ML Ops: Machine Learning as an Engineering Discipline

As ML matures from research to applied business solutions, so do we need to improve the maturity of its operation processes

Cristiano Breuel
Jan 3 · 10 min read

So, your company decided to invest in machine learning. You have a talented team of Data Scientists churning out models to solve important problems that were out of reach just a few years ago. All performance metrics are looking great, the demos cause jaws to drop and executives to ask how soon you can have a model in production.

It should be pretty quick, you think. After all, you already solved all the advanced scienc-y, math-y problems, so all that's left is routine IT work. How hard can it be?

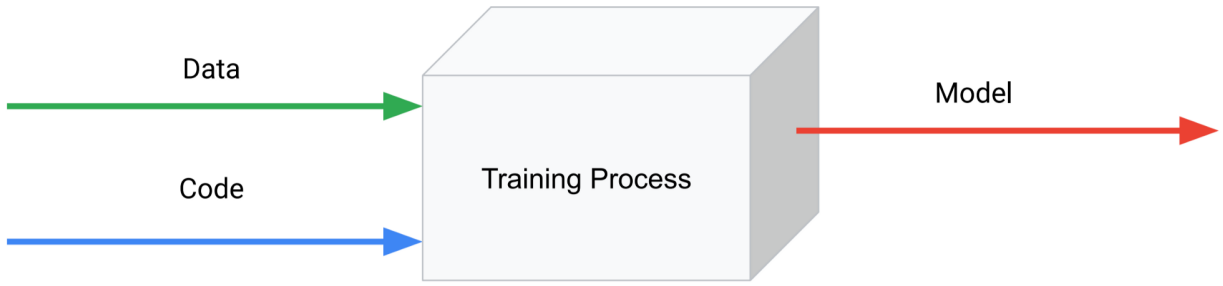
Pretty hard, it turns out. Deeplearning.ai reports that "only 22 percent of companies using machine learning have successfully deployed a model". What makes it so hard? And what do we need to do to improve the situation?

Let's start by looking at the root causes.

Challenges

In the world of traditional software development, a set of practices known as **DevOps** have made it possible to ship software to production in minutes and to keep it running reliably. DevOps relies on tools, automation and workflows to abstract away the accidental complexity and let developers focus on the actual problems that need to be solved. This approach has been so successful that many companies are already adept at it, so why can't we simply keep doing the same thing for ML?

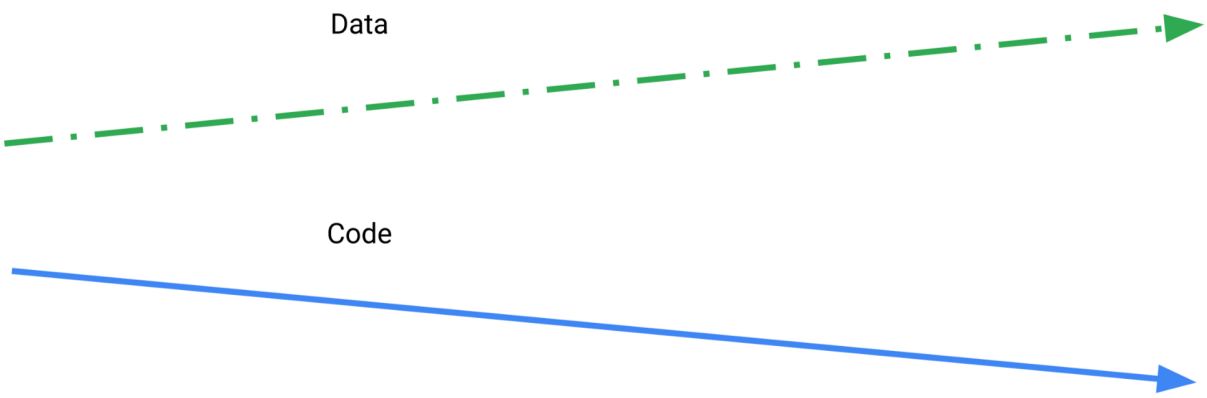
The root cause is that there's a fundamental difference between ML and traditional software: **ML is not just code, it's code plus data**. An ML model, the artifact that you end up putting in production, is created by applying an algorithm to a mass of training data, which will affect the behavior of the model in production. Crucially, the model's behavior also depends on the input data that it will receive at prediction time, which you can't know in advance.



ML = Code + Data

While code is carefully crafted in a controlled development environment, data comes from that unending entropy source known as "the real world". It never stops changing, and you can't control how it will change. A useful way to think of their relationship is as if code and data live in separate planes, which share the time dimension but are independent in all others.

The challenge of an ML process is to create a bridge between these two planes in a controlled way.



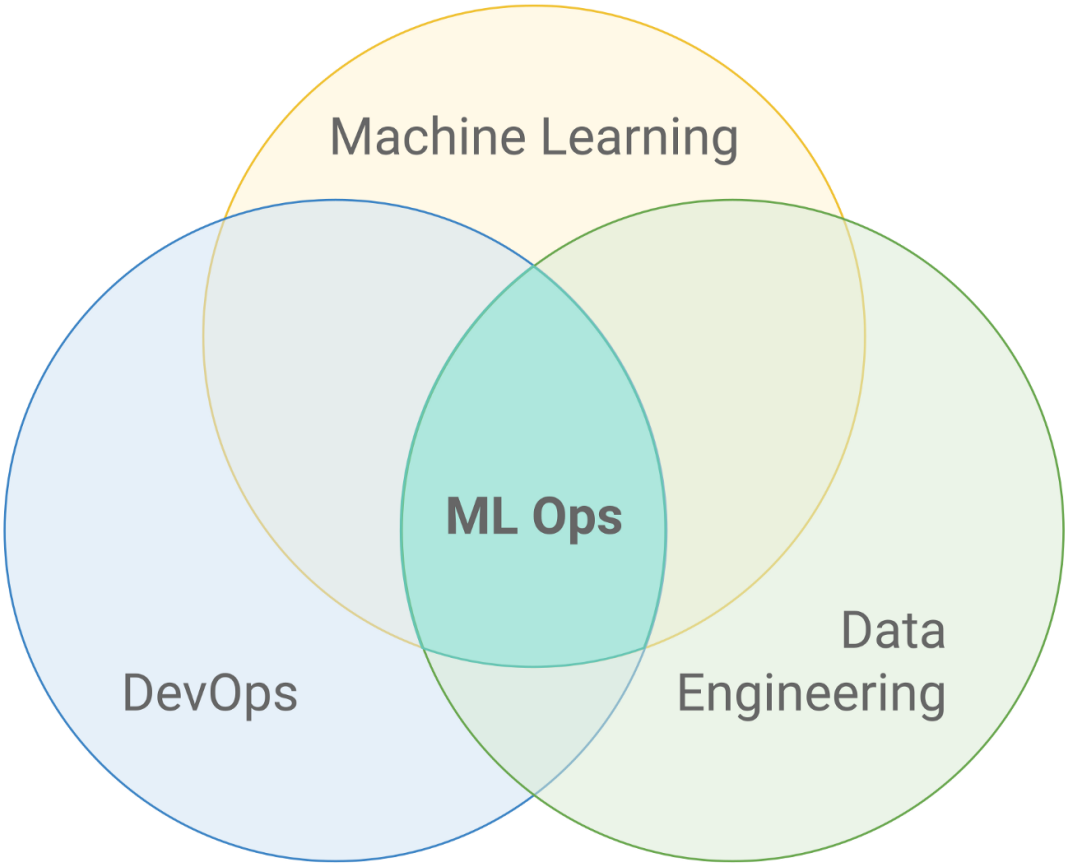
Code and data evolve independently. We can think of them as separate planes with a common time dimension.

This fundamental disconnect causes several important challenges that need to be solved by anyone trying to put an ML model in production successfully, for example:

- Slow, brittle and inconsistent deployment
- Lack of reproducibility
- Performance reduction (training-serving skew)

Since the word “data” has been already used several times in this article, you may be thinking of another discipline that could come to our rescue: **Data Engineering**. And you would be right: Data Engineering does provide important tools and concepts that are indispensable to solving the puzzle of ML in production. In order to crack it, we need to combine practices from DevOps and Data Engineering, adding some that are unique to ML.

Thus, ML Ops can be defined by this intersection:



ML Ops is the intersection of Machine Learning, DevOps and Data Engineering

Thus, we could define ML Ops as follows:

> ***ML Ops is a set of practices that combines Machine Learning, DevOps and Data Engineering, which aims to deploy and maintain ML systems in production reliably and efficientl y.***

Let's now see what this actually means in more detail, by examining the individual practices that can be used to achieve ML Ops' goals.

Practices

Hybrid Teams

Since we've already established that productionizing an ML model requires a set of skills that so far were considered separate, in order to be successful we need a hybrid team that, together, covers that range of skills. It is of course possible that a single person might be good enough at all of them, and in that case we could call that person a full **ML Ops Engineer**. But the most likely scenario right now is that a successful team would include a Data Scientist or ML Engineer, a DevOps Engineer and a Data Engineer.

The exact composition, organization and titles of the team could vary, but the essential part is realizing that a Data Scientist alone cannot achieve the goals of ML Ops. Even if an organization includes all necessary skills, it won't be successful if they don't work closely together.

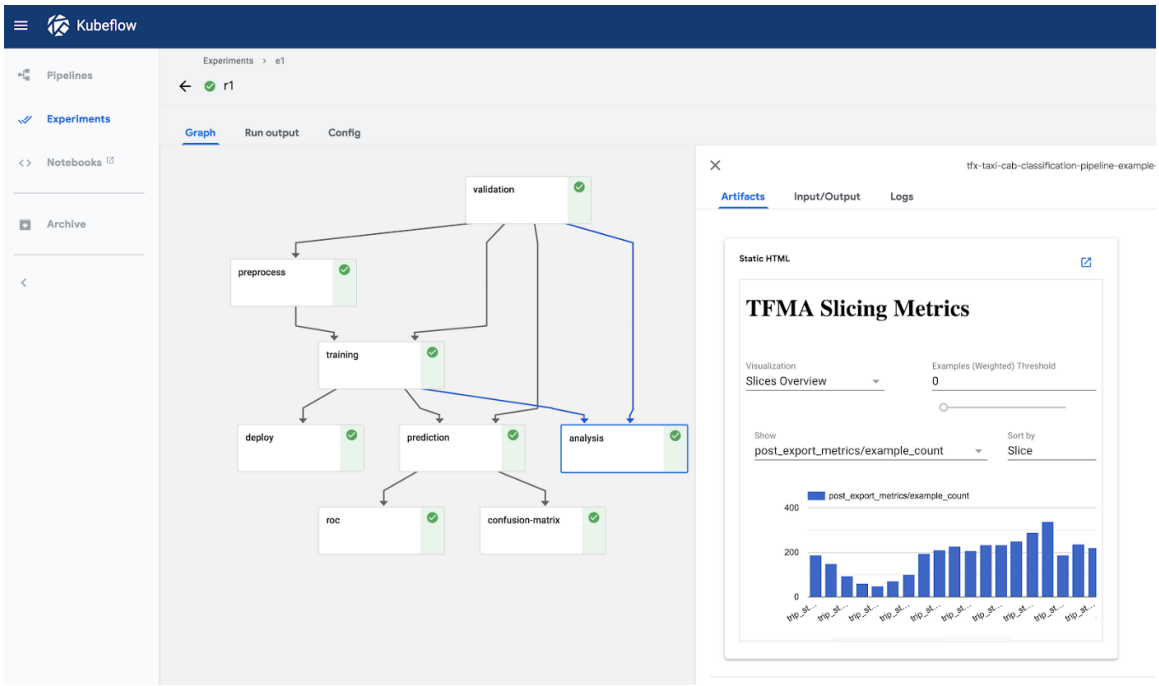
Another important change is that Data Scientists must be proficient in basic software engineering skills like code modularization, reuse, testing and versioning; getting a model to work great in a messy notebook is not enough. This is why many companies are adopting the title of ML Engineer, which emphasizes these skills. In many cases, ML Engineers are, in practice, performing many of the activities required for ML Ops.

ML Pipelines

One of the core concepts of Data Engineering is the **data pipeline**. A data pipeline is a series of transformations that are applied to data between its source and a destination. They are usually defined as a graph in which each node is a transformation and edges represent dependencies or execution order. There are many specialized tools that help create, manage and run these pipelines. Data pipelines can also be called ETL (extract, transform and load) pipelines.

ML models always require some type of data transformation, which is usually achieved though scripts or even cells in a notebook, making them hard to manage and run reliably. Switching to proper data pipelines provides many advantages in code reuse, run time visibility, management and scalability.

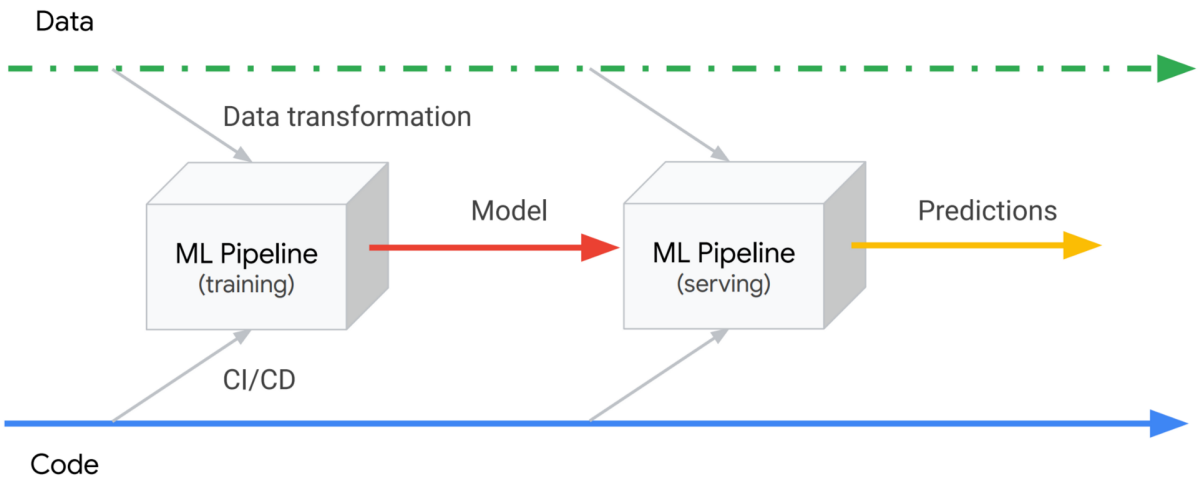
Since ML training can also be thought of as a data transformation, it is natural to include the specific ML steps in the data pipeline itself, turning it into an **ML Pipeline**. Most models will need 2 versions of the pipeline: one for training and one for serving. This is because, usually, the data formats and way to access them are very different between each moment, especially for models that are served in real-time requests (as opposed to batch prediction runs).



Visual representation of an ML pipeline in Kubeflow Pipelines

The ML Pipeline is a pure code artifact, independent from specific data instances. This means that it's possible to track its versions in source

control and automate its deployment with a regular **CI/CD pipeline**, a core practice from DevOps. This lets us connect the code and data planes in a structured and automated way:



ML Pipelines connect data and code to produce models and predictions

Note that there are two distinct ML pipelines: the training pipeline and the serving pipeline. What they have in common is that the data transformations that they perform need to produce data in the same format, but their implementations can be very different. For example, the training pipeline usually runs over batch files that contain all features, while the serving pipeline often runs online and receives only part of the features in the requests, retrieving the rest from a database.

It is important, however, to ensure that these two pipelines are consistent, so one should try to reuse code and data whenever possible. Some tools can help with that goal, for example:

- Transformation frameworks like TensorFlow Transform can ensure that calculations based on training set statistics, like averages and standard deviations for normalization, are consistent.

- Feature Stores are databases that store values that are not part of a prediction request, for example features that are calculated over a user's history.

Model and Data Versioning

In order to have reproducibility, consistent version tracking is essential. In a traditional software world, versioning code is enough, because all behavior is defined by it. In ML, we also need to track model versions, along with the data used to train it, and some meta-information like training hyperparameters.

Models and metadata can be tracked in a standard version control system like Git, but data is often too large and mutable for that to be efficient and practical. It's also important to avoid tying the model lifecycle to the code lifecycle, since model training often happens on a different schedule. It's also necessary to version data and tie each trained model to the exact versions of code, data and hyperparameters that were used. The ideal solution would be a purpose-built tool, but so far there is no clear consensus in the market and many schemes are used, most based on file/object storage conventions and metadata databases.

Model validation

Another standard DevOps practice is test automation, usually in the form of unit tests and integration tests. Passing these tests is a prerequisite for a new version to be deployed. Having comprehensive automated tests can give great confidence to a team, accelerating the pace of production deployments dramatically.

ML models are harder to test, because no model gives 100% correct results. This means that model validation tests need to be necessarily statistical in nature, rather than having a binary pass/fail status. In order

to decide whether a model is good enough for deployment, one needs to decide on the right metrics to track and the threshold of their acceptable values, usually empirically, and often by comparison with previous models or benchmarks.

It's also not enough to track a single metric for the entirety of the validation set. Just as good unit tests must test several cases, model validation needs to be done individually for relevant segments of the data, known as slices. For example, if gender could be a relevant feature of a model, directly or indirectly, tracking separate metrics for male, female and other genders. Otherwise, the model could have fairness issues or under-perform in important segments.

If you manage to get models validated in an automated and reliable way, along with the rest of the ML pipeline, you could even close the loop and implement online model training, if it makes sense for the use case.

Data validation

A good data pipeline usually starts by validating the input data. Common validations include file format and size, column types, null or empty values and invalid values. These are all necessary for ML training and prediction, otherwise you might end up with a misbehaving model and scratching your head looking for the reason. **Data validation is analogous to unit testing in the code domain**.

In addition to basic validations that any data pipeline performs, ML pipelines should also validate higher level statistical properties of the input. For example, if the average or standard deviation of a feature change considerably from one training dataset to another, it will likely affect the trained model and its predictions. This could be a reflection of actual change in the data or it could be an anomaly caused by how the data

is processed, so it's important to check and rule out systemic errors as causes that could contaminate the model, and fix them if necessary

An example of statistical data profiling in TensorFlow Data Validation

Monitoring

Monitoring production systems is essential to keeping them running well. For ML systems, monitoring becomes even more important, because their performance depends not just on factors that we have some control over, like infrastructure and our own software, but also on data, which we have much less control over. Therefore, in addition to monitoring standard metrics like latency, traffic, errors and saturation, we also need to monitor model prediction performance.

An obvious challenge with monitoring model performance is that we usually don't have a verified label to compare our model's predictions to,

since the model works on new data. In some cases we might have some indirect way of assessing the model's effectiveness, for example by measuring click rate for a recommendation model. In other cases, we might have to rely on comparisons between time periods, for example by calculating a percentage of positive classifications hourly and alerting if it deviates by more than a few percent from the average for that time.

Just like when validating the model, it's also important to monitor metrics across slices, and not just globally, to be able to detect problems affecting specific segments.

Summary

As ML matures from research to applied business solutions, so do we need to improve the maturity of its operation processes. Luckily, we can extend many practices from disciplines that came before ML.

The following table summarizes ML Ops' main practices and how they relate to DevOps and Data Engineering practices:

This is a brand new and exciting discipline, with tools and practices that are likely to keep evolving very quickly. There's certainly a lot of opportunity in developing and applying production techniques to ML.

. . .

References

- Kaz Sato — ML Ops Best Practices on Google Cloud (Cloud Next '19)
- Kaz Sato — What is ML Ops? Best Practices for DevOps for ML (Cloud Next '18)
- Kubeflow — The Machine Learning Toolkit for Kubernetes
- TensorFlow Extended (TFX) — an end-to-end platform for deploying production ML pipelines
- Rules of Machine Learning: Best Practices for ML Engineering

Data Science Machine Learning Data Engineering DevOps

Towards Data Science