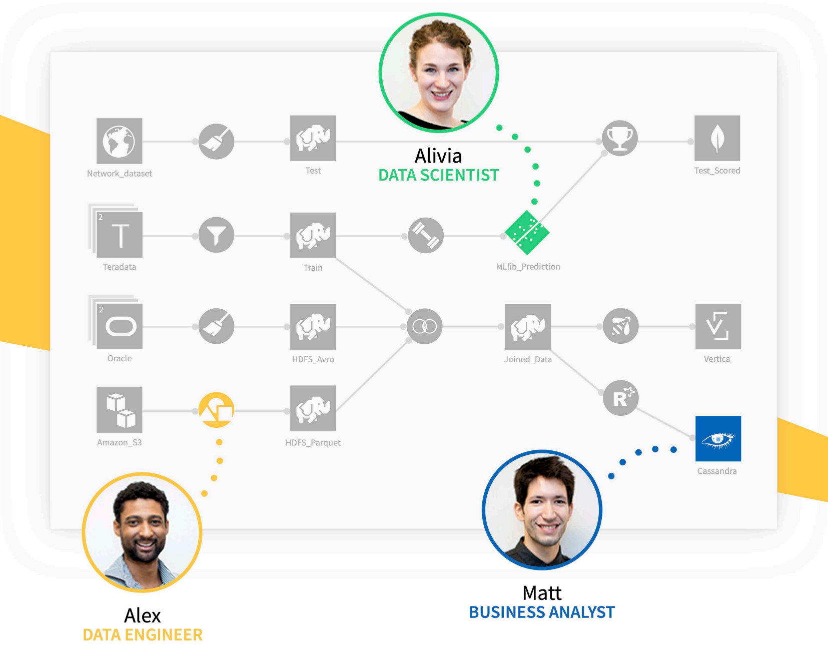
MASTERING MLOps WITH DATAIKU

Dataiku is the only platform that provides one simple, consistent UI for data connection, wrangling, mining, visualization, machine learning, deployment, and model monitoring, all at enterprise scale.

Key features for a scalable MLOps strategy include:

1. Model input drift detection that looks at the recent data the model has had to score and statistically compares it with the data on which the model was evaluated.
2. Easier creation of validation feedback loops via Dataiku Evaluation Recipes to compute the true performance of a saved model against a new validation dataset, plus automated retraining and redeployment.
3. Dashboard interfaces dedicated to the monitoring of global pipelines.
4. ...and more! Go in-depth on all the features Dataiku has to offer with the complete data sheet.

GET THE DATAIKU DATA SHEET



Introducing MLOps

How to Scale Machine Learning in the Enterprise

Clément Stenac, Léo Dreyfus-Schmidt, Kenji Lefèvre, Nicolas Omont, and Mark Treveil

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

Introducing MLOps
by Clément Stenac, Léo Dreyfus-Schmidt, Kenji Lefèvre, Nicolas Omont, and Mark Treveil

Printed in the United States of America.

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

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

Editors: Angela Rufino and Rebecca Novack
Production Editor: Katherine Tozer
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

February 2021: First Edition

Revision History for the Early Release

2020-05-19: First Release
2020-07-13: Second Release
2020-09-18: Third Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492083290* for release details.

978-1-098-11640-8

Table of Contents

CHAPTER 1

Why Now and Challenges

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 first 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 *mlops@dataiku.com*.

Machine learning operations (MLOps) is quickly becoming a critical component of successful data science project deployment in the enterprise (Figure 1-1). Yet it's a relatively new concept, so why has it seemingly skyrocketed into the data science lexicon overnight? This introductory chapter will delve into what MLOps is at a high level, its challenges, why it's become essential to a successful data science strategy in the enterprise, and — critically — why it is coming to the forefront now.

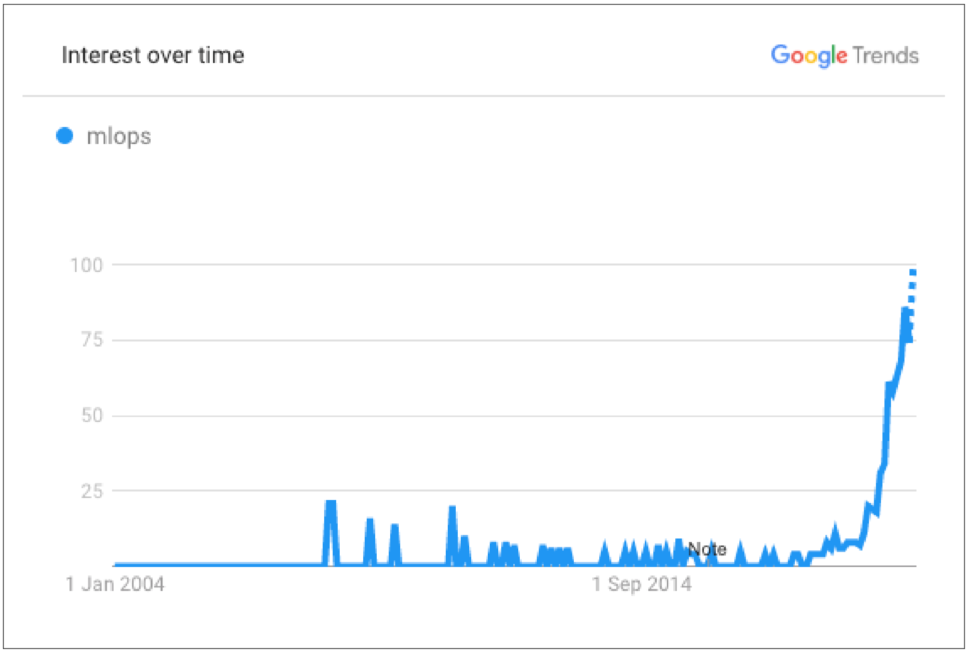


Figure 1-1. The exponential growth of MLOps. This represents only the growth of MLOps, not the parallel growth of the term ModelOps (subtle differences explained in the sidebar MLOps vs. ModelOps vs. AIOps).

MLOps vs. ModelOps vs. AIOps

MLOps (or ModelOps) is a relatively new discipline, emerging under these names particularly in late 2018 and 2019. The two — MLOps and ModelOps — are, at the time this book is being written and published, largely being used interchangeably. However, some argue that ModelOps is more general than MLOps, as it's not only about machine learning models but any kind of model (e.g., rule-based models). For the purpose of this book, we'll be specifically discussing the machine learning model lifecycle and will thus use MLOps.

AIOps, though sometimes confused with MLOps, is another topic entirely and refers to the process of solving operational challenges through the use of artificial intelligence (i.e., AI for DevOps). An example would be a form of predictive maintenance but for network failures, alerting DevOps teams to possible problems before they arise. While important and interesting in its own right, AIOps is outside the scope of this book.

Defining MLOps and Its Challenges

At its core, MLOps is the standardization and streamlining of machine learning lifecycle management (Figure 1-2). But taking a step back, why does the machine learning lifecycle need to be streamlined? Surface-level, in looking at the steps to go from business problem to a machine learning model at a very high level, it seems straightforward:

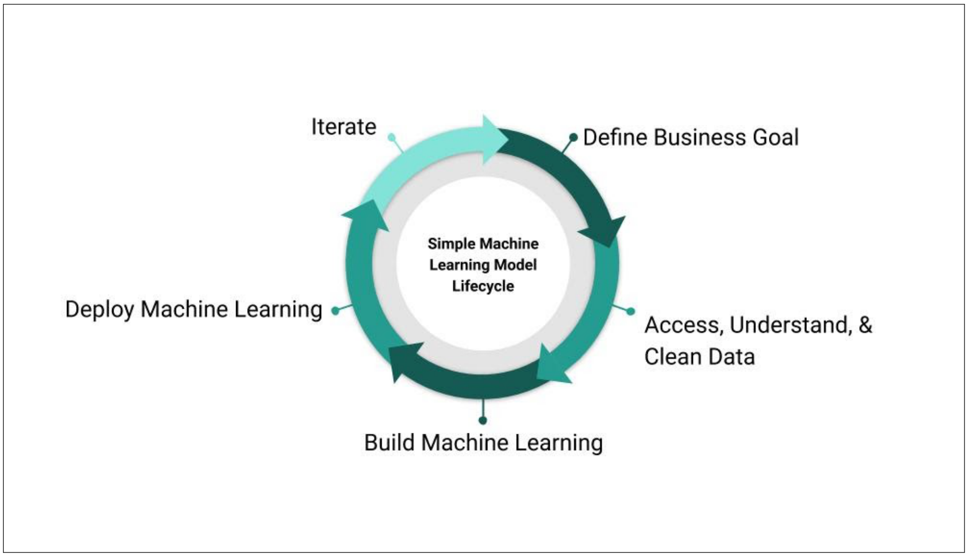


Figure 1-2. A simple representation of the machine learning model lifecycle, which often underplays the need for MLOps; compare to Figure 3, which is a more realistic representation of how the machine learning model lifecycle plays out in today's organizations, which are complex in terms of needs as well as tooling.

For most traditional organizations, the development of multiple machine learning models and their deployment in a production environment are relatively new. Until recently, the number of models may have been manageable at a small scale, or there was simply less interest in understanding these models and their dependencies at a company-wide level. With decision automation, models become more critical, and in parallel, managing model risks becomes more important at the top level.

The reality of the machine learning lifecycle in an enterprise setting is much more complex (Figure 1-3). There are three key reasons that managing machine learning lifecycles at scale are challenging:

There are many dependencies: Not only is data constantly changing, but business needs shift as well. Results need to be continually relayed back to the business to ensure that the reality of the model in production and on production data aligns with expectations and — critically — addresses the original problem or meets the original goal.

Not everyone speaks the same language: Even though the machine learning lifecycle involves people from the business, data science, and IT teams, none of these groups are using the same tools or even — in many cases — share the same fundamental skills to serve as a baseline of communication.

- Data scientists are not software engineers: Most are specialized in model building and assessment, and they are not necessarily experts in writing applications. Though this may start to shift over time as some data scientists become specialists more on the deployment or operationalization side, for now, many data scientists find themselves having to juggle many roles, making it challenging to do any of them thoroughly. Data scientists being stretched too thin becomes especially problematic at scale with increasingly more models to manage. The complexity becomes exponential when considering the turnover of staff on data teams when suddenly, data scientists have to manage models they did not create.

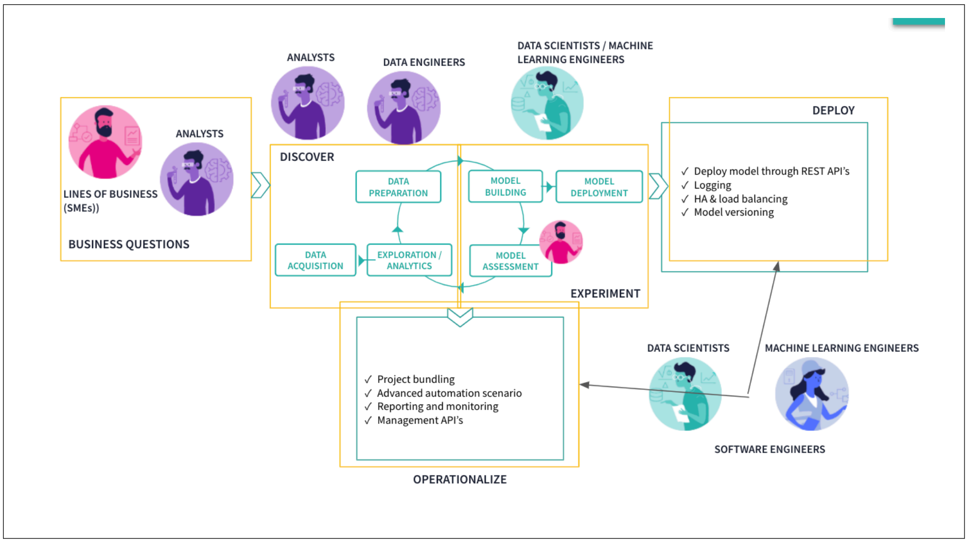


Figure 1-3. The realistic picture of a machine learning model lifecycle inside an average organization today, which involves many different people with completely different skill sets and who are often using entirely different tools.

If the definition (or even the name MLOps) sounds familiar, that's because it pulls heavily from the concept of DevOps, which streamlines the practice of software changes and updates. Indeed, the two have quite a bit in common: for example, they both center around:

Robust automation and trust between teams.

The idea of collaboration and increased communication between teams.

The end-to-end service lifecycle (build-test-release).

- Prioritizing continuous delivery as well as high quality.

Yet there is one critical difference between MLOps and DevOps that makes the latter not immediately transferable to data science teams: deploying software code in production is fundamentally different than deploying machine learning models into production. While software code is relatively static ("relatively" because many modern SaaS companies do have DevOps teams that can iterate quite quickly and deploy in production multiple times per day), data is always changing, which means machine learning models are constantly learning and adapting — or not, as the case may be — to new inputs. The complexity of this environment, including the fact that machine learning models are made up of both code as well as data, is what makes MLOps a new and unique discipline.

What About DataOps?

To add to the complexity of MLOps vs. DevOps, there is also DataOps, a term introduced in 2014 by IBM. DataOps seeks to provide business-ready data that is quickly available for use, with a large focus on data quality and metadata management. For example, if there's a sudden change in data that a model relies on, a DataOps system would alert the business team to deal more carefully with the latest insights, and the data team would be notified to investigate the change or revert a library upgrade and rebuild the related partition.

The rise of MLOps, therefore, intersects DataOps at some level, though MLOps goes a step further and brings even more robustness through additional key features (discussed in more detail in chapter 3, MLOps: Key Features).

As was the case with DevOps and later DataOps, until recently, teams have been able to get by without defined and centralized MLOps processes mostly because — at an enterprise level — they weren't deploying machine learning models into production at a large enough scale. Now, the tables are turning and teams are increasingly looking for ways to formalize a multi-stage, multi-discipline, multi-phase process with a heterogeneous environment and a framework for MLOps best practices, which is no small task. Part II of this book (***MLOps: How***) will provide this guidance.

MLOps to Mitigate Risk

MLOps is important to any team that has even one model in production, as depending on the model, continuous performance monitoring and adjusting is essential.

Think about a travel site whose pricing model would require top-notch MLOps to ensure that the model is continuously delivering business results.

However, MLOps really tips the scales as critical for risk mitigation when a centralized team (with unique reporting of its activities, meaning that there can be multiple such teams at any given enterprise) has more than a handful of operational models. At this point, it becomes difficult to have a global view of the states of these models without some standardization.

Pushing machine learning models into production without MLOps infrastructure is risky for many reasons, but first and foremost because fully assessing the performance of a machine learning model can often only be done in the production environment. Why? Because prediction models are only as good as the data they are trained on, which means the training data must be a good reflection of the data encountered in the production environment. If the production environment changes, then the model performance is likely to decrease rapidly.

Another major risk factor is that machine learning model performance is often very sensitive to the production environment it is running in, including the versions of software and operating systems they use. They tend not to be buggy in the classic software sense, because most weren't written by hand but rather were machine-generated. Instead, the problem is they are often built on a pile of open-source software (e.g., libraries — like Scikit-Learn — to Python to Linux), and having versions of this software in production that match those that the model was verified on is critically important.

Ultimately, pushing models into production is not the final step of the machine learning lifecycle and is, in fact, far from it. It's often just the beginning of monitoring its performance and ensuring that it behaves as expected. As more data scientists start pushing more machine learning models into production, MLOps becomes critical in mitigating the potential risks, which (depending on the model) can be devastating for the business.

MLOps for Responsible AI

A responsible use of machine learning (more commonly referred to as Responsible AI) covers three main dimensions:

Accountability: Ensuring that machine learning models are designed and behave in ways aligned with their purpose. Note that for publicly-traded companies in the United States, this is related to the notion of full disclosure.

Sustainability: Establishing the continued reliability of machine learning models in their operation as well as execution.

- Governability: Centrally controlling, managing, and auditing machine learning capabilities in the enterprise.

These principles may seem obvious, but it's important to consider that machine learning models lack the transparency of traditional imperative code. In other words, it is much harder to understand what features are used to determine a prediction, which in turn can make it much harder to demonstrate that models comply with the necessary regulatory or internal governance requirements.

The reality is that introducing automation vis-à-vis machine learning models shifts the fundamental onus of accountability from the bottom of the hierarchy to the top. That is, decisions that were perhaps previously made by individual contributors who operated within a margin of guidelines (for example, what the price of a given product should be or whether or not a person should be accepted for a loan) are now being made by a model. The person responsible for the automated decisions of said model is likely a data team manager or even executive, and that brings the concept of Responsible AI even more to the forefront.

Given the previously discussed risks as well as these particular challenges and principals, it's easy to see the interplay between MLOps and Responsible AI — teams must have good MLOps principles to practice Responsible AI, and Responsible AI necessitates MLOps strategies.

MLOps for Scale

MLOps isn't just important because it helps mitigate the risk of machine learning models in production, but it is also an essential component to scaling machine learning efforts (and in turn benefiting from the corresponding economies of scale). Going from one or a handful of models in production to tens, hundreds, or thousands that have a positive business impact will require MLOps discipline.

Good MLOps practices will help teams at a minimum:

Keep track of versioning, especially with experiments in the design phase.

Understand if retrained models are better than the previous versions (and promoting models to production that are performing better).

- Ensure (at defined periods — daily, monthly, etc.) that model performance is not degrading in production.

Closing Thoughts

Key features will be discussed at length in Chapter 3, but the point here is that these are not optional practices — they are essential tasks for not only efficiently scaling data science and machine learning at the enterprise level, but also doing it in a way that doesn't put the business at risk. Teams that attempt to deploy data science without proper MLOps practices in place will face issues with model quality, continuity, or worse — they will introduce models that have a real, negative impact on the business (e.g., a model that makes biased predictions that reflect poorly on the company).

MLOps is also, at a higher level, a critical part of transparent strategies for machine learning. Upper management and the C-suite should be able to understand as well as data scientists what machine learning models are deployed in production and what effect they're having on the business. Beyond that, they should arguably be able to drill down to understand the whole data pipeline behind those machine learning models. MLOps, as described in this book, can provide this level of transparency and accountability.

CHAPTER 2

People of Model Ops

Even though machine learning models are primarily built by data scientists, it's a misnomer that only data scientists can benefit from robust MLOps processes and systems. In fact, MLOps is an essential piece of Enterprise AI strategy and affects everyone working — or benefiting from — the machine learning model lifecycle.

This chapter will cover the roles each of these people play in the machine learning lifecycle, who they should ideally be connected and working together with under a top-notch MLOps program to achieve the best possible results from machine learning efforts, and what MLOps requirements they may have. Before diving into the details.

Table 2-1. provides an overview:

Role	Role in Machine Learning Model Lifecycle	MLOps Requirements
Subject Matter Experts	• Provide business questions, goals, or KPIs around which machine learning models should be framed. • Continually evaluate and ensure that model performance aligns with or resolves the initial need.	• Easy way to understand deployed model performance in business terms. • Mechanism or feedback loop for flagging model results that don't align with business expectations.
Data Scientists	• Build models that address the business question or needs brought by subject matter experts. • Deliver operationalizable models so that they can be properly used in the production environment and with production data. • Assess model quality (of both original and tests) in tandem with subject matter experts to ensure they answer initial business questions or needs.	• Automated model packaging and delivery for quick and easy (yet safe) deployment to production. • Ability to develop tests to determine the quality of deployed models and to make continual improvements. • Visibility into the performance of all deployed models (including side-by-side for tests) from one central location. • Ability to investigate data pipelines of each model to make quick assessments and adjustments regardless of who originally built the model.
Data Engineers	• Optimize the retrieval and use of data to power machine learning models.	• Visibility into performance of all deployed models. • Ability to see the full details of individual data pipelines to address underlying data plumbing issues.
Software Engineers	• Integrate machine learning models in the company's applications and systems. • Ensure that machine learning models work seamlessly with other non-machine learning-based applications.	• Versioning and automatic tests. • The ability to work in parallel on the same application.
DevOps	• Conduct and build operational systems and test for security, performance, availability. • Continuous Integration/Continuous Delivery (CI/CD) pipeline management.	• Seamless integration of MLOps into the larger DevOps strategy of the enterprise. • Seamless deployment pipeline.
Model Risk Managers / Auditors	• Minimize overall risk to the company as a result of machine learning models in production. • Ensure compliance with internal and external requirements before pushing machine learning models to production.	• Robust — likely automated — reporting tools on all models (currently or ever in production), including data lineage.
Machine Learning Architects	• Ensure a scalable and flexible environment for machine learning model pipelines, from design to development and monitoring. • Introduce new technologies when appropriate that improve machine learning model performance in production.	• High-level overview of models and their resources consumed. • Ability to drill down into data pipelines to assess and adjust infrastructure needs.

Subject Matter Experts

The first profile to consider as part of MLOps efforts are the subject matter experts; after all, the machine learning model lifecycle starts and ends with them. While the data-oriented profiles (data scientist, engineer, architect, etc.) have expertise across many areas, one where they tend to lack is a deep understanding of the business and the problems or questions at hand that need to be addressed using machine learning.

Role in the Machine Learning Model Lifecycle

Subject matter experts usually come to the table — or at least, they *should* come to the table — with clearly defined goals, business questions, and/or key performance indicators (KPIs) that they want to achieve or address. In some cases, they might be extremely well defined (e.g., "In order to hit our numbers for the quarter, we need to reduce customer churn by 10%" or "We're losing $N per quarter due to unscheduled maintenance, how can we better predict downtime?") In other cases, less so (e.g., "Our service staff needs to better understand our customers to upsell them" or "How can we get people to buy more widgets?").

In organizations with healthy processes, starting the machine learning model lifecycle with a more well-defined business question isn't necessarily always an imperative, or even an ideal scenario. Working with a less-defined business goal can be a good opportunity for subject matter experts to work directly with data scientists upfront to better frame the problem and brainstorm possible solutions before even beginning any data exploration or model experimentation.

Without this critical starting point from subject matter experts, other data professionals (particularly data scientists) risk starting the machine learning lifecycle process trying to solve problems or provide solutions that don't serve the larger business. Ultimately, this is detrimental not only to the subject matter experts who need to partner with data scientists and other data experts to build solutions, but to data scientists themselves who might struggle to provide larger value (data teams might in turn see trust in — or budgets for — data initiatives fall). Business decision modeling methodologies can be applied to formalize the business problems to be solved and frame the role of machine learning in the solution.

Business Decision Modeling

Decision modeling creates a business blueprint of the decision-making process, allowing subject matter experts to directly structure and describe their needs. Decision models can be helpful because they put machine learning in context for subject matter experts, allowing them to integrate with business rules as well to fully understand decision contexts and the potential impact of model changes.

MLOps strategies that include a component of business decision modeling for subject matter experts can be an effective tool for ensuring real-world machine learning model results are properly contextualized for those that don't have deep knowledge of how the underlying models themselves work. Further reading on building decision requirement models.

Subject matter experts have a role to play not only at the beginning of the machine learning model lifecycle, but the end (post-production) as well. Oftentimes, to understand if a machine learning model is performing well or as expected, data scientists need subject matter experts to close the feedback loop — traditional metrics (accuracy, precision, recall, etc.) are not enough.

For example, data scientists could build a simple churn prediction model that has very high accuracy in a production environment; however, marketing does not manage to prevent anyone from churning. From a business perspective, that means the model didn't work, and that's important information that needs to make its way back to those building the machine learning model so that they can find another possible solution — e.g., introducing uplift modeling that helps marketing better target potential churners who might be receptive to marketing messaging.

Role In and Needs From MLOps

Given their role in the machine learning model lifecycle, it's critical when building MLOps processes for subject matter experts to have an easy way to understand deployed model performance in business terms. That is, not just model accuracy, precision, and recall, but its results or impact on the business process identified upfront. In addition, when there are unexpected shifts in performance, subject matter experts need a scalable way through MLOps processes for flagging model results that don't align with business expectations.

On top of these explicit feedback mechanisms, more generally, MLOps should be built in a way that increases transparency for subject matter experts. That is, they should be able to use MLOps processes as a jumping-off point for exploring the data pipelines behind the models, understanding what data is being used, how it's being transformed and enhanced, and what kind of machine learning techniques are being applied.

For subject matter experts who are also concerned with compliance of machine learning models with regulations (either internal or external), MLOps serves as an additional way to bring transparency and understanding to these processes. This includes being able to dig into individual decisions made by a model to understand why the model came to that decision — this should be complementary to statistical and aggregated feedback.

Ultimately, MLOps is most relevant for subject matter experts as a feedback mechanism and a platform for communication with data scientists about the models they are building. However, there are other MLOps needs as well — specifically around transparency, which ties up into Responsible AI — that are relevant for subject matter experts and make them an important part of the MLOps picture.

Data Scientists

The needs of data scientists are the most critical ones to consider when building an MLOps strategy. To be sure, they have a lot to gain; data scientists at most organizations today are often dealing with siloed data, processes, and tools, making it difficult to effectively scale their efforts. MLOps is well positioned to change this.

Role in the Machine Learning Model Lifecycle

Though most see data scientists' role in the machine learning model lifecycle as strictly the model building portion, it is actually — or at least, it should be — much wider. From the very beginning, data scientists need to be involved with subject matter experts, understanding and helping to frame business problems in such a way that they can build a viable machine learning solution.

The reality is that this very first, critical step in the machine learning model lifecycle is often the hardest. It's challenging particularly for data scientists first and foremost because it's not where their training lies; that is, both formal and informal data science programs in universities and online heavily emphasize technical skill and not necessarily skills for communicating effectively with subject matter experts from the business side of the house who usually are not intimately familiar with machine learning techniques. Once again, business decision modeling techniques can help here.

It's also a challenge because it can take time — for data scientists who want to dive in and get their hands dirty, spending weeks framing and outlining the problem before getting started on solving it can be torture. To top it all off, data scientists are often siloed (physically, culturally, or both) from the core of the business and from subject matter experts, so they simply don't have the organizational infrastructure that facilitates easy collaboration between these profiles. Robust MLOps systems can help address some of these challenges.

After overcoming the first hurdle, depending on the organization, the project might get handed off to either data engineers or analysts to do some of the initial data gathering, preparation, and exploration. In some cases, data scientists themselves manage these parts of the machine learning model lifecycle. But in any case, data scientists step back in when it comes time to build, test, robustify, and then deploy the model.

Following deployment, data scientists' roles include constantly assessing model quality to ensure the way it's working in production answers initial business questions or needs. The underlying question in many organizations is often whether data scientists monitor only the models they have had a hand in building, or if there is one person who handles all monitoring. In the former scenario, what happens when there is staff turnover? In the latter scenario, building good MLOps practices is critical, as the person monitoring also needs to quickly be able to jump in and take action should the model drift and start negatively affecting the business. If they weren't the ones who built it, how can MLOps make this process seamless?

Operationalization and MLOps

Throughout 2018 and the beginning of 2019, operationalization was the key buzzword when it came to machine learning model lifecycles and AI in the enterprise. Put simply, operationalization of data science is the process of pushing models to production and measuring their performance against business goals. So how does operationalization fit into the MLOps story? MLOps takes operationalization one step further, encompassing not just the push to production but the maintenance of those models — and the entire data pipeline — in production.

Though they are distinct, MLOps might be considered the new operationalization. That is, where many of the major hurdles for businesses to operationalize have disappeared, MLOps is the next frontier and presents the next big challenge for machine learning efforts in the enterprise.

Role In and Needs From MLOps

All of the questions in the previous section lead directly here: data scientists' needs when it comes to MLOps. Starting from the end of the process and working backwards, MLOps must provide data scientists with visibility into the performance of all deployed models as well as any models being A/B tested. But taking that one step further, it's not just about monitoring — it's also about action. Top-notch MLOps should also allow data scientists the flexibility to select winning models from tests and easily deploy them.

Transparency is an overarching theme in MLOps, so it's no surprise that it's also a key need for data scientists. The ability to drill down into data pipelines and to make quick assessments and adjustments (regardless of who originally built the model) is critical. Automated model packaging and delivery for quick and easy (yet safe) deployment to production is another important point for transparency, and it's a crucial component of MLOps, especially to bring data scientists together to a place of trust with software engineers and DevOps teams.

In addition to transparency, perhaps another theme for mastering MLOps — especially when it comes to meeting the needs of data scientists — is pure efficiency. In an enterprise setting, agility and speed matter. It's true for DevOps, and the story for MLOps is no different. Of course, data scientists can deploy, test, and monitor models in an ad-hoc fashion. But they will lose enormous amounts of time re-inventing the wheel with every single machine learning model, and that will never add up to scalable machine learning processes for the organization.

Data Engineers

Data pipelines are at the core of the machine learning model lifecycle, and data engineers are, in turn, at the core of data pipelines. Because data pipelines can be abstract and complex, data engineers have a lot of efficiencies to gain from MLOps.

Role in the Machine Learning Model Lifecycle

In large organizations, managing the flow of data itself outside of the application of machine learning models is a full-time job. Depending on the technical stack and organizational structure of the enterprise, data engineers might, therefore, be more focused on databases themselves than on pipelines (especially if the company is leveraging data science and machine learning platforms that facilitate the visual building of pipelines by other data practitioners, like business analysts).

Ultimately, despite these slight variations in the role by an organization, the role of data engineers in the lifecycle is to optimize the retrieval and use of data to eventually power machine learning models. Generally, this means working closely with business teams, particularly subject matter experts, to identify the right data for the project at hand and possibly also prepare it for use. On the other end, they work closely with data scientists as well to resolve any data plumbing issues that might cause a model to behave undesirably in production.

Role In and Needs From MLOps

Given data engineers' central role in the machine learning model lifecycle, underpinning both the building and monitoring portions, MLOps can bring significant efficiency gains. Data engineers will namely require not only visibility into the performance of all models deployed in production, but the ability to take it one step further and directly drill down into individual data pipelines to address any underlying issues.

Ideally, for maximum efficiency for the data engineer profile (and for others as well - including data scientists), MLOps must not consist of simple monitoring, but be a bridge to underlying systems for investigating and tweaking machine learning models.

Software Engineers

It would be easy to exclude classical software engineers from MLOps consideration, but it is crucial from a wider organizational perspective to consider their needs to build a cohesive enterprise-wide strategy for machine learning.

Role in the Machine Learning Model Lifecycle

Software engineers usually aren't building machine learning models, but on the other hand, most organizations are not *only* producing machine learning models, but classic software and applications as well. It's important that software engineers and data scientists work together to ensure the functioning of the larger system. After all, machine learning models aren't just stand-alone experiments; the machine learning code, training, testing, and deployment has to fit into the CI/CD pipelines that the rest of the software is using.

For example, consider a retail company that has built a machine learning-based recommendation engine for their website. The machine learning model was built by the data scientist, but to integrate it into the larger functioning of the site, software engineers will necessarily need to be involved. Similarly, software engineers are responsible for the maintenance of the website as a whole, and a large part of that includes the functioning of the machine learning models in production.

Role In and Needs From MLOps

Given this interplay, software engineers need MLOps to provide them with model performance details as a larger picture of software application performance for the enterprise. MLOps is a way for data scientists and software engineers to speak the same language and have the same baseline understanding of how different models deployed across the silos of the enterprise are working together in production.

Other important features for software engineers include versioning, in order to be sure of what they are currently dealing with; automatic tests, in order to be as sure as possible that what they are currently dealing with is working; and the ability to work in parallel on the same application (thanks to a system that allows branches and merges like Git).

DevOps

MLOps was born out of DevOps principles, but that doesn't mean they can be run in parallel as completely separate and siloed systems.

Role in the Machine Learning Model Lifecycle

DevOps teams have two primary roles in the machine learning model lifecycle: first, they are the people conducting and building operational systems as well as tests to ensure security, performance, and availability of machine learning models. Secondly, they are responsible for CI/CD pipeline management. Both of these roles require tight collaboration with data scientists as well as data engineers and data architects. Tight collaboration is, of course, easier said than done, but that is where MLOps can add value.

Role In and Needs From MLOps

For DevOps teams, MLOps needs to be integrated into the larger DevOps strategy of the enterprise, bridging the gap between traditional CI/CD and modern machine learning. That means systems that are fundamentally complementary and that allow DevOps teams to automate tests for machine learning just as they can automate tests for traditional software.

Model Risk Manager/Auditor

In certain industries (particularly the financial services sector), the model risk management (MRM) function is crucial for regulatory compliance. But it's not only highly-regulated industries that should be concerned or that should have a similar function; MRM can protect companies in any industry from catastrophic loss introduced by poorly performing machine learning models. What's more, audits play a role in many industries and can be labor-intensive, which is where MLOps comes into the picture.

Role in the Machine Learning Model Lifecycle

When it comes to the machine learning model lifecycle, model risk managers play the critical role of analyzing not just model outcomes, but the initial goal and business questions machine learning models seek to resolve to minimize overall risk to the company. They should be involved along with subject matter experts at the very beginning of the lifecycle to ensure that an automated, machine learning-based approach in and of itself doesn't present risk.

And, of course, they have a role to play in monitoring — their more traditional place in the model lifecycle — to ensure that risk stays at bay once models are in production. In between conception and monitoring, MRM also is a factor post-model development and pre-production, ensuring initial compliance with internal and external requirements.

Role In and Needs From MLOps

MRM professionals and teams have a lot to gain from MLOps, as their work is often painstakingly manual, and as teams often use different tools, standardization can offer a huge leg-up in the speed at which auditing and risk management can occur.

When it comes to specific MLOps needs, robust reporting tools on all models — whether they are currently in production or have been in production in the past — is the primary one. This reporting should include not just performance details, but the ability to see data lineage. Automated reporting adds an extra layer of efficiency for MRM and audit teams in MLOps systems and processes.

Machine Learning Architect

Traditional data architects are responsible for understanding the overall enterprise architecture and ensuring that it meets the requirements for data needs from across the business. They generally play a role in defining how data will be stored and consumed.

Today, demands on architects are much greater, and they often have to be knowledgeable not only on the ins and outs of data storage and consumption, but on how machine learning models work in tandem. This adds a lot of complexity to the role and increases their responsibility in the MLOps lifecycle, and it's why in this section, we have called them machine learning architects instead of the more traditional data architect title.

Role in the Machine Learning Model Lifecycle

Machine learning architects play a critical role in the machine learning model lifecycle, ensuring a scalable and flexible environment for model pipelines. In addition, data teams need their expertise to introduce new technologies (when appropriate) that improve machine learning model performance in production. It is for this reason that the data architect title isn't enough; they need to have an intimate understanding of machine learning — not just enterprise architecture — to play this key role in the machine learning model lifecycle.

This role requires collaboration across the enterprise, from data scientists and engineers to DevOps and software engineers. Without a complete understanding of the needs of each of these people and teams, machine learning architects cannot properly allocate resources to ensure optimal performance of machine learning models in production.

Role In and Needs From MLOps

When it comes to MLOps, machine learning engineers' role is about having a centralized view of resource allocation. As they have a strategic, tactical role, they need an overview of the situation to identify bottlenecks and use that information to find long-term improvements. Their role is one of pinpointing possible new technology or infrastructure for investment, not necessarily operational quick fixes that don't address the heart of the scalability of the system.

Closing Thoughts

MLOps isn't just for data scientists; a diverse group of experts across the organization have a role to play not only in the machine learning model lifecycle, but the MLOps strategy as well. In fact, each person — from the subject matter expert on the business side to the most technical machine learning architect — plays a critical part in the maintenance of machine learning models in production. This is ultimately important not only to ensure the best possible results from machine learning models (good results generally lead to more trust in machine learning-based systems as well as increased budget to build more), but perhaps more pointedly, to protect the business from the risks outlined in Chapter 1.

CHAPTER 3

Key MLOps Features

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 third 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 *mlops@dataiku.com*.

As discussed in Chapters 1 and 2, MLOps affects many different roles across the organization and, in turn, many parts of the machine learning lifecycle. This chapter will introduce the five key components of MLOps (development, deployment, monitoring, iteration, and governance) at a high level as a foundation for Chapters 4-8, which delve into the more technical details and requirements of these components.

A Primer on Machine Learning

To understand the key features of MLOps, it's essential first to understand how machine learning works and be intimately familiar with its specificities. Though often overlooked in its role as a part of MLOps, ultimately, algorithm selection (or how machine learning models are built) can have a direct impact on MLOps processes.

At its core, machine learning (ML) is the science of computer algorithms that automatically learn and improve from experience rather than being explicitly programmed. The algorithms analyze sample data—known as training data—to build a model in software that can make predictions.

For example, an image recognition model might be able to identify the type of electricity meter from a photograph by searching for key patterns in the image that distinguish each type of meter. Another concrete example is an insurance recommender model, which might suggest additional insurance products that a specific existing customer is most likely to buy based on the previous behavior of similar customers.

When faced with unseen data, be it a photo or a customer, the ML model uses what it has learned from previous data to make the best prediction it can based on the assumption that the unseen data is somehow related to the previous data.

ML algorithms use a wide range of mathematical techniques and the models and take many different forms, from simple decision trees to logistic regression algorithms to much more complex deep learning models. Machine learning algorithms can tackle problems that were either infeasible or too costly with previous software algorithms. While ML techniques do need large amounts of data and compute power to build their models, these resources are now available at increasingly low costs.

Different ML Algorithms, Different MLOps Challenges

What ML algorithms all have in common is that they model patterns in past data to make predictions, and the quality and relevance of this past experience is the key factor in their effectiveness. Where they differ is that each style of model has specific characteristics and presents different challenges in MLOps (outlined in Table 3-1).

Type	*Name*	*MLOps Considerations*
Linear	Linear Regression	There is a tendency for overfitting
	Logistic Regression	
Tree-Based	Decision Tree	Can be unstable—small changes in data can lead to a large change in the structure of the optimal decision tree.
	Random Forest	Predictions can be difficult to understand, which is challenging from a Responsible AI perspective. Random Forest models can also be relatively slow to output predictions, which can present challenges for applications.
	Gradient Boosting	Like Random Forest, predictions can be difficult to understand. Also, a small change in the feature or training set can create radical changes in the model.
Deep Learning	Neural Networks	In terms of interpretability, deep learning models are almost impossible to understand. Deep learning algorithms, including neural networks, are also extremely slow to train and require a lot of power (and data). Is it worth the resources, or would a simpler model work just as well?

Model Development

Table 3-1 shows why the development, specifically choice of model, can ultimately affect MLOps. This section will take a deeper look into ML model development as a whole for an even more complete understanding of its components, all of which can have an impact on MLOps after deployment.

Establishing Business Objectives

The process of developing a machine learning model typically starts with a business objective, which can be as simple as reducing fraudulent transactions to $< 0.1\%$ or having the ability to identify people's faces on their social media photos. Business objectives naturally come with performance targets, technical infrastructure requirements, and cost constraints; all of these factors can be captured as key performance indicators, or KPIs, which will ultimately enable the business performance of models in production to be monitored.

It's important to recognize that ML projects don't happen in a vacuum—they are generally part of a larger project that in turn impacts technologies, processes, and people. That means part of establishing objectives also includes change management, which may even provide some guidance for how the ML model should be built. For example, the required degree of transparency will strongly influence the choice of algorithms and may drive the need to provide explanations together with predictions so that predictions are turned into valuable decisions at the business level.

Data Sources and Exploratory Data Analysis

With clear business objectives defined, it is time to bring together subject matter expert(s) and data scientist(s) to begin the journey of developing the ML model. This starts with the search for suitable input data. Finding data sounds simple, but in practice, it can be the most arduous part of the journey.

Key questions for finding data to build ML models include:

- What relevant datasets are available?
- Is this data sufficiently accurate and reliable?
- How can stakeholders get access to this data?
- What data properties (known as features) can be made available by combining multiple sources of data?
- Will this data be available in real time?
- Is there a need to label some of the data with the "ground truth" that is to be predicted? If so, how much will this cost in terms of time and resources?
- How will data be updated once the model is deployed?
- Will the use of the model itself reduce the representativeness of the data?
- How will the KPIs, which were established along with the business objectives, be measured?

- The constraints of data governance bring even more questions, including:
- Can the selected datasets be used for this purpose?
- What are the terms of use?
- Is there personally identifiable information (PII) that must be redacted or anonymized?
- Are there features, such as gender, that legally cannot be used in this business context?

Since data is the essential ingredient to power ML algorithms, it always helps to build an understanding of the patterns in data before attempting to train models. Exploratory data analysis (EDA) techniques can help build hypotheses about the data, identify data cleaning requirements, and inform the process of selecting potentially significant features. EDA can be carried out visually for intuitive insight and statistically if more rigor is required.

Feature Selection and Engineering

EDA leads naturally into feature selection and feature engineering. This step is the most intensive in terms of data handling, but it is also critically important, as feature engineering can make significant improvements to the classic algorithms of the SciKit Learn library, including Random Forest (decision trees), logistic regression, and support-vector machines (SVMs).

Feature Engineering

Feature engineering is the process of taking raw data from the selected datasets and transforming it into "features" that better represent the underlying problem to be solved. It includes data cleansing, which can represent the largest part of an ML project in terms of time spent.

Feature engineering can also improve the performance of deep learning algorithms in many use cases where feature detection is part of the algorithm itself. For example, organizing the representation of colors in an image can be important.

Spending time to build business-friendly features will often improve the final performance and ease the adoption by end users, as model explanations are likely to be simpler. It also reduces modeling debt, allowing data scientists to understand the main prediction drivers and ensure that they are robust. Of course, there are trade-offs to consider between the cost of time spent to "understand" the model and the expected value as well as risks associated with the model's use.

Training

After data preparation by way of feature engineering and selection, the next step is training. The process of training and optimizing a new ML model is iterative; several algorithms may be tested, features can be automatically generated, feature selections may be adapted, and algorithm hyper parameters tuned. In addition to—or in many cases because of—its iterative nature, training is also the most intensive step of the ML model lifecycle when it comes to computing power.

Some ML algorithms can best support specific use cases, but governance considerations may also play a part in the choice of algorithm. In particular, highly regulated environments where decisions must be explained (e.g., financial services) cannot use opaque algorithms, like neural networks, and have to favor simpler techniques, such as decision trees. In many use cases, it's not so much a trade-off on performance but rather a trade-off on cost. That is, simpler techniques usually require more costly manual feature engineering to reach the same level of performance as more complex techniques.

Keeping track of the results of each experiment when iterating becomes complex quickly. Nothing is more frustrating than a data scientist not being able to recreate the best results because (s)he cannot remember the precise configuration. An experiment tracking tool can greatly simplify the process of remembering the data, the features selection, and model parameters alongside the performance metrics. These enable experiments to be compared side-by-side, highlighting the differences in performance.

Reproducibility

While many experiments may be short-lived, significant versions of a model need to be saved for possible later use. The challenge here is reproducibility, which is an important concept in experimental science in general. The aim in ML is to save enough information about the environment the model was developed in so that the model can be reproduced with the same results again from scratch.

Without reproducibility, data scientists have little chance of being able to confidently iterate on models, and worse, they are unlikely to be able to hand over the model to DevOps to see if what was created in the lab can be faithfully reproduced in production. True reproducibility requires version control of all of the assets and parameters involved, including the data used to train and evaluate the model, as well as a record of the software environment.

Where large amounts of data are involved, versioning can be problematic; data snapshots may be too large, and increment data versioning technologies are in their infancy. Compromises may be necessary (such as storing sampled data).

Additional Considerations for Model Development

Being able to reproduce the model is only part of the operationalization challenge—the DevOps team also needs to understand how to verify the model (i.e., what does the model do, how should it be tested, and what are the expected results?). Those in highly regulated industries are likely required to document even more detail, including how the model was built and how it was tuned. In critical cases, the model may be independently re-coded and rebuilt.

Documentation is still the standard solution to this communication challenge. Automated model document generation can make the task less onerous, but in almost all cases, some documentation will need to be written by hand to explain the choices made.

ML models are challenging to understand—it is a fundamental consequence of their statistical nature. While model algorithms come with standard performance measures to assess their efficacy, these don't explain how the predictions are made. The "how" is important as a way to sanity-check the model or help better engineer features, and it may be necessary to ensure that fairness requirements (e.g., around features like sex, age, or race) have been met. This is the field of explainability, which is connected to Responsible AI as discussed in Chapter 1 (and which will be discussed in further detail in Chapter 4).

Explainability techniques are becoming increasingly important as global concerns grow about the impact of unbridled AI. They offer a way to mitigate uncertainty and help prevent unintended consequences. The techniques most commonly use today include:

Partial dependencies plots, which look at the marginal impact of features on the predicted outcome.

Subpopulation analyses, which look at how the model treats specific subpopulations and that are the basis of many fairness analyses.

Individual model predictions, such as Shapley Values, which explain how the value of each feature contributes to a specific prediction.

Productionalization and Deployment

Productionalizing and deploying models is a key component of MLOps that presents an entirely different set of technical challenges than developing the model. It is the domain of the software engineer and the DevOps team, and the organizational challenges in managing the information exchange between the data scientist(s) and these teams must not be underestimated. As touched on in Chapter 1, without effective collaboration between the teams, delays or failures to deploy are inevitable.

Model Deployment Types and Contents

To understand what happens in these phases, it's helpful to take a step back and ask: what exactly is going into production, and what does a model consist of?

There are commonly two types of model deployment.

Model-as-a-service, or live-scoring model. Typically the model is deployed into a simple framework to provide a REST API endpoint that responds to requests in real time.

Embedded model. Here the model is packaged into an application, which is then published. A common example is an application that provides batch-scoring of requests.

What to-be-deployed models consist of depends, of course, on the technology chosen, but typically they comprise a set of code (commonly Python, R, or Java) and data artifacts. Any of these will have version dependencies on runtimes and packages that need to match in the production environment—the use of different versions may cause model predictions to differ.

One approach to reducing the dependencies on the production environment is to export the model to a portable format such as PMML, PFA, ONNX, POJO, or MOJO. These aim to improve model portability between systems and simplify deployment. However, they come at a cost: each format supports a limited range of algorithms, and sometimes the portable models behave in subtly different ways than the original. They are a choice to be made based on a thorough understanding of the technological and business context.

Containerization

Containerization is an increasingly popular solution to the headaches of dependencies when deploying ML models. Container technologies such as Docker are lightweight alternatives to virtual machines, allowing applications to be deployed in independent, self-contained environments, matching the exact requirements of each model.

They also enable new models to be seamlessly deployed using the Blue-Green Deployment technique. Model compute resources can be scaled elastically using multiple containers, too. Orchestrating many containers is the role of technologies such as Kubernetes and can be used both on-cloud and on-premise.

Model Deployment Requirements

So what about the productionalization process between completing model development and physically deploying into production. What needs to be addressed? Rapid, automated deployment will always be preferred to labor-intensive processes.

For short-lifetime, self-service applications, there often isn't much need to worry about testing and validation. If the maximum resource demands of the model can be securely capped by technologies such as Linux cgroups, then a fully automated single-step push-to-production may be entirely adequate. It is even possible to handle simple user Interfaces with frameworks like Flask when using this lightweight deployment mode. Along with integrated data science and machine learning platforms (e.g., Dataiku), some business rule management systems may also allow some sort of autonomous deployment of basic ML models.

In customer-facing, mission-critical use cases, a more robust CI/CD pipeline is required. This will typically involve:

- Ensuring all coding, documentation and sign-off standards have been met
- Recreating the model in something approaching the production environment
- Re-validating the model accuracy
- Explainability checks
- Ensuring all governance requirements have been met
- Checking the quality of any data artifacts
- Testing resource usage under load
- Embedding into a more complex application, including integration tests

In heavily regulated industries (e.g., finance and pharmaceuticals), governance and regulatory checks will be extensive and are likely to involve manual intervention. The desire in MLOps, just as in DevOps, is to automate the CI/CD pipeline as far as possible. This not only speeds up the deployment process, but enables more extensive regression testing and reduces the likelihood of errors in the deployment.

Monitoring

Once a model is deployed to production, it is crucial that it continues to perform well over time. But good performance means different things to different people, in particular to the DevOps team, to data scientists, and to the business.

DevOps Concerns

The concerns of the DevOps team are very familiar and include questions like:

- Is the model getting the job done quickly enough?
- Is it using a sensible amount of memory and processing time?

This is traditional IT performance monitoring, and DevOps teams know how to do this well already. The resource demands of ML models are not so different from traditional software in this respect.

Scalability of compute resources can be an important consideration, for example, if you are retraining models in production. Deep learning models will have greater resource demands than much simpler decision trees. But overall, the existing expertise in DevOps teams for monitoring and managing resources can be readily applied to ML models.

Data Scientist Concerns

The data scientist is interested in monitoring ML models for a new, more challenging reason: they can degrade over time, since ML models are effectively models of the data they were trained on. This is not a problem faced by traditional software, but it is inherent to machine learning. ML mathematics builds a concise representation of the important patterns in the training data with the hope that this is a good reflection of the real world. If the training data reflects the real world well, then the model should be accurate and, thus, useful.

But the real-world doesn't stand still. The training data used to build a fraud detection model six months ago won't reflect a new type of fraud that has started to occur in the last three months. If a given website starts to attract an increasingly younger user base, then a model that generates advertisements is likely to produce less and less relevant adverts.

At some point, the performance will be unacceptable, and model retraining becomes necessary. How soon models need to be retrained will depend on how fast the real world is changing and how accurate the model needs to be, but also—importantly—how easy it is to build and deploy a better model.

But first, how can data scientists tell a model's performance is degrading? It's not always easy. There are two common approaches, one based on ground truth and the other on input drift.

Ground Truth

The ground truth, put simply, is the correct answer to the question that the model was asked to solve. For example, "Is this credit card transaction actually fraudulent?" In knowing the ground truth for all the predictions a model has made, one can judge how well that model is performing.

Sometimes ground truth is obtained rapidly after a prediction—for example, in models deciding which advertisements to display to a user on a webpage. The user is likely to click on the advertisements within seconds, or not at all.

However, in many use cases, obtaining the ground truth is much slower. If a model predicts that a transaction is fraudulent, how can this be confirmed? In some cases, verification may only take a few minutes, such as a phone call placed to the cardholder. But what about the transactions the model thought were OK but actually weren't? The best hope is that they will be reported by the cardholder when they review their monthly transactions, but this could happen up to a month after the event (or not at all).

In the fraud example, ground truth isn't going to enable data science teams to monitor performance accurately on a daily basis. If the situation requires rapid feedback, then input drift may be a better approach.

Input Drift

Input drift is based on the principle that a model is only going to predict accurately if the data it was trained on is an accurate reflection of the real world. So if a comparison of recent requests to a deployed model against the training data shows distinct differences, then there is a strong likelihood that the model performance is compromised. This is the basis of input drift monitoring. The beauty of this approach is that all the data required for this test already exists—no need to wait for ground truth or any other information.

Chapter 7 will go further in technical depth about data scientists' concerns when it comes to model monitoring, as it is one of the most important components of an agile MLOps—and indeed an overall agile Enterprise AI—strategy.

Business Concerns

The business has a holistic outlook on monitoring, and some of their concerns might include questions like:

- Is the model delivering value to the enterprise?
- Do the benefits of the model outweigh the cost of developing and deploying the model? (And how can we measure this?)

The KPIs identified for the original business objective are one part of this process. Where possible, these should be monitored automatically, but this is rarely trivial. The previous example objective of reducing fraud to less than 0.1% of transactions is reliant on establishing the ground truth. But even monitoring this doesn't answer the question: what is the net gain to the business in dollars?

This is an age-old challenge for software, but with ever-increasing expenditure on ML, the pressure for data scientists to demonstrate value is only going to grow. In the absence of a *dollar-o-meter*, effectively monitoring the business KPIs is the best option available. The choice of the baseline is important here and should ideally allow for differentiation of the value of the ML subproject specifically and not of the global project. For example, the ML performance can be assessed with respect to a rule-based decision model based on subject matter expertise to set apart the contribution of decision automation and of ML per se.

Iteration and Lifecycle

Developing and deploying improved versions of a model is an essential part of the MLOps lifecycle, and one of the more challenging. There are various reasons to develop a new model version, one of which is model performance degradation due to model drift, as discussed in the prior section. Sometimes there is a need to reflect refined business objectives and KPIs, and other times, it's just that the data scientists have come up with a better way to design the model.

Iteration

In some fast-moving business environments, new training data becomes available every day, Daily retraining and redeployment of the model are often automated to ensure that the model reflects recent experience as closely as possible.

Retraining an existing model with the latest training data is the simplest scenario for iterating a new model version. But while there are no changes to feature selection or algorithm, there are still plenty of pitfalls. In particular:

- Is the new training data sane? Automated validation of the new data through pre-defined metrics and checks is essential.
- Is the data complete and consistent?
- Are the distributions of features broadly similar to those in the previous training set? Remember that the goal is to refine the model, not radically change it.

With a new model version built, the next step is to compare the metrics with the current live model version. Doing so requires evaluating both modes on the same development dataset, whether it be the previous or latest version. If metrics and checks suggest a wide variation between the models, automated scripts should not redeploy, but seek manual intervention.

Even in the "simple" automated retraining scenario with new training data, there is a need for multiple development datasets based on scoring data reconciliation (with ground truth when it becomes available), data cleaning and validation, the previous model version, and a set of carefully considered checks. Retraining in other scenarios is likely to be even more complicated, rendering automated redeployment unlikely.

As an example, consider retraining motivated by the detection of significant input drift. How can the model be improved? If new training data is available, then retraining with this data is the action with the highest benefit-cost ratio, and it may suffice. However, in environments where it's slow to obtain the ground truth, there may be little new labeled data.

This case requires direct invention from data scientists who need to understand the cause of the drift and work out how the existing training data could be adjusted to more accurately reflect the latest input data. Evaluating a model generated by such changes is difficult. The data scientist will have to spend time assessing the situation —time that increases with the amount of modeling debt—as well as estimate the potential impact on performance and design custom mitigations measures. For example, removing a specific feature or sampling the existing rows of training data may lead to a better-tuned model.

The Feedback Loop

DevOps best practice inside large enterprises will typically dictate that the live model scoring environment and the model retraining environment are distinct. As a result, the evaluation of a new model version on the retraining environment is likely to be compromised.

One approach to mitigating this uncertainty is shadow testing, where the new model version is deployed into the live environment alongside the existing deployed model. All live scoring is handled by the incumbent model version, but each new request is then scored again by the new model version and the results logged—but not returned to the requestor. Once sufficient requests have been scored by both versions, the results can be compared statistically. Shadow scoring also gives more visibility to the SMEs on the future versions of the model and may thus allow for a smoother transition.

In the advertisement generation model previously discussed, it is impossible to tell if the ads selected by the model are good or bad without allowing the end user the chance to click on them. In this use case, shadow testing has limited benefits, and A/B Testing is more common.

In A/B testing, both models are deployed into the live environment, but input requests are split between the two models. Any request is processed by one or the other model, not both. Results from the two models are logged for analysis (but never for the same request). Drawing statistically meaningful conclusions from an A/B requires careful planning of the test.

Chapter 7 will cover the how-to of A/B testing in more detail, but as a preview, the simplest form of A/B testing is often referred to as a fixed-horizon test. That's because, in the search for a statistically meaningful conclusion, one has to wait until the carefully predetermined number of samples have been tested before concluding a result. "Peeking" at the result before the test is finished is unreliable. However, the test is running live, and in a commercial environment, every bad prediction is likely to cost money, so not being able to stop a test early could be expensive.

Bayesian, and in particular multi-arm bandit tests, are an increasingly popular alternative to the "frequentist" fixed-horizon test, with the aim of drawing conclusions more quickly. Multi-arm bandit testing is adaptive—the algorithm that decides the split between models adapts according to live results and reduces the workload of underperforming models. While multi-arm bandit testing is more complex, it can reduce the business cost of sending traffic to a poorly performing model.

Iterating on an ML model deployed to millions of devices, such as smartphones, sensors, or cars, presents different challenges to iteration in a corporate IT environment. One approach is to relay all the feedback from the millions of model instances to a central point and perform training centrally. Tesla's autopilot system, running in over 500,000 cars, does exactly this. Full retraining of their 50 or so neural networks takes 70,000 GPU hours.

Google has taken a different approach with its smartphone keyboard software GBoard. Instead of centralized retraining, every smartphone retrains the model locally and sends a summary of the improvements it has found to Google centrally. These improvements from every device are averaged and the shared model updated. This federated learning approach means that an individual user's personal data doesn't need to be collected centrally, the improved model on each phone can be used immediately, and the overall power consumption goes down.

Governance

Governance is a theme that has already been touched upon many times throughout the first two chapters of this book (and this won't be the last time, as Chapter 8 will go into even more depth on the "how."). This section will levelset on the definition and components of governance.

Governance is the set of controls placed on a business to ensure that it delivers on its responsibilities to all stakeholders, from shareholders and employees to the public and national governments. These responsibilities include financial, legal, and ethical obligations. Underpinning all three of these is the fundamental principle of fairness.

Legal obligations are the easiest to understand. Businesses were constrained by regulations long before the advent of machine learning. Many regulations target specific industries; for example, financial regulations aim to protect the public and wider economy from finance mismanagement and fraud, while pharmaceutical industries must comply with rules to safeguard the health of the public. Business practice is impacted by broader legislation to protect vulnerable sectors of society and to ensure a level playing field on criteria such as sex, race, age, or religion.

Recently, governments across the world have imposed regulations to protect the public from the impact of the use of personal data by businesses. The 2016 EU General Data Protection Regulation (GDPR) and the 2018 California Consumer Privacy Act (CCPA) typify this trend, and their impact on ML—with its total dependency on data—has been immense.

GDPR attempts to protect personal data from industrial misuse and so limit the potential discrimination against individuals.

The GDPR sets out principles for the processing of personal data, and it's worth noting that the CCPA was built to closely mirror its principles, though it does have some significant differences. Processing includes the collection, storage, alteration, and use of personal data. These principles are:

- Lawfulness, fairness, and transparency
- Purpose limitation
- Data minimization
- Accuracy
- Storage limitation
- Integrity and confidentiality (security)
- Accountability

Governments are now starting to turn their regulatory eye to ML specifically, hoping to mitigate the negative impact of its use. The European Union is leading the way with planned legislation to define the acceptable uses of various forms of AI. This is not necessarily about reducing use—for example, it may enable beneficial applications of facial recognition technology that are currently restricted by data privacy regulations. But what is clear is that the businesses will have to take heed of yet more regulation when applying ML.

Do businesses care about moral responsibilities to society, beyond formal legislation? Increasingly, the answer is yes, as seen in the current development of Environmental Social and Governance (ESG) performance indicators. Trust matters to consumers, and a lack of trust is bad for business. With increasing public activism on the subject, businesses are engaging with ideas of Responsible AI, the ethical, transparent and accountable application of AI technology. Trust matters to shareholders too, and full disclosure of ML risks is on its way.

Governance and MLOps

Applying good governance to MLOPs is challenging. The processes are complex, the technology is opaque, and the dependence on data is fundamental. Governance initiatives in MLOps broadly fall into one of two categories:

1. Process governance: The use of well-defined processes to ensure all governance considerations have been addressed at the correct point in the lifecycle of the model and that a full and accurate record has been kept.
2. Data governance: A framework for ensuring appropriate use and management of data.

Data Governance

Data governance concerns itself with the data being used, especially that for model training, and it can address questions like:

- What is the data's provenance?
- How was the original data collected and under what terms of use?
- Is the data accurate and up to date?
- Is there Personally Identifiable Information (PII) or other forms of sensitive data that should not be used?

ML data pipelines usually involve significant pipelines of data cleaning, combination, and transformation. Understanding the data lineage is complex, especially at the feature level, but is essential for compliance with GDPR style regulations. How can teams—and more broadly organizations, as it matters at the top as well—be sure that no PII is used to train this model?

Anonymizing or pseudo-anonymizing data is not always a sufficient solution to managing personal information. If not performed correctly, it can still be possible to single out an individual and their data, contrary to the requirements of GDPR.

Inappropriate biases in models can arise quite accidentally despite the best intentions of data scientists. An ML recruitment model famously discriminated against women by identifying that certain schools—all-female schools—were less well represented in the company's upper management, which reflected the historical dominance of men in the organization. The point is that making predictions based on past experience is a powerful technique, but sometimes the consequences are not only counterproductive, but illegal.

Data governance tools that can address these problems are in their infancy. Most focus on answering the two questions of data lineage:

1. Where did the information in this dataset come from, and what does this tell me about how I can use it?
2. How is this dataset used, and if I change it in some way, what are the implications downstream?

Neither question is easy to answer fully and accurately in real world data preparation pipelines. For example, if a data scientist writes a Python function to in-memory process several input datasets and outputs a single dataset, how can I be sure from what information each cell of the new dataset was derived?

Process Governance

Process governance focuses on formalizing the steps in the MLOps process and associating actions with those. Typically these actions are reviews, sign-offs, and the capture of supporting materials such as documentation. The aim is twofold:

1. To ensure every governance-related consideration is made at the correct time, and correctly acted upon. For example, models should not be deployed to production until all validation checks have been passed.

2. Enable oversight from outside of the strict MLOps process. Auditors, risk managers, compliance officers, and the business as a whole all have an interest in being able to track progress and review decisions at a later stage.

Effective implementation of process governance is hard:

- Formal processes for the ML lifecycle are rarely easy to define accurately. The understanding of the complete process is usually spread across the many teams involved, often with no one person having a detailed understanding of it as a whole.
- For the process to be applied successfully, every team must be willing to adopt it wholeheartedly.
- If the process is just too heavy-weight for some use-case, teams will certainly subvert it, and much of the benefit will be lost.

Today, process governance is most commonly found in organizations with a traditionally heavy burden of regulation and compliance, such as finance. Outside of these, it is rare. With ML creeping into all spheres of commercial activity, and with rising concern about responsible AI, we will need new and innovative solutions to this problem that can work for all businesses.

Closing Thoughts

Given this overview of features required for and processes affected by MLOps, it's clearly not something data teams—or even the data-driven organization at large—can ignore. Nor is it an item to check off of a list ("yes, we do MLOps!"), but rather a complex interplay between technologies, processes, and people that requires discipline and time to do correctly.

The following chapters will go even deeper into each one of the ML model lifecycle components at play in MLOps, providing a look more at how each should be done to get closer to the ideal MLOps implementation.

CHAPTER 4

Developing 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 fourth 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 *mlops@dataiku.com*.

Anyone who wants to be serious about MLOps needs to have at least a cursory understanding of the model development process. Depending on the situation, this process can range from quite simple to extremely complex, and it dictates the constraints of subsequent usage, monitoring, and maintenance of models.

The implications of the data collection process on the rest of the model's life is quite straightforward, and one easily sees how a model can become stale. For other parts of the model, the effects may be less obvious.

For example, take feature creation, where feeding a date to the model versus a flag indicating whether the day is a public holiday may make a big difference in performance but also come with significantly different constraints on updating the model. Or consider how the metrics used for evaluating and comparing models may enable automatic switching to the best possible version down the line, should the situation require it.

This chapter will therefore briefly cover the basics of model development, specifically in the context of MLOps — that is, how models might be built and developed in ways that make MLOps considerations easier to implement down the line.

What is a Machine Learning Model?

Machine learning models are leveraged both in academia and in the real world (i.e., business contexts), so it's important to distinguish what they represent in theory vs. how they are implemented in practice. This section will dive into both.

In Theory

A machine learning model is a projection of reality; that is, a partial and approximate representation of some aspect (or aspects) of a real thing or process. Which aspect(s) often depends on what is available and useful. A machine learning model, once trained, boils down a mathematical formula that yields a result when fed some inputs — say, a probability estimation of some event happening or the estimated value of a raw number.

Machine learning models are based on statistical theory, and machine learning algorithms are the tools that build models from training data. Their goal is to find a synthetic representation of the data they are fed with, and this data represents the world as it was at the time of collection. Therefore, machine learning models can be used to make predictions when the future looks like the past, because their synthetic representation is still valid.

Generalization Capacity

Machine learning models' ability to accurately predict for cases that are not exactly like the input data is called their generalization capacity. Even when they yield outputs like horses with zebra stripes that do not exist in training datasets, they do it by modeling a probability distribution that allows them to have this kind of surprising generalization capacity.

An often-used example for how machine learning models can predict and generalize is the price of a house. Of course, the selling price of a house will depend on too many factors too complex to model precisely, but getting close enough to be useful is not so difficult. The input data for that model may be things inherent to the house like surface area, number of bedrooms and bathrooms, year of construction, location, etc., but also other more contextual information like the state of the housing market at the time of sale, whether the seller is in a hurry, and so on. With complete enough historical data — and provided the market conditions do not change too much — an algorithm can compute a formula that provides a reasonable estimate.

Another frequent example is a health diagnosis or prediction that someone will develop a certain disease within a given timeframe. This kind of classification model often outputs the probability of some event, sometimes also with a confidence interval.

In Practice

In practice, a model is the set of parameters necessary to rebuild and apply the formula. It is usually stateless and deterministic (i.e., the same inputs always give the same outputs, with some exceptions — see Sidebar: Online Learning Models).

This includes the parameters of the end formula itself, but it also includes all the transformations to go from the input data that will be fed to the model to the end formula that will yield a value plus the possible derived data (like a classification or a decision). Given this description in practice, it usually does not make a difference whether the model is ML-based or not: it is just a computable mathematical function applied to the input data, one row at a time.

In the house price case for instance, it may not be practical to gather enough pricing data for each and every zip code to get a model that's accurate enough in all target locations. Instead, maybe the zip codes will be replaced with some derived inputs that are deemed to have the most influence on price — say, average income, population density, or proximity to some amenities. But since end users will continue to input the zip code and not these derived inputs, for all intents and purposes, all of this transformation is also part of the pricing model.

Outputs can also be richer than a single number. A system that detects fraud, for example, will often provide some kind of probability rather than a binary answer (and in some cases maybe also a confidence interval). Depending on the acceptability of fraud and the cost of subsequent verification or denial of transaction, it may be set up to only classify fraudulent instances where the probability reaches some fine-tuned threshold. Some models even include recommendations or decisions, such as which product to show a visitor that maximizes spend or which treatment provides the most probable recovery.

All of these transformations and associated data are part of the model to some degree; however, this does not mean they are always bundled in a monolithic package, as it could quickly get unwieldy, and some part of this information comes with varying constraints (different refresh rates, external sources, etc.).

Required Components

Building a machine learning model requires many components, outlined in Table 4-1.

Training data	Training data is usually labeled for the prediction case with examples of what is being modeled (supervised learning). It might sound obvious, but it's important to have good training data — an illustrative example of when this was not the case was data from damaged planes during World War II, which suffered from survivor bias and therefore was not good training data.

A performance metric	A performance metric is what the model being developed seeks to optimize. It should be chosen carefully to avoid unintended consequences, like the cobra effect (a famous anecdote where a reward for dead cobras led some to breed them). For example, if 95% of the data has class A, optimizing for raw accuracy may produce a model that always predicts A and is 95% accurate.
ML algorithm	There are a variety of models that work in various ways and have different pros and cons. It is important to note that some algorithms are more suited to certain tasks than others, but their selection also depends on what needs to be prioritized: performance, stability, interpretability, computation cost, etc.
Hyperparameters	Hyperparameters are configurations for ML algorithms. The algorithm contains the basic formula, the parameters it learns are the operations and operands that make up this formula for that particular prediction task, and the hyperparameters are the ways that the algorithm may go to find these parameters. For example, in a decision tree (where data continues to be split in two according to what looks to be the best predictor in the subset that reached this path), one hyperparameter is the depth of the tree (i.e., the number of splits).
Evaluation dataset	When using labeled data, an evaluation data set that is different from the training set will be required to evaluate how the model performs on unseen data (i.e., how well it can generalize).

Different ML Algorithms, Different MLOps Challenges

What ML algorithms all have in common is that they model patterns in past data to make predictions, and the quality and relevance of this past experience is the key factor in their effectiveness. Where they differ is that each style of model has specific characteristics and presents different challenges in MLOps (outlined in Table 4-3).

Type	Name	MLOps Considerations
Linear	Linear Regression	There is a tendency for overfitting
	Logistic Regression	
Tree-Based	Decision Tree	Can be unstable — small changes in data can lead to a large change in the structure of the optimal decision tree.
	Random Forest	Predictions can be difficult to understand, which is challenging from a Responsible AI perspective. Random Forest models can also be relatively slow to output predictions, which can present challenges for applications.
	Gradient Boosting	Like Random Forest, predictions can be difficult to understand. Also, a small change in the feature or training set can create radical changes in the model.
Deep Learning	Neural Networks	In terms of interpretability, deep learning models are almost impossible to understand. Deep learning algorithms, including neural networks, are also extremely slow to train and require a lot of power (and data). Is it worth the resources, or would a simpler model work just as well?

Some ML algorithms can best support specific use cases, but governance considerations may also play a part in the choice of algorithm. In particular, highly regulated environments where decisions must be explained (e.g., financial services) cannot use opaque algorithms like neural networks, and have to favor simpler techniques, such as decision trees. In many use cases, it's not so much a tradeoff on performance but rather a tradeoff on cost. That is, simpler techniques usually require more costly manual feature engineering to reach the same level of performance as more complex techniques.

Computing Power

When talking about components of machine learning model development, it's impossible to ignore computing power. Some say planes fly thanks to human ingenuity, but also thanks to a lot of fuel. This holds true with machine learning as well: its development is inversely proportional to the cost of computing power.

From hand-computed linear regression of the early 20th century to today's largest deep learning models, new algorithms arose when the required computing power became available. For example, mainstream algorithms like random forest and gradient boosting both require a computing power that was expensive 20 years ago.

In exchange, they brought an ease-of-use that considerably lowered the cost of developing ML models, thus making viable new use cases. The decrease of the cost of data also helped, but it was not the first driver: very few algorithms leverage big data technology in which both data and computation are distributed over a large number of computers — most of them still operate with all the training data in memory.

Data Exploration

Let's assume that the data sources to address a business problem have been identified (See **Chapter 8: Model Governance** and Data Sources and Exploratory Data Analysis in chapter 3 for details). When data scientists or analysts consider these sources to train a model, they need to first get a grasp of what that data looks like — even a model trained using the most effective algorithm can only be as good as its training data. At this stage, a number of issues can prevent all or part of the data from being useful, including incompleteness, inaccuracy, inconsistency, etc.

An example of such a process can include:

- Documenting how the data was collected and what assumptions were already made.
- Looking at summarizing statistics of the data: what is the domain of each column? Are there some rows with missing values? Obvious mistakes? Strange outliers? No outliers at all?
- Taking a closer look at the distribution of the data.
- Cleaning, filling, reshaping, filtering, clipping, sampling, etc.

- Checking correlations between the different columns, running statistical tests on some subpopulations, fit distribution curves.
- Comparing that data to other data or models in the literature: is there some usual information that seems to be missing? Is this data comparably distributed?

Of course, domain knowledge is required to make informed decisions during this exploration. Some oddities may be hard to spot without specific insight, and assumptions made can have consequences that are not obvious to the untrained eye.

Feature Engineering and Selection

Features are how data is presented to a model, serving to inform that model on things it may not infer by itself (see Table 4-1). For instance, in trying to estimate the potential duration of a business process given the current backlog, if one of the inputs is a date, it is pretty common to derive the corresponding day of the week or how far ahead the next public holiday is from that date. If the business serves multiple locations that observe different business calendars, that information may also be important.

Derivatives	Infer new information from existing information — e.g., what day of the week is this date?
Enrichment	Add new external information — e.g., is this day a public holiday?
Encoding	Present the same information differently — e.g., day of the week or weekday vs weekend.
Combination	Link features together — e.g., the size of the backlog might need to be weighted by the complexity of the different items in it.

Another example, to follow up on the house pricing scenario from the previous section, would be using average income and population density, which hopefully allows the model to better generalize and train on more diverse data than trying to segment by area (i.e., zip code).

Feature Engineering Techniques

A whole market exists for such complementary data that extends far beyond the open data that public institutions and companies share. Some services provide direct enrichment that can save a lot of time and effort.

There are, however, many cases when information data scientists need for their models is not available. In this case there are techniques like impact coding whereby data scientists replace a modality by the average value of the target for this modality, thus allowing the model to benefit from data in the similar range (at the cost of some information loss).

Ultimately, most ML algorithms require a table of numbers as input, each row representing a sample, and all samples coming from the same dataset. When the input data is not tabular, data scientists can use other tricks to transform it.

The most common one is one-hot encoding. For example, a feature that can take three values (e.g., Raspberry, Blueberry, and Strawberry) is transformed into three features that can take only two values — yes or no (e.g., Raspberry yes/no, Blueberry yes/no, Strawberry yes/no).

Text or image inputs, on the other hand, require more complex engineering. Deep learning has recently revolutionized this field by providing models that transform images and text into tables of numbers that are usable by ML algorithms. These tables are called embeddings, and they allow data scientists to perform transfer learning because they can be used in domains on which they were not trained.

For example, even if a particular deep learning model was trained on images that did not contain any forks, it may give a useful embedding to be used by a model that is trained to detect them because a fork is an object, and that model was trained to detect similar human-made objects.

How Feature Selection Impacts MLOps Strategy

When it comes to feature creation and selection, the question of how much and when to stop comes up regularly. Adding more features may produce a more accurate model, achieve more fairness when splitting into more precise groups, or compensate for some other useful missing information. However, it also comes with downsides, all of which can have a significant impact on MLOps strategies down the line:

- The model can become more and more expensive to compute
- More features requires more inputs and more maintenance down the line
- More features means a loss of some stability
- The sheer number of features can raise privacy concerns

Automated feature selection can help by using heuristics to estimate how critical some features will be for the predictive performance of the model. For instance, one can look at the correlation with the target variable or quickly train a fast and simple model on a representative subset of the data then look at which features are the strongest predictors.

Which inputs to use, how to encode them, how they interact or interfere with each other — such decisions require a certain understanding of the inner workings of the ML algorithm. The good news is that some of this can be partly automated, e.g., by using tools such as auto-sklearn or AutoML applications that cross-reference features against a given target to estimate which features, derivatives, or combinations are likely to yield the best results, leaving out all the features that would probably not make that much of a difference.

Other choices still require human intervention, such as deciding whether to try and collect additional information that might improve the model. Spending time to build business-friendly features will often improve the final performance and ease the adoption by end users, as model explanations are likely to be simpler. It can also reduce modeling debt, allowing data scientists to understand the main prediction drivers and ensure that they are robust. Of course, there are trade-offs to consider between the cost of time spent to understand the model and the expected value, as well as risks associated with the model's use.

Feature Stores

Feature factories, or feature stores, are repositories of different features associated with business entities that are created and stored in a central location for easier reuse. They usually combine an offline part (slower, but potentially more powerful) and an online part (quicker and more useful for real-time needs), making sure both of them remain consistent with each other.

Given how time consuming feature engineering is for data scientists, this concept has huge potential to free up their time for even more valuable tasks. Machine learning is still often the "The High-Interest Credit Card of Technical Debt." Reversing this will require huge efficiency gains in the data-to-model-to-production and in the MLOps process, and feature stores is one way to get there.

Experimentation

Experimentation takes place throughout the entire model development process, and usually every important decision or assumption comes with at least some experiment or previous research to justify those decisions. Experimentation can take many shapes, from building full-fledged predictive ML models to doing statistical tests or charting data.

The goal of experimentation is multifold and includes:

- Assessing how useful or how good of a model can be built given the elements outlined in Figure 4-1 (the next section will cover model evaluation and comparison in more detail).
- FInding the best modeling parameters (algorithms, hyperparameters, feature preprocessing, etc.).
- Tuning the bias/variance tradeoff (see Bias & Variance Sidebar) for a given training cost to fit that definition of "best."

- Finding a balance between model improvement and improved computation costs (since there's always room for improvement, how good is good enough?)

Bias & Variance

A high bias model (also known as underfitting) fails to discover some of the rules that could have been learned from the training data, possibly because of reductive assumptions making the model overly simplistic.

A high variance model (or overfitting) sees patterns in noise and seeks to predict every single variation, resulting in a complex model that does not generalize well beyond its training data.

When experimenting, data scientists need to be able to quickly iterate through all the possibilities for each of the model building blocks outlined in Figure 4-1. Fortunately, there are tools to do all of this semi-automatically, where you only need to define what should be tested (the space of possibilities) depending on prior knowledge (what makes sense) and the constraints (e.g., computation, budget, etc.), and the different combinations are tried out.

Some tools allow for even more automation, for instance by offering stratified model training. For example, say the business wants to predict customer demand for products to optimize inventory, but behavior varies a lot from one store to the next. Stratified modeling consists of training one model per store that can be better optimized for each store rather than a model that tries to predict in all stores.

How Experimentation Impacts MLOps Strategy

Trying all combinations of every possible hyperparameter, feature handling, etc., quickly becomes untraceable. Therefore, it is useful to define a time and/or computation budget for experiments as well as an acceptability threshold for usefulness of the model (more on that notion in the next section).

Notably, all or part of this process may need to be repeated every time anything in the situation changes (including whenever the data and/or problem constraints change significantly — see Chapter 7, specifically the section on model drift). Ultimately this means that all experiments that informed the final decisions the data scientist made to arrive at the model as well as all the assumptions and conclusions along the way may need to be re-run and reexamined.

Fortunately, more and more data-science and machine learning platforms (like Dataiku) allow for automating these workflows not only on the first run but also to preserve all the processing operations for repeatability. Some also allow for the use of version control and experimental branch spin-off to test out theories, then merge, discard, or keep them (see upcoming section on Version Management & Reproducibility).

Evaluating and Comparing Models

”Essentially, all models are wrong, but some are useful” – George E.P. Box (20th century British statistician)

A model should not aim to be perfect, but it should pass the bar of “good enough to be useful” while keeping an eye on the uncanny valley — typically a model that *looks* like it’s doing a good job but does a bad (or catastrophic) job for a specific subset of cases (say, an underrepresented population).

With this in mind, it’s important to evaluate a model in context and have some ability to compare to what existed before the model — whether a previous model or rules-based process — to get an idea of the outcome of replacing the current model or decision process by the new one.

A model with an absolute performance that could technically be considered disappointing can still possibly enhance an existing situation. For instance, having a slightly more accurate forecast of demand for a certain product or service may have huge cost-saving impacts.

On the contrary, a model that gets a perfect score is suspicious, as most problems have noise in the data that’s at least somewhat hard to predict. A perfect or nearly-perfect score may be a sign that there is a leak in the data (i.e., that the target to be predicted is also in the input data or that an input feature is very correlated to the target but, practically, available only once the target is known), or that the model overfits the training data and will not generalize well.

Choosing Evaluation Metrics

Choosing the proper metric by which to evaluate and compare different models for a given problem can lead to very different models (think of the cobra effect mentioned in Figure 4-1). A simple example: accuracy is often used for automated classification problems but is rarely the best fit when the classes are unbalanced (i.e., when one of the outcomes is very unlikely compared to the other). In a binary classification problem where the positive class (i.e., the one that is interesting to predict because its prediction will trigger an action) is rare, say 5% of occurrences, a model that constantly predicts the negative class is therefore 95% accurate, while also utterly useless.

Unfortunately there is no one-size-fits-all metric — you need to pick one that matches the problem at hand, which means understanding the limits and tradeoffs of the metric (the mathematics side) and their impact on the optimization of the model it will bring (the business side).

To get an idea of how well a model will generalize, that metric should be evaluated on a part of the data that was not used for the model's training (a *holdout* dataset), a method called cross-testing. There can be multiple steps where some data is held for evaluation and the rest is used for training or optimizing, such as metric evaluation or hyperparameter optimization. There are different strategies as well, not necessarily just a simple split. In k-fold, for example, data scientists rotate the parts that they hold out to evaluate and train multiple times. This multiplies the training time but gives an idea of the stability of the metric.

With a simple split, the holdout dataset can consist of the most recent records instead of randomly chosen one. Indeed, as models are usually used for future predictions, it is likely that assessing them as if they were used for prediction on the most recent data leads to the more realistic estimations. In addition, it allows to assess whether the data drifted between the training and the holdout dataset (See Drift Detection in Practice in Chapter 7 for details).

As an example, Figure 4-1 shows a scheme in which a test dataset is a holdout (in grey) in order to perform the evaluation. The remaining is split into three parts in order to find the best hyperparameter combination by training it three times with a given combination on each of the blue datasets, validating its performance on their respective green datasets. The grey dataset is used only once with the best hyperparameter combination, while the other ones are used with all of them.

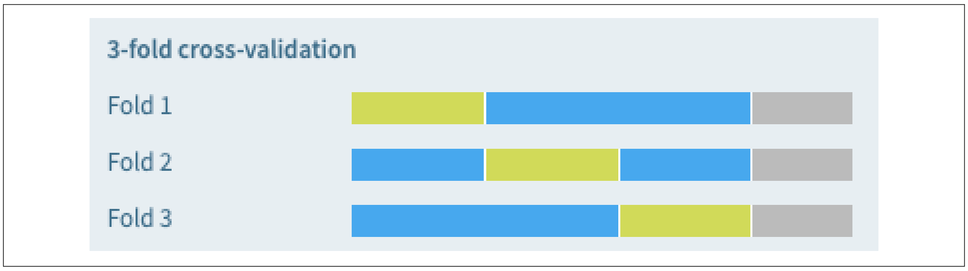


Figure 4-1.

Oftentimes, data scientists may want to periodically retrain models with the same algorithms, hyperparameters, features, etc., but on more recent data. Naturally, the next step is to compare the two models and see how the new version fares. But it's also important to make sure all previous assumptions still hold: that the problem hasn't fundamentally shifted, that the modeling choices made still fit the data, etc. This is more specifically part of performance and drift monitoring (find more details on this in Chapter 7).

Cross-Checking Model Behavior

Beyond the raw metrics, when evaluating a model, it's critical to understand how it will behave. Depending on the impact of the model's predictions, decisions, or classifications, a more or less deep understanding may be required. For example, data scientists should take reasonable steps (with respect to that impact) to ensure that the model is not actively harmful — a model that would predict that *all* patients need to be checked by a doctor may score high in terms of raw prevention, but not so much on realistic resource allocation.

Examples of these reasonable steps include:

- Cross-checking different metrics (and not only the ones initially optimized for)
- Checking how the model reacts to different inputs — e.g., plot the average prediction (or probability for classification models) for different values of some inputs and see whether there are oddities or extreme variability.
- Splitting one particular dimension and checking the difference in behavior and metrics across different subpopulations — e.g., is the error rate the same for males and females?

These kinds of global analyses should not be understood as causality, just as correlation — that is, they do not necessarily imply a specific causal relationship between some variables and an outcome; they merely show how the *model* sees that relationship. In other words, the model should only be used with care for what-if analysis — if one feature value is changed, the model prediction is likely to be wrong if the new feature value has never been seen in the training dataset or if it has never been seen in combination with the values of the other features in this dataset.

When comparing models, those different aspects should be accessible to data scientists, who need to be able to go deeper than a single metric. That means the full environment (interactive tooling, data, etc.) needs to be available for all models, ideally allowing for comparison from all angles and between all components (e.g., for drift — same settings but different data, or for modeling performance — same data but different settings, etc.).

Impact of Responsible AI on Modeling

Depending on the situation (and sometimes depending on the industry or sector of the business), on top of a general understanding of model behavior, data scientists may also need models' individual predictions to be explainable, including having an idea of what specific features pushed the prediction one way or the other. Sometimes it may be very different for a specific record than on average. Popular methods to compute them include Shapley value and individual conditional expectation computations.

For example, the measured level of a specific hormone could generally push a model to predict someone has a health issue, but for a pregnant woman, that level makes the model infer she is at no such risk. Some legal frameworks mandate some kind of explainability for decisions made by a model that have consequences on humans, like recommending a loan to be denied. Chapter 8, specifically Key Elements of Responsible AI — Element 2, Bias, will discuss this topic in detail.

Note that the notion of explainability has several dimensions. In particular, deep learning networks are sometimes called "black box" models because of their complexity. Mostly improperly because when you can read the model coefficients, a model is fully specified and it is usually a conceptually remarkably simple formula, but a very large formula that becomes impossible to intuitively apprehend. Conversely, global and local explanation tools such as partial dependence plots or Shapley value computations give some insights but arguably do not make the model intuitive, to actually communicate a rigorous and intuitive understanding of what exactly the model is doing, limiting the model complexity is required.

Fairness requirements can also have dimensioning constraints on the model development. Consider a theoretical example to better understand what is at stake when it comes to bias: a US-based organization regularly hires people who do the same types of jobs. Data scientists could train a model to predict the workers' performance according to various characteristics, and people would then be hired based on the probability that they would be high-performing workers.

Though this seems like a simple problem, unfortunately, it's fraught with pitfalls. To make this problem completely hypothetical and to detach it from the complexities and problems of the real world, let's say everyone in the working population belongs to one of two groups: Weequay or Togruta.

For this hypothetical example, let's claim that a far larger population of Weequay attend university. Off the bat, there would be an initial bias in favor of Weequay (amplified by the fact they would have been able to develop their skills through years of experience).

As a result, there would not only be more Weequay than Togruta in the pool of applicants, but Weequay applicants would tend to be more qualified. The employer has to hire 10 people during the month to come. What should it do?

- As an equal opportunity employer, it should ensure the fairness of its recruitment process as it controls it. That means in mathematical terms, for each applicant and all things being equal, being hired (or not) should not depend on their group (in this case, Weequay or Togruta). However, this results in bias in and of itself, as Weequay are more qualified. Note that "all things being equal" can be interpreted in various ways, but the usual interpretation is that the organization is likely not considered accountable for processes it does not control.

- The employer may also have to avoid disparate impact; that is, practices in employment that adversely affect one group of people of a protected characteristic more than another. Disparate impact is assessed on sub-populations and not on individuals — practically, it assesses whether proportionally speaking, the company has hired as many Weequay as Togruta. Once again, the target proportions may be those of the applicants or those of the general population, though the former is more likely, as the organization can't be accountable for biases in processes out of its control.

The two objectives are mutually exclusive. In this scenario, equal opportunity would lead to hiring 60% (or more) Weequay and 40% (or fewer) Togruta. As a result, the process has a disparate impact on the two populations because the hiring rates are different.

Conversely, if the process is corrected so that 40% of people hired are Togruta to avoid disparate impact, it means that some rejected Weequay applicants will have been predicted as more qualified than some accepted Togruta applicants (contradicting the equal opportunity assertion).

There needs to be a tradeoff — the law sometimes refers to the 80% rule. In this example, it would mean that the hiring rate of Togruta should be equal to or larger than 80% of the hiring rate of Weequay. In this example, it means that it would be OK to hire up to 65% Weequay.

The point here is to say that clearly, defining these objectives cannot be a decision made by data scientists alone. But even once the objectives are defined, the implementation itself may also be problematic:

- Without any indications, data scientists naturally try to build equal opportunity models because they correspond to models of the world as it is. Most of the tools data scientists employ also try to achieve this because this is the most mathematically sound option, yet some ways to achieve this goal may be unlawful. For example, the data scientist may choose to implement two independent models: one for Weequay and one for Togruta. It could be a reasonable way to address the biases induced by a training dataset in which Weequay is overrepresented, but it would induce a disparate treatment of men and women that could be considered discriminatory.
- In order to let data scientists use their tools in the way they were designed (i.e., to model the world as it is), they may decide to post-process the predictions so that they fit with the organization's vision of the world as it should be. The simplest way of doing it is to choose a higher threshold for Weequay than for Togruta. The gap between both will adjust the tradeoff between "equal opportunity" and "equal

impact;" however, it may still be considered discriminatory because of the disparate treatment.

Data scientists are unlikely to be able to sort this problem out alone (See Key Elements of Responsible AI Chapter 8 for a broader view on the subject). This simple example illustrates the complexity of the subject and the fact that bias is as much a business question as a technical question, and it may be even more complex given that there may be many protected attributes.

Consequently, the solution heavily depends on the context. For instance, this example of Weequay and Togruta is representative of processes that give access to privileges. The situation is different if the process has negative impacts on the user (like fraud prediction that leads to transaction rejection) or neutral (like disease prediction).

Solutions also depend heavily on the applicable law. For example, France forbids the collection of information on race or ethnicity, so fairness assessment can be very hard, though risks of disparate treatment are lower.

Version Management & Reproducibility

Version management and reproducibility addresses two different needs:

1. During the experimentation phase, data scientists may find themselves going back and forth on different decisions, trying out different combinations and reverting when they don't produce the desired results. That means having the ability to go back to different "branches" of the experiments — for example, restoring a previous state of a project when the experimentation process led to a dead end.
2. Data scientists or others (auditers, managers, etc.) may need to be able to replay the computations that led to model deployment for an audit team several years after the experimentation was first done.

This particular problem has arguably been somewhat solved when everything is code-based, with source version control technology. Modern data processing platforms typically offer similar capabilities for data transformation pipelines, model configuration, etc. Merging several parts is, of course, less straightforward than merging code that diverged, but the basic need is to be able to go back to some specific experiment, if only to be able to copy its settings to replicate them in another branch.

Another very important property of a model is reproducibility. After a lot of experiments and tweaking, data scientists may arrive at a model that fits the bill. But after that, operationalization necessitates model reproduction not only in another environment, but also possibly from a different starting point. Repeatability also makes debugging much easier (sometimes even simply possible). To this end, all facets of the model need to be documented and reusable, including:

Assumptions
: When a data scientist makes decisions and assumptions about the problem at hand, its scope, the data, etc., they should all be explicit and logged so that they can be checked against any new information down the line.

Randomness
: A lot of ML algorithms and processes — such as sampling — make use of pseudo-random numbers. To be able to precisely reproduce an experiment (e.g., for debugging) means to have control over that pseudo-randomness, most often by controlling the "seed" of the generator (i.e., the same generator initialized with the same seed would yield the same sequence of pseudo-random numbers).

Data
: To get repeatability, the same data must be available. This can sometimes be tricky because the storage capacity required to version data can be prohibitive depending on the rate of update and quantity. Also, branching on data does not yet have as rich an ecosystem of tools as branching on code.

Settings
: This one is a given: all processing that has been done must be reproducible with the same settings.

Results
: While developers use merging tools to compare and merge different text file versions, data scientists need to be able to compare in-depth analysis of models (from confusion matrices to partial dependencies plots) in order to obtain models that satisfy the requirements.

Implementation
: Ever-so-slightly different implementations of the same model can actually yield different models — enough to change the predictions on some close calls. And the more sophisticated the model, the higher the chances that these discrepancies happen. On the other hand, scoring a dataset in bulk with a model comes with different constraints than scoring a single record live in an API, so different implementations may sometimes be warranted for the same model. But when debugging and comparing, data scientists need to keep the possible differences in mind.

Environment
: Given all the steps covered in this chapter, it's clear that a model is not just its algorithm and parameters. From the data preparation to the scoring implementation, including feature selection, feature encoding, enrichment, etc., The envi-

ronment in which several of those steps run may be more or less implicitly tied to the results. For instance, a slightly different version of a Python package involved in one step may change the results in ways that can be hard to predict. Preferably, data scientists should make sure that the runtime environment is also repeatable. Given the pace at which ML is evolving, this might require techniques that freeze the computation environments.

Part of the underlying documentation task can be automated. The use of an integrated platform for design and deployment can greatly decrease the reproducibility costs by ensuring structured information transfer.

Closing Thoughts

Model development is one of the most critical and consequential steps of MLOps. The many technical questions that are necessarily answered one way or another during this phase have big repercussions on all aspects of the MLOps process throughout the life of the models. Therefore, exposure, transparency, and collaboration are crucial to long-term success.

The model development stage is also the one that has also been practiced the most, and in the pre-MLOps world, often represents the whole effort that is produced, yielding a model that will then get used as-is (with all its consequences and limitations).

CHAPTER 5

Monitoring and Feedback Loop

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 seventh 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 *mlops@dataiku.com*.

When a machine learning model is deployed in production, it can start degrading in quality fast—and without warning—until it's too late (i.e., it's had a potentially negative impact on the business). That's why model monitoring is a crucial step in the ML model lifecycle and a critical piece of MLOps.

Machine learning models need to be monitored at two levels:

- At the resource level, including ensuring the model is running correctly in the production environment. Key questions include: Is the system alive? Is the CPU, RAM, network usage, and disk space as expected? Are requests being processed at the expected rate?
- At the performance level, meaning monitoring the pertinence of the model over time. Key questions include: Is the model still an accurate representation of the pattern of new incoming data, and is it still performing as well as during its design phase?

The first level is a traditional DevOps topic that has been extensively addressed in the literature (and, indeed, has been covered already in Chapter 6, **Deployment to Pro-**

duction). However, the latter is more complicated. Why? Because how well a model performs is a reflection of the data used to train it; in particular, how representative that training data is of the live request data. As the world is constantly changing, a static model cannot catch up with new patterns that are emerging and evolving without a constant source of new data.

Model performance monitoring attempts to track this degradation, and at an appropriate time, it will also trigger the retraining of the model with more representative data. This chapter will delve into detail on how data teams should handle both monitoring as well as subsequent retraining.

How Often Should Models Be Retrained?

One of the key initial questions teams have around monitoring and retraining is: How often should models be retrained? Unfortunately, there is no easy answer, as this question depends on many factors, including:

- The domain: Models in areas like cybersecurity or real-time trading need to be updated regularly to keep up with the constant changes inherent in these fields. On the other hand, physical models—like voice recognition—are generally more stable, as the patterns don't often abruptly change over time. However, even more stable physical models need to adapt to change; what happens to a voice recognition model if the person has a cough and the tone of his voice changes?
- The cost: Organizations need to consider whether the cost of retraining is worth the improvement in performance. For example, if it takes one week to run the whole data pipeline and retrain the model, is it worth the 1% improvement?
- The model performance: In some situations, the model performance is restrained by the limited number of training examples, and thus the decision to retrain hinges on collecting enough new data.

Whatever the domain, the delay to obtain the ground truth (more on ground truth in the sections to come) is key to defining a lower bound to the retraining period. It is very risky to use a prediction model when there is a risk that it drifts faster than the lag between prediction time and ground truth obtention time. In this scenario, the model can start giving bad results without any recourse measure other than to withdraw the model if the drift is too significant. What this means in practice is that it is unlikely a model with a lag of one year is retrained more than a few times a year.

For the same reason, it is unlikely that a model is trained on data collected during a period smaller than this lag. Retraining will not be performed in a shorter period, either. In other words, if the model retraining occurs way more often than the lag, there will be almost no impact of the retraining on the performance of the model.

There are also two organizational bounds to consider when it comes to retraining frequency:

- An upper bound: It is better to perform retraining once every year to ensure that the team in charge has the skills to do it (despite potential turnover—i.e., the possibility that the people retraining the model were not the ones who built it) and to ensure that the computing toolchain is still up.
- A lower bound: Take, for example, a model with near-instantaneous feedback (like a recommendation engine where the user clicks on the product offerings within seconds after the prediction). Advanced deployment schemes will involve shadow testing or A/B testing to make sure that the model performs as well as the model designer anticipated. As it is a statistical validation, it takes some time to gather the required information—this necessarily sets a lower bound to the retraining period. Even with a simple deployment, the process will probably allow for some human validation or for the possibility of manual rollback, which means it's unlikely that the retraining will occur more than once a day.

Therefore, it is very likely that retraining will be done between once a day and once a year. As a result, the simplest solution that consists of retraining the model in the same way and in the same environment it was trained in for the first time is acceptable. Some critical cases may require the deployment of the retraining in a production environment even though the initial training was done in a design environment, but the retraining method is most of the time identical to the training one so that the overall complexity is limited. As always, there is an exception to this rule: online learning.

Online Learning

Sometimes, the use case requires teams to go further than the automation of the existing manual ML pipeline by using dedicated algorithms that can train themselves iteratively (standard algorithms, by contrast, are retrained from scratch most of the time, with the exception of deep learning algorithms).

While conceptually attractive, these algorithms are more costly to set up. The designer has to not only test the performance of the model on a test dataset but also to qualify its behavior when data changes (the latter because it's difficult to mitigate bad learning once the algorithm is deployed and because it is hard to reproduce the behavior when each training recursively relies on the previous one—i.e., one needs to replay all the steps to understand the bad behavior).

There is no standard way—similar to cross-validation—to do this process, so the design costs will be higher. Online machine learning is a vivid branch of research with some mature technologies like state-space models, though they require significant skills to be used effectively. Online learning is typically appealing in streaming use cases, though mini-batches may be more than enough to handle it.

In any case, some level of model retraining is definitely necessary—it's not a question of if, but when. Deploying ML models without considering retraining would be like launching an unmanned aircraft from Paris in the exact right direction and hoping it will land safely in New York City without further control.

The good news is that if it was possible to gather enough data to train the model for the first time, then most of the solutions for retraining are already available (with the possible exception of cross-trained models that are used in a different context—for example, trained with data from one country but used in another). It is therefore critical for organizations to have a clear idea of deployed models' drift and accuracy by setting up a process that allows for easy monitoring and notifications. For example, an ideal scenario would be a pipeline that automatically triggers checks for the degradation of model performance.

It's important to note that the goal of notifications is not necessarily to kick off an automated process of retraining, validation, and deployment. Model performance can change for a variety of reasons, and retraining may not always be the answer. The point is to alert the data scientist of the change, who can then diagnose the issue and evaluate the next course of action.

It is, therefore, critical that as a part of MLOps and the ML model lifecycle that data scientists as well as their managers and the organization as a whole (who are ultimately the ones that have to deal with the business consequences of degrading model performances and any subsequent changes) understand model degradation. Practically, every deployed model should come with monitoring metrics and corresponding warning thresholds to detect meaningful business performance drops as quickly as possible. The following sections will focus on understanding these metrics in order to be able to define them for a particular model.

Understanding Model Degradation

Once a machine learning model is trained and deployed in production, there are two approaches to monitor its performance degradation: ground truth evaluation or input drift detection. Understanding the theory behind and limitations of these approaches is critical to determining the best strategy.

Ground Truth Evaluation

The approach of ground truth retraining is to wait for the label event. For example, in a fraud detection model, the ground truth would be whether or not a specific transaction was actually fraudulent. For a recommendation engine, it would be whether or not the customer clicked on—or ultimately bought—one of the recommended products.

With the new ground truth collected, the next step is to compute the performance of the model based on ground truth and compare it with registered metrics in the training phase. When the difference surpasses a threshold, the model can be deemed as outdated, and it should then be retrained.

The metrics to be monitored can be of two varieties:

- Statistical metrics like accuracy, ROC AUC, log loss, etc.. As the model designer has probably already chosen one of these metrics to pick the best model, it is a first-choice candidate for monitoring. For more complex models where the average performance is not enough, it may be necessary to look at metrics computed by subpopulations.
- Business metrics, like cost/benefit assessment. For example, the credit scoring business has developed its own specific metrics.

The main advantage of the first kind of metric is that they are domain agnostic, so the data scientist likely feels comfortable setting thresholds. So as to have the earliest meaningful warning, it is even possible to compute p-values to assess the probability that the observed drop is not due to random fluctuations.

The drawback is that the drop may be statistically significant without having any noticeable impact. Or worse, the cost of retraining and the risk associated with a redeployment may be higher than the expected benefits. Business metrics are far more interesting because they ordinarily have a monetary value—subject matter experts can better handle the cost / benefit tradeoff of the retraining decision.

When available, ground truth monitoring is the best solution. However, it may be problematic. There are three main challenges:

- Ground truth is not always immediately—or even imminently—available: For some types of models, teams need to wait months (or longer) for ground truth labels to be available, which can mean significant economic loss if the model is degrading quickly. As said before, deploying a model for which the drift is faster than the lag is risky. However, by definition, drifts are not forecastable, so models with long lags need mitigation measures.
- Decoupling of ground truth and prediction: To compute the performance of the deployed model on new data, it's necessary to be able to match ground truth with the corresponding observation. In many production environments, this is a challenging task because these two pieces of information are generated and stored in different systems and at different timestamps. In other words, for low-cost or short-lived models, it might not be worth automated ground truth collection.

Note that it is, in general, rather short-sighted, as sooner or later, the model will need to be retrained.

- Partially available ground truth: In some situations, it is extremely expensive to retrieve the ground truth for all the observations, which means choosing which samples to label and thus inadvertently introducing bias into the system.

For the last challenge, fraud detection is one such use case where it is easy to illustrate. Given that each transaction needs to be examined manually and the process takes a long time, does it make sense to establish ground truth for only suspect cases (i.e., cases where the model gives a high probability of fraud)?

At first glance, the approach seems reasonable; however, a critical mind understands that this creates a feedback loop that will amplify the flaws of the model. Fraud patterns that were never captured by the model (i.e., those that have a low fraud probability according to the model) will never be taken into account in the retraining process.

One solution to this challenge might be to randomly label, establishing a ground truth for just a subsample of transactions in addition to those that were flagged as suspicious. Another solution might be to reweigh the biased sample so that its characteristics match the general population more closely. For example, if the system awarded little credit to people with low revenue, the model should reweight them according to their importance in the applicant or even in the general population.

The bottom line is that, whatever the mitigation measure, the labeled sample subset must cover all the possible future predictions so that the trained model makes good predictions whatever the sample, as it will sometimes mean making suboptimal decisions for the sake of checking that the model continues to generalize well.

Once this problem is solved for retraining, the solution (reweighting, random sampling) can be used for monitoring. Input drift detection complements this approach, as it is needed to make sure that ground truth covering new, unexplored domain is made available to retrain the model.

Input Drift Detection

Given the challenges and limitations of ground truth retraining presented in the previous section, a more practical approach might be input drift detection. This section will take a brief but deep dive into the underlying logic behind drift as well as present different scenarios that can cause models and data to drift.

For a relatable example, say the goal is to predict the quality of Bordeaux wines using the UCI Wine Quality dataset as training data, which contains information about red and white variants of the Portuguese "Vinho Verde" along with a quality score varying between 0 and 10.

The following features are provided for each wine: type, fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxide, density, pH, sulphates, and alcohol rate.

To simplify the modeling problem, say that a good wine is one with a quality score equal to or greater than 7. The goal is thus to build a binary model that predicts this label from the wine's attributes. For purposes of demonstrating data drift, we explicitly split the original dataset into two:

- The first one contains all wines with an alcohol rate above 11%, wine_alcohol_above_11.
- The second one contains those with an alcohol rate below 11%, wine_alcohol_below_11.

We split wine_alcohol_above_11 to train and score our model, and the second dataset wine_alcohol_below_11 will be considered as new incoming data that needs to be scored once the model has been deployed.

We have artificially created a big problem—it is very unlikely that the quality of wine is independent from the alcohol level. Worse, the alcohol level is likely to be correlated differently with the other features in the two datasets. As a result, what is learned on one dataset ("if the residual sugar is low and the pH is high, then the probability that the wine is good is high") may be wrong on the other one because, for example, the residual sugar is not important anymore when the alcohol level is high.

Mathematically speaking, the samples of each dataset cannot be assumed to be drawn from the same distribution (i.e., they are not "identically distributed"). Another mathematical property is necessary to ensure that ML algorithms perform as expected: independence. This property is broken if samples are duplicated in the dataset or if it is possible to forecast the "next" sample given the previous one, for example.

Let's assume that despite the obvious problems, we train the algorithm on the first dataset and then deploy it on the second one. The resulting distribution shift is then called a drift. It will be called a feature drift if the alcohol level is one of the features used by the ML model (or if the alcohol level is correlated with other features used by the model) and a concept drift if it is not.

Drift Detection in Practice

As explained previously, to be able to react in a timely manner, model behavior should be monitored solely based on the feature values of the incoming data without waiting for the ground truth to be available.

The logic is that if the data distribution (e.g., mean, standard deviation, correlations between features, etc.) diverges between the training and testing phases on one side and the development phase on the other, it is a strong signal that the model's performance won't be the same. It is not the perfect mitigation measure, as retraining on the drifted dataset will not be an option, but it can be part of mitigation measures (e.g., reverting to a simpler model, reweighting differently, etc.).

Example Causes of Data Drift

There are two frequent root causes of data drift:

- Sample selection bias, where the training sample is not representative of the population. For instance, building a model to assess the effectiveness of a discount program will be biased if the best discounts are proposed for the best clients. Selection bias often stems from the data collection pipeline itself. In the wine example, the original dataset sample with alcohol levels above 11% surely does not represent the whole population of wines—this is sample selection at its best. It could have been mitigated if a few samples of wine with an alcohol level above 11% had been kept and reweighted according to the expected proportion in the population of wines to be seen by the deployed model. Note that this task is easier said than done in real life, as the problematic features are often unknown or maybe even not available.
- Non-stationary environment, where training data collected from the source population does not represent the target population. This often happens for time-dependent tasks—such as forecasting use cases—with strong seasonality effects, where learning a model over a given month won't generalize to another month. Back to the wine example again: one can imagine a case where the original dataset sample only includes wines from a specific year, which might represent a particularly good (or bad) vintage. A model trained on this data may not generalize to other years.

Input Drift Detection Techniques

After understanding the possible situations that can cause different types of drift, the next logical question is: how can drift be detected? This section presents two common approaches. The choice between them depends on the expected level of interpretability.

Organizations that need proven and explainable methods should prefer univariate statistical tests. If complex drift involving several features simultaneously is expected or if the data scientists want to reuse what they already know—and assuming the organization doesn't dread the black-box effect—the domain classifier approach may be a good option, too.

Univariate Statistical Tests

This method requires—for each feature—applying a statistical test on data from the source distribution and the target distribution. A warning will be raised when the results of those tests are significant.

The choice of hypothesis tests have been extensively studied in the literature, but the basic approaches rely on these 2 tests:

- For continuous features, Kolmogorov-Smirnov test is a non-parametric hypothesis test that is used to check whether two samples come from the same distribution. It measures a distance between the empirical distribution functions.
- For categorical features, Chi-squared test is a practical choice that checks whether the observed frequencies for a categorical feature in the target data match the expected frequencies seen from the source data.

The main advantage of p-values is that they help detect drift as quickly as possible. On the other hand, the main drawback is that they detect an effect, but they do not quantify the level of the effect (i.e., on large datasets, they detect very small changes which may be completely impactless).

As a result, if development datasets are very large, it is necessary to complement them with business-significant metrics. For example, on a sufficiently large dataset, the average age may have significantly drifted from a statistical perspective, but if the drift is only a few months, this is probably an insignificant value for many business use cases.

Domain Classifier

In this approach, a data scientist trains a model that tries to discriminate between the original dataset (input features and, optionally, predicted target) and the one from the development dataset. In other words, (s)he stacks the two datasets and trains a classifier that aims at predicting data's origin. The performance of the model (its accuracy, for example) can then be considered as a metric for the drift level.

If this model is successful in its task, and thus has a high drift score, it implies that data used at training time and new data can be distinguished, so it's fair to say that the new data has drifted. To gain more insights, in particular to identify the feature(s) that are responsible for the drift, one can use the feature importance of the trained model.

Interpretation of Results

Both domain classifier and univariate statistical tests point to the importance of features or on the target to explain drift. Drift attributed to the target is important to identify because it often directly impacts the bottom line of the business (think, for example, of credit score: if the scores are lower overall, the number of awarded loans is likely to be lower, and therefore the revenues).

Drift attributed to features is useful to mitigate the impact, as it may hint at the need for:

- Reweighting according to this feature (e.g., if customers above 60 now represent 60% of users but it was only 30% in the train set, then double their weight and retrain the model).
- Removing the feature and training a new model without it.

In all cases, it is very unlikely that automatic actions exist if drift is detected. It could happen if it is costly to deploy retrained models: the model would be retrained on new data only if the ground-truth evaluated performance has dropped or if significant drift is detected. In this peculiar case, new data is indeed available to mitigate the drift.

The Feedback Loop

The data feedback loop was presented by Martin Fowler in his article about Continuous Delivery for Machine Learning. The idea is that all effective machine learning projects implement a form of data feedback loop; that is, information from the production environment flows back to the model prototyping environment for further improvement.

One can see in Figure 5-1 that data collected in the Monitoring and Observability phase is sent to the Model Building phase (details about this data is covered in Chapter 6). From there, the system analyzes whether the model is working as expected. If it is the case, no action is required. If the model's performance is degrading, an update will be triggered, either automatically or manually by the data scientist. In practice, as seen at the beginning of this chapter, this usually means either retraining the model with new labeled data or developing a new model with additional features.

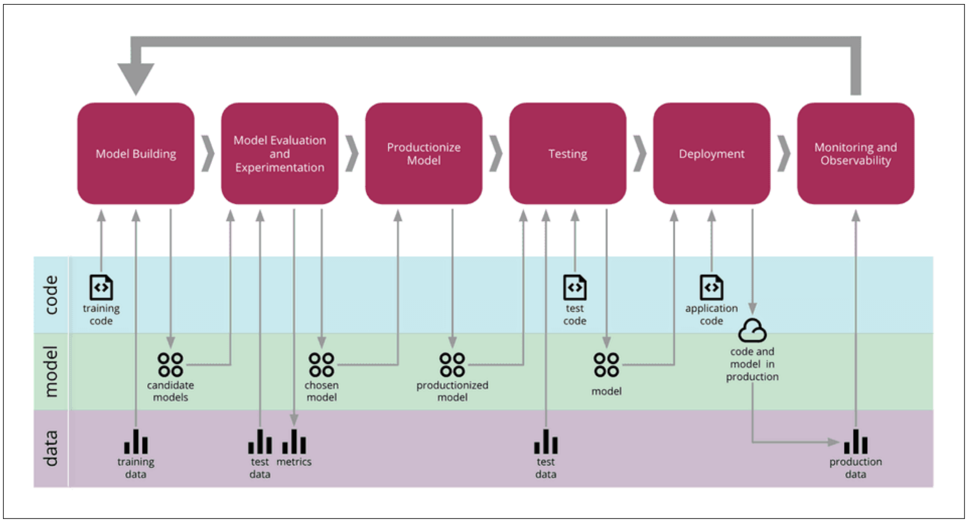


Figure 5-1. Continuous Delivery for end-to-end machine learning process; source. Note: The idea is not ultimately to use this exact image, but rather to use the image of the life-cycle presented in Chapter 1 and then build on it in Chapter 6 and then again in this chapter.

In either case, the goal is to be able to capture the emerging patterns and make sure that the business is not negatively impacted. This infrastructure is comprised of three main components, which in addition to the concepts discussed in the first part of this chapter, are critical to robust MLOps capabilities:

- A logging system that collects data from several production servers.
- A model evaluation store that does versioning and evaluation between different model versions.
- An online system that does model comparison on production environments, either with the shadow scoring (champion/challenger) setup or with A/B testing.

The following sections will address each of these components individually, including their purpose, key features, and challenges.

Logging

Monitoring a live system—with or without machine learning components—means collecting and aggregating data about its states. Nowadays, as production infrastructures are getting more and more complex with several models deployed simultaneously across several servers, an effective logging system is more important than ever.

Data from these environments need to be centralized in a place to be analyzed and monitored, either automatically or manually. This will enable continuous improvement of the ML system. An event log of a machine learning system is a record with a timestamp as well as the information presented in Table 7-1.

Model metadata	Identification of the model and the version.
Model inputs	Feature values of new observations, which allows for verification of whether the new incoming data is what the model was expecting and thus allowing for detection of data drift (as explained in the previous section).
Model outputs	Predictions made by the model which, along with the ground truth collected later on, give a concrete idea about the model performance in a production environment.
System action	It's rare that the model prediction is the end product of a machine learning application; the more common situation is that the system will take an action based on this prediction. For example, in a fraud detection use case, when the model gives high probability, the system can either block the transaction or it can send a warning to the bank. This type of information is important because it affects the user reaction and thus indirectly affects the feedback data.
Model explanation	In some highly regulated domains such as finance or healthcare, predictions must come with an explanation (i.e., which features have the most influence on the prediction). This kind of information is usually computed with techniques as LIME or Shapley and should be logged to identify potential issues with the model (e.g., bias, overfitting, etc).

Requirements and Challenges

As mentioned earlier, when deploying a model, there are several possible scenarios:

- One model deployed on one server
- One model deployed on multiple servers
- Multiple versions of a model deployed on one server
- Multiple versions of a model deployed on multiple servers
- Multiple versions of multiple models deployed on multiple servers

Therefore, an effective logging system should be able to generate centralized datasets that can be exploited by the model designer or the ML engineer, usually outside of the production environment. More specifically, it should cover all of the following situations:

- The system can access and retrieve scoring logs from multiple servers, either in a real-time scoring use case or in a batch scoring use case.
- When a model is deployed on multiple servers, the system can handle the mapping and aggregation of all information per model across servers.

- When different versions of a model are deployed, the system can handle the mapping and aggregation of all information per version of the model across servers.

In terms of challenges, for large-scale machine learning applications, the number of raw event logs generated can be an issue if there are no preprocessing steps in place to filter and aggregate data. For real-time scoring use cases, logging streaming data requires setting up a whole new set of tooling that entails a significant engineering effort to maintain. However, in both cases, as the goal of monitoring is usually to estimate aggregate metrics, saving only a subset of the predictions may be acceptable in many cases.

Model Evaluation Store

With a logging system in place, periodically that system fetches data from the production environment for monitoring. Everything goes well until one day, the data drift alert is triggered: the incoming data distribution is drifting away from the training data distribution. It is possible that the model performance is degrading.

After review, data scientists decide to retrain the model—using the techniques described earlier in this chapter—in order to improve it. With several trained candidate models, the next step is to compare them with the deployed model.

In practice, this means evaluating all the models (the candidates as well as the deployed model) on the same dataset. If one of the candidate models outperforms the active model, there are two ways to proceed: either to update the model on the production environment, or move to an online evaluation via a champion/challenger or A/B testing setup (more details on these setups to come in the following sections).

In a nutshell, this is the notion of model store—it is a structure that allows data scientists to:

- Compare multiple, newly trained model versions against existing deployed versions.
- Compare completely new models against versions of other models on labeled data.
- Track model performance over time.

Formally, the Model Evaluation Store serves as a structure that centralizes the data related to model lifecycle to allow comparisons (though note that comparing models makes sense only if they address the same problem). By definition, all these comparables are grouped under the umbrella of a logical model.

Logical Model

Building a machine learning application is an iterative process, from deploying to production, monitoring performance, retrieving data, and looking for ways to improve the way the system addresses the target problem. There are many ways to iterate, some of which have already been discussed in this chapter, including:

- Retraining the same model on new data
- Adding new features to the model
- Developing new algorithms

For those reasons, the machine learning model itself is not a static object; it constantly changes with time. It is therefore helpful to have a higher abstraction level to reason about machine learning applications, thus the necessity of presenting the term logical model.

A logical model is a collection of model templates and their versions that aims at solving a business problem. A model version is obtained by training a model template on a given dataset. All versions of model templates of the same logical model can usually be evaluated on the same kinds of datasets —i.e., on datasets with the same feature definition and/or schema—however, this may not be the case if the problem did not change but the features available to solve it did. Model versions could be implemented using completely different technologies, and there could even be several implementations of the same model version (Python, SQL, Java, etc.); regardless, they are supposed to give the same prediction if given the same input.

Let's get back to the wine example introduced earlier in this chapter. Three months after deployment, let's say there is new data about less alcoholic wine. We can retrain our model on the new data, thus obtaining a new model version using the same model template. While investigating the result, we discover new emerging patterns. We may decide to create new features that capture this information and add it to the model, or we may decide to use another ML algorithm (like deep learning) instead of XGBoost. This would result in a new model template.

As a result, our model has two model templates and three versions:

- The first version that is live in production, based on the original model template.
- The second version, still based on the original template, but it is trained on new data.
- The third version uses the deep learning-based template with additional features, and is trained on the same data as the second version.

The information about the evaluation of these versions on various datasets (both the test datasets used at training time and the development datasets that may be scored after training) is then stored in the model evaluation store.

Model Evaluation Store

The two main tasks of a model evaluation store are:

1. Versioning the evolution of a logical model through time. Each logged version of the logical model must come with all the essential information concerning its training phase, including:

   The list of features used

   The preprocessing techniques that are applied to each feature

   The algorithm used, along with the chosen hyperparameters

   The training dataset

   The test dataset used to evaluate the trained model (this is necessary for the version comparison phase)

   Evaluation metrics

2. Comparing the performance between different versions of a logical model. To decide which version of a logical model to deploy, all of them must be evaluated (the candidates and the deployed one) on the same dataset.

The choice of dataset to evaluate is crucial. If there is enough new labeled data to give a reliable estimation of the model performance, this is the preferred choice—it is the closest data to what we are expecting to receive in the production environment.

Otherwise, another option is to use the original test set of the deployed model. Assuming that the data has not drifted, this gives us a concrete idea about the performance of the candidate models compared to the original one.

After identifying the best candidate model, the job is not done yet. In practice, there is often a substantial discrepancy between the offline and online performance of the models. Therefore, it's critical to take the testing to the production environment. This online evaluation gives the most truthful feedback about the behavior of the candidate model when facing real data.

Online Evaluation

There two main modes of online evaluation are:

- Champion/challenger (otherwise known as shadow testing), where the candidate model shadows the deployed model and scores the same live requests.

- A/B testing, where the candidate model scores a portion of the live requests and the deployed model scores the others.

Note that both cases require ground truth, so the evaluation will necessarily take longer than the lag between prediction and ground truth obtention. In addition, whenever shadow testing is possible, it should be used over A/B testing because it is far simpler to understand and to set up, and what's more, it detects differences more quickly.

Champion / Challenger

Champion/challenger (otherwise known as shadow testing) involves deploying one or several additional models (the challengers) to the production environment. These models receive and score the same incoming requests as the active one (the champion model). However, they do not return any response or prediction to the system: that's still the job of the old model—the predictions are simply logged for further analysis. That's why it is also called "shadow model" or "dark launch."

This setup allows us for two things:

- Verification that the performance of the new models are better, or at least as good as the old one. As the two models are scoring on the same data, there is a direct comparison of the accuracy of the two models in the production environment. Note that this could also be done offline by using the new models on the dataset made of new requests scored by the champion model.
- Measurement of how the model handles realistic load. As the new model can have new features, new preprocessing techniques, or even a new algorithm, the prediction time for a request won't be the same as that of the original one, and it is important to have a concrete idea of this change. Of course, this is the main advantage of doing it online.

The other advantage of this deployment scheme is that the data scientist or the ML engineer is giving visibility to other stakeholders on the future champion model: instead of being locked in the data science environment, the challenger model results are exposed to the business leaders, which decreases the perceived risk to switch to a new model.

To be able to compare the two champion/challenger models, the same information must be logged for both, including input data, output data, processing time, etc. This means updating the logging system so that it can differentiate between the two sources of data.

How long should both models be deployed before it's clear that one is better than the other? Intuitively, long enough so that the metric fluctuations due to randomness are dampened because enough predictions have been made.

This can be assessed graphically by checking that the metric estimations are not giggling anymore or by doing a proper statistical test (as most metrics are averages of row-wise scores, the most usual test is a paired sample T-test) that yields the probability that the observation that a metric is higher than the other is due to these random fluctuations. The wider the metric difference, the fewer predictions necessary to be sure enough that the difference is significant.

Depending on the use case and the implementation of the champion/challenger system, server performance can be a concern. If two memory-intensive models are called synchronously, they can slow the system down. This will not only have a negative impact on the user experience but also corrupt the data collected about the functioning of the models.

Another concern is the communication with the external system. If the two models use some external API to enrich their features, that doubles the number of requests to these services, thus doubling costs. If that API service has some caching system in place, then the second request will be processed much faster than the first one, which can bias the result when comparing the total prediction time of the two models. Note that the challenger may be used only for a random subset of the incoming requests, which will alleviate the load at the expense of increased time before a conclusion can be drawn.

Finally, when implementing a challenger model, it's important to ensure it doesn't have any influence on the system's actions. This implies two scenarios:

- When the challenger model encounters an unexpected issue and fails, the production environment will not experience any discontinuation or degradation in terms of response time.
- Actions taken by the system depend only on the prediction of the champion model, and they happen only once. For example, in a fraud detection use case, imagine that by mistake the challenger model is plugged directly to the system, charging each transaction twice—a catastrophic scenario.

In general, some effort needs to be spent on the logging, monitoring, and serving system to make sure that the production environment functions as usual and is not impacted by any issues coming from the challenger model.

A/B testing

A/B testing is a widely used technique in website optimization. For ML models, it should be used only when champion/challenger is not possible. This might happen when:

The ground truth cannot be evaluated for both models. For example, for a recommendation engine, the prediction gives a list of items on which a given customer is likely to click if they are presented. Therefore, it is impossible to know if the customer would have clicked if it is not presented, so some kind of A/B testing will have to be done. In other words, some customers will be shown the recommendations of model A, some other the recommendations of model B. Similarly, for a fraud detection model, as heavy work is needed to obtain the ground truth, it may not be possible to do it for the positive predictions of two models because it would increase the workload too much, as some frauds are detected by only one model. As a result, randomly applying only the B model to a small fraction of the requests will allow for the workload to remain constant.

The objective to optimize is only indirectly related to the performance of the prediction. Imagine an ad engine based on an ML model that predicts if a user will click on the ad. Now imagine that it is evaluated on the buy rate, i.e., whether the user bought the product or service. Once again, it is not possible to record the reaction of the user for two different models, so in this case, A/B testing is the only way.

Complete books are dedicated to A/B testing; this section just presents its main idea and a simple walkthrough. Unlike the champion/challenger framework, with A/B testing, the candidate model returns predictions for certain requests, and the original model handles the other ones. Once the test period is over, statistical tests compare the performance of the two models, and teams can make a decision based on the statistical significance of those tests.

In an MLOps context, some considerations need to be made. A walkthrough of the considerations is available in Table 7-2.

Before the A/B test	Define a clear goal: i.e., a quantitative business metric that needs to be optimized. For example, click-through rate. Define a precise population: Carefully choosing a segment for the test along with a splitting strategy that assures no bias between groups (this is the so-called experimental design or randomized control trial that's been popularized by drug studies). This may be a random split, but it may well be way more complex. For example, the situation might dictate that all the requests of a particular customer are handled by the same model. Define the statistical protocol: the resulting metrics are compared using statistical tests, and the null hypothesis is either rejected or retained. To make the conclusion robust, teams need to define beforehand the sample size for the desired minimum effect size, which is the minimum difference between the two models' performance metrics. Teams must also fix a test duration (or alternatively have a method to handle multiple tests). Note that with similar sample sizes, the power to detect meaningful differences will be lower than with champion/challenger because unpaired sample tests have to be used (it is usually impossible to match each request scored with model B with a request scored with model A; with champion/challenger, this is trivial).
During the A/B test	The easiest scheme consists of not stopping the experiment before the test duration is over, even if the statistical test starts to return a significant metric difference. This practice (also called p-hacking) produces unreliable and biased results due to cherry picking the desired outcome.
After the A/B test	Once the test duration is over, check the collected data to make sure that the quality is good. From there, run the statistical tests; if the metric difference is statistically significant in favor of the candidate model, the original model can be replaced with the new version.

Closing Thoughts

Ordinary software is built to satisfy specifications. Once an application is deployed, its ability to fulfill its objective does not degrade. On the contrary, ML models have objectives statistically defined by their performance on a given dataset. As a result, their performance changes—usually for worse—when the statistical properties of the data change.

In addition to ordinary software maintenance needs (bug correction, release upgrades, etc.), this performance drift has to be carefully monitored. We have seen that ground-truth based performance monitoring is the cornerstone, while drift monitoring can provide early warning signals. Among possible drift mitigation measures, the workhorse is definitely retraining on new data, while model modification remains an option. Once a new model is ready to be deployed, its improved performance can be validated thanks to shadow scoring or, as a second choice, A/B testing. This allows proving that the new model is better in order to improve the performance of the system.

About the Authors

Clément Stenac is a passionate software engineer, CTO and co-founder at Dataiku. He oversees the design, development of the Dataiku DSS Entreprise AI Platform. Clément was previously head of product development at Exalead, leading the design and implementation of web-scale search engine software. He also has extensive experience with open source software, as a former developer of the VideoLAN (VLC) and Debian projects.

Léo Dreyfus-Schmidt is a mathematician and holds a PhD in pure mathematics from University of Oxford and University of Paris VII. After five years focusing on homological algebra and representation theory in Paris, Oxford, and the University of California - Los Angeles, he joined Dataiku where he has been developing solutions for predictive maintenance, personalized ranking systems, price elasticity, and natural language applications.

Kenji Lefèvre is VP Product at Dataiku. He oversees the product roadmap and the user experience of the Dataiku DSS Entreprise AI Platform. He holds a PhD in pure mathematics from University of Paris VII, and he directed documentary movies before switching to Data Science and product management.

Nicolas Omont is a Product Manager at Dataiku in charge of Machine-Learning and advanced analytics. He holds a PhD in Computer Science, and he's been working in operations research and statistics for the past 15 years mainly in the telecommunication and in the energy utility sectors.

Mark Treveil has designed products in fields as diverse as telecoms, banking, and online trading. His own startup led a revolution in governance in the UK local government, where it still dominates. He is now part of the Dataiku Product Team based in Paris.