Machine Learning Observability 101

A comprehensive guide on how to move beyond mere model monitoring

arize

Table of contents

Introduction

AI is everywhere. With global business spending on artificial intelligence (AI) and machine learning (ML) forecast to eclipse $200 billion by 2025, enterprises are urgently deploying models across nearly every facet of their businesses. Arize AI—a pioneer and early leader in ML observability—tracks billions of ML model predictions daily that influence everything from how crops are harvested to a customer's experience in a fast-food drive-through, including whether the credit card transaction gets approved.

Given the high stakes for both companies and society at large, it's more important than ever to have software that helps humans understand AI—and know how to fix it when it breaks. That is why Arize AI exists: to help teams troubleshoot the most complex systems ever built and provide guardrails when those systems make high-stakes decisions.

ML observability is how that mission is accomplished and the topic of this ebook. While ML monitoring alerts you when the performance of your model is degrading, ML observability helps you get to the bottom of *why*—a bigger, harder problem.

With ML observability, companies gain insight into model and feature drift detection, input and output data quality, model performance, and explainability. When problems arise, ML observability gives practitioners the ability to pinpoint why a model's performance is not as expected in production as well as clear signals for when they should retrain their model, update their training datasets, add new features to their model, or even go back to the drawing board.

What is ML Observability?

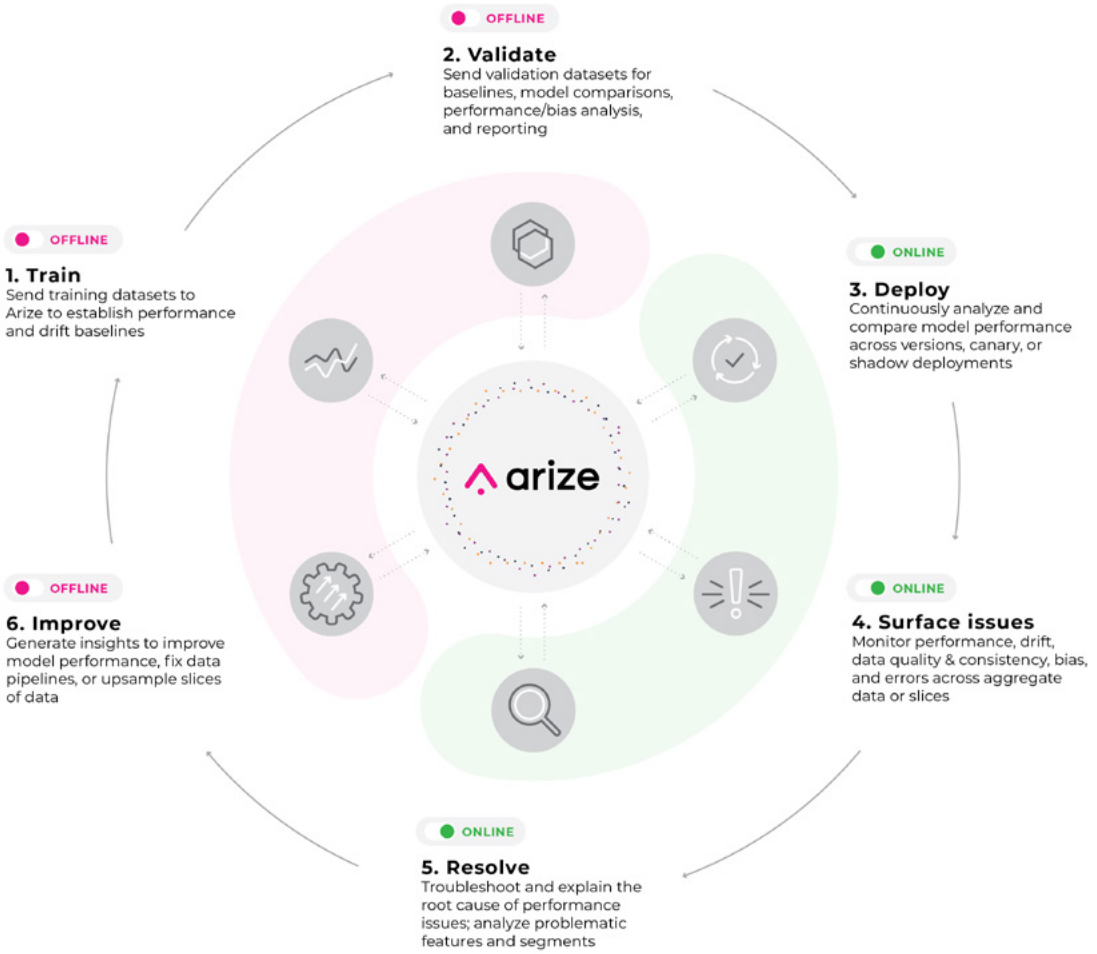
ML Observability is the practice of obtaining a deep understanding of a model's performance across all stages of the model development cycle: as it's being built, once it's been deployed, and long into its life in production.

In practice, ML observability is often the key difference between a team that flies blind after deploying a model and a team that can iterate and improve on their models quickly.

Is Model Observability just a fancy word for ML monitoring?

Model observability begins with the process of collecting model evaluations in environments such as training, validation, and production, then tying them together with analytics that allows one to connect these points to solve ML engineering problems. These inferences are stored in a model evaluation store (credit to Josh Tobin for this term), which hosts the raw inference data.
An evaluation store holds the response of the model, a signature of the model decisions, to every piece of input data for every model version, in every environment.

An ML observability platform allows teams to analyze model degradation and to root cause any issues that arise. This ability to diagnose the root cause of a model's issues, by connecting points across validation and production, is what differentiates model observability from traditional model monitoring. While model monitoring consists of setting up alerts on key model performance metrics such as accuracy, or drift, model observability implies a higher objective of getting to the bottom of any regressions in performance or anomalous behavior. *We are interested in the why*. Monitoring is interested in only aggregates and alerts. Observability is interested in what we can infer from the model's predictions, explainability insights, the production feature data, and the training data, to understand the cause behind model actions and build workflows to improve.

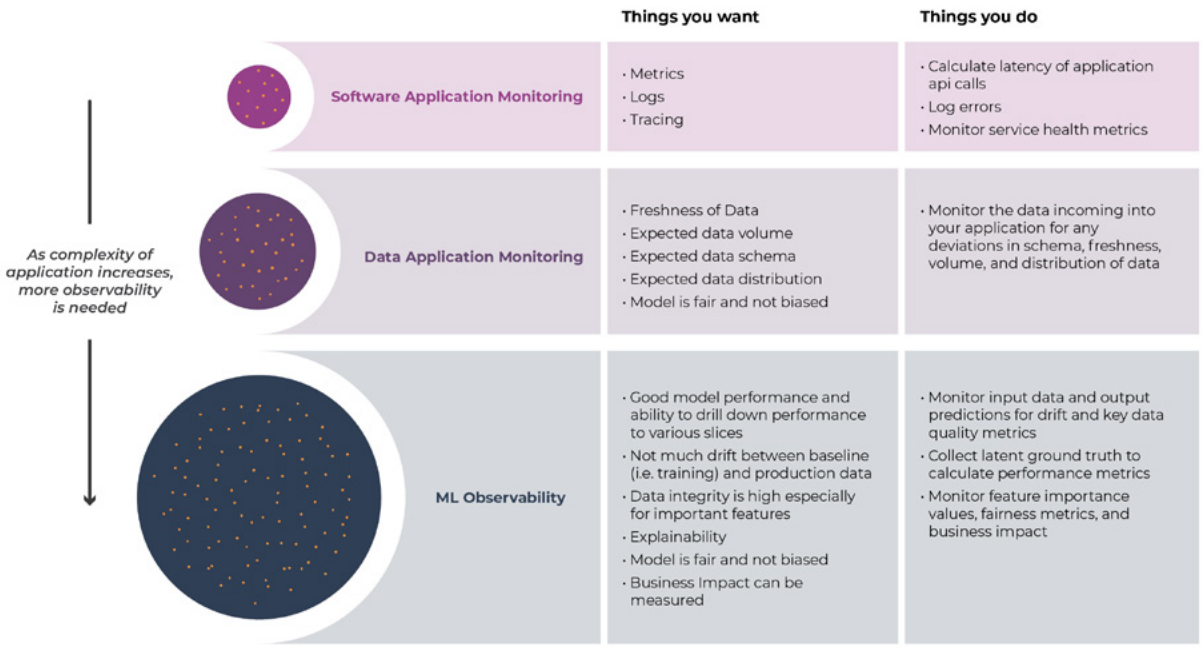


ML Observability backed by an Evaluation Store:

- Move seamlessly between production, training, and validation dataset environments
- Natively support model evaluation analysis by environment
- Designed to analyze performance Facets/Slices of predictions
- Explainability attribution designed for troubleshooting and regulatory analysis
- Performance analysis with ground truth—Accuracy, F1, MAE
- Proxy performance without ground truth—prediction drift
- Distribution drift analysis between data sets and environments
- Designed to answer the why behind performance changes
- Integrated validation
- Architected to iterate and improve

How does ML observability differ from data and software application observability?

It's also worth noting that ML observability is distinct from its equivalents in the world of software or data applications. Data observability, for example, typically uses tables as the base for monitoring and examines data and schema changes to get to the bottom of issues. Software or infrastructure observability traces and troubleshoots response times for a given app or system. Machine learning observability, in contrast, treats models as the base of monitoring and sets baselines from training, validation, or prior time periods in production to then compare shifts, perform analysis and root cause performance degradation.



Understanding Model Failure Modes

Once bought into the need for observability for detecting and diagnosing regressions in models, a question naturally arises: **what should I monitor in production**? The answer, of course, depends on **what can go wrong.**

In this section, we will be providing some more concrete examples of potential failure modes along with the most common symptoms that they exhibit in your production model's performance.

As with most things in the world, a model's task is likely to change over time. **Concept Drift** or **Model Drift** is a model failure mode that is caused by shifts in the underlying task that a model performs, which can gradually or suddenly cause a regression in the model's performance.

Concept Drift Example

Input → output relationship has changed

1930 DATA

Slang	Sentiment
Bad	Negative
Bash	Negative
Killer	Negative

1980 DATA

Slang	Sentiment
Bad	Positive
Bash	Positive
Killer	Positive

To put it a bit differently, *the task that your model was trained to solve may not accurately reflect the task it is now faced in production.*

For example, imagine that you are trying to predict the sentiment of a particular movie review and your model was trained on reviews from the early 1970s. If your model is any good, it has probably learned that the review, "Wow that movie was 'bad'", carries negative sentiment; however fast forward into the world of 1980s slang, and that exact same review might mean that the movie was fantastic.

So the input to the model didn't change, but the result did—what happened? The fundamental task of mapping natural language to sentiment has shifted since the model was trained, causing the model to start making mistakes where it used to predict correctly.

This shift can happen gradually over time, but as we are too often reminded these days, the world doesn't always change gradually. This fundamental unpredictability about when exactly concept drift is going to happen necessitates a good suite of model monitoring tools.

If you notice that not much has changed in the distribution of your model inputs, yet your model performance is regressing, concept drift may be a contributing factor.

Data lies at the very center of model creation, and data practices can make or break how a model performs in production. In the real world, the distribution of your model's inputs are almost certain to change over time, which leads us to our next model failure mode: Data Drift or Feature Drift.

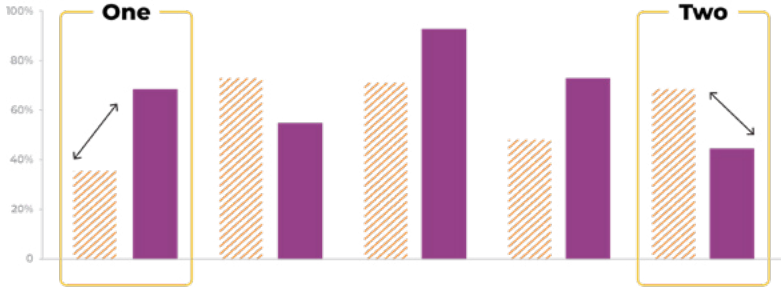
Feature Drifting from Training

Let's imagine you've just deployed a model that predicts how many streams an album is going to get in its first day on Spotify or Apple Music. Now that your model has been deployed to production for a while, and has been correctly predicting Drake's meteoric smash hits, you start to notice that your model is starting to make some dramatic mistakes.

Feature Drift Causing Model Performance Issues

After some inspection you notice that some new artists and trendy genres are attracting a majority of the streams, and your model is making large mistakes on these albums.

So what could have possibly gone wrong?



*Do changes in **one** and **two** cause poor model performance?*

The distribution in your model's inputs are statistically different from the distribution on which it was trained. In other words, your data has drifted and your model is now out of date. Just like concept drift, data drift can creep up on your model slowly or hit it hard and fast.

One important thing to mention is that it's quite common to confuse data drift with **training-prod skew**. Training-prod skew is when the distribution of your training data differs meaningfully from the distribution of production data. Going back to our previous example, if you had only trained your album model with country music, it will likely not perform well when it sees a new jazz album hit the discovery tab.

Training-production Skew

In both the case of data drift and training-prod skew, the model is experiencing a distribution of model inputs in production that it did not see while training, which can lead to worse performance than when validating the model. So how can we tell the difference?

Training vs Production

If your model is unable to achieve close to its validation performance as soon as you deploy you model, you likely are facing training-prod skew, and you might want to rethink your data sampling techniques to curate a more representative dataset; however if your model match has exhibited good production results in the past and you are seeing a slow or sudden dip in performance, it's very possible that you are dealing with the effects of **data drift**.

Cascading Model Failures

As machine learned models take the world by storm, it has become more and more common for products to contain a number of machine learned components. In many cases the output of one model is even used directly or indirectly as an input to another model.

Since these models are often trained and validated separately on their own datasets, a head-scratching failure mode is bound to pop up. While both models' performance can improve in their offline validation performance, they can regress the product as a whole when deployed together.

To demonstrate this concept, let's pretend that you work on Alexa. Your team is responsible for a speech recognition model that transcribes a user's speech into text, while a partner team is responsible for classifying these transcribed queries into an action that Alexa can perform for the user.

One day, you discover a breakthrough that improves your team's speech recognition results by 10% on the validation datasets; however, when you get back to your desk the next day after deploying the updated model you see a huge regression in Alexa's accuracy in selecting the action that the user requested.

What happened? Well, in making this change to the speech recognition model, the output distribution of the transcribed utterances changed in a statistically significant way. Since these outputs are the inputs to the action classification model, this caused the action predictions to regress, causing your users to sit through a terrible song when they just wanted to set a timer.

To diagnose these cascading model failures, a model observability tool must be able to track the changes to the input and output distributions of each model to be able pinpoint which model introduced the regression in overall performance.

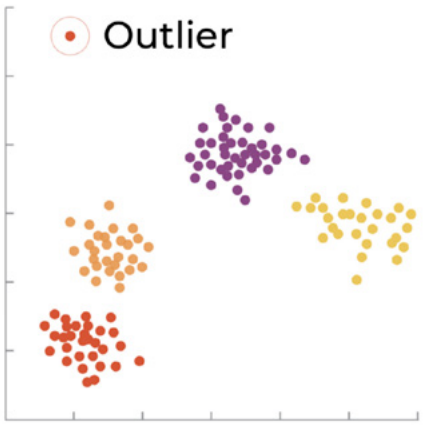
In the previous sections we have been primarily concerned with group statistics surrounding the model's performance; however, sometimes how your model is performing on a few examples is more concerning.

In an ideal world, there would be no surprises when you deploy your model to production. Unfortunately, we do not live in an ideal world. Your model will likely face anomalous inputs and occasionally produce anomalous results.

Sometimes finding these examples are like finding a needle in a haystack. You can't address the underlying problem in your model if you can't find it.

Outlier: Lower Dimensional Mapping

Understanding outliers is typically viewed as a multivariate analysis across all of the input features finding individual predictions that are outliers. Contrast this with the drift description above which is really a general statistic of a single feature over a group of predictions.



Drift: Group of Predictions—Univariate statistic on a feature or model output

Outlier: Individual Prediction or Small Group of Predictions—Multivariate analysis across features

Model owners need a line of defense to detect and protect against these nasty inputs. One technique to accomplish this is to employ an unsupervised learning method to categorize model inputs and predictions, allowing you to discover cohorts of anomalous examples and predictions.

This is where model observability tools shine in helping you tighten your training, evaluating, deploying, and monitoring loop.

If you are seeing a number of examples that don't fit well into the groups of the more prototypical examples, this could be evidence of some edge cases your model has never seen. In either case these examples may be good candidates for you to include and potentially upsample in your training to shore up these gaps.

Anomalous examples can occur just by chance, but they can also be generated by an adversary who is trying to trick your model. In many business critical applications, especially in the financial sector, model owners have to be hyper-vigilant to monitor for adversarial inputs that are designed to make a model behave in a certain way.

In 2018, Google brought attention to adversarial attacks on machine learned image classification models by demonstrating how a small amount of noise, imperceptible to the human eye, added to an image could cause the model to wildly misclassify.

In applications where every hour counts, the speed in which you can identify a new attack, patch the vulnerability, and redeploy your model may make all the difference in the success of your business.

In summary, there are a number of model failure modes to be on the lookout for, and they rarely affect your model in isolation. To piece together why your model's performance may have degraded or why your model is behaving erratically in particular cases, you must have the proper measurements to deduce what's going on. Model monitoring tools fill this role in the machine learning workflow and empower teams to constantly improve models after they've been shipped into the world.

ML Observability: Best Practices

There are several components that are needed to make a model work successfully: the service that actually renders the model, the data that is flowing in and out of the model—including features and predictions—and the model itself. Each of these components need to be working as expected in order for any model to be successful.

A failure in any of these components—say, concept drift causing performance degradation—can negatively impact the very business results that the model was designed to improve. Therefore, we need to measure all of these to have a complete picture. The sections below break out best practices across each, diving into model performance, data quality, drift, explainability and service health.

Model Performance

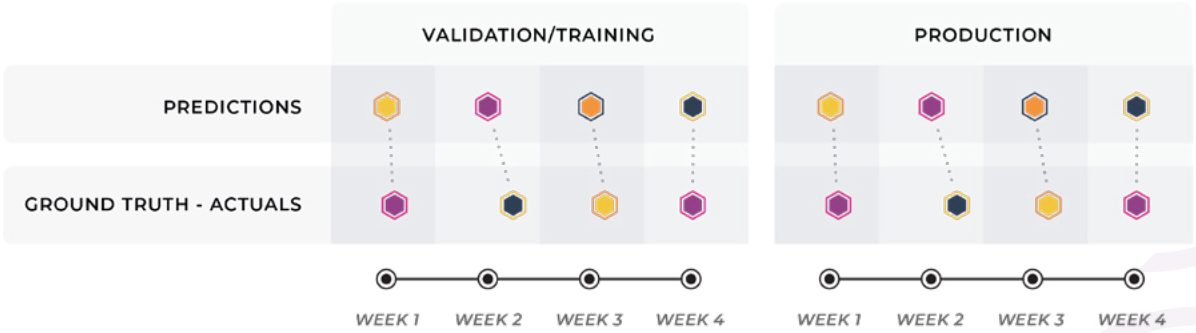
Performance analysis of production models can be complex, and every situation comes with its own set of challenges. Unfortunately, not every model application scenario has an obvious path to measuring performance like the toy problems that are taught in school.

This section will cover a number of challenges connected to availability of ground truth and discuss the performance metrics that are available to measure models in each scenario.

The ideal ML deployment scenario, often what they teach you in the classroom, is when you get fast actionable and fast performance information back on the model as soon as you deploy your model to production.

This ideal view looks something like this:

Ground Truth in Production



In this example ground truth is surfaced to you for every prediction and there is a direct link between predictions and ground truth, allowing you to directly analyze the performance of your model in production.

There are many industries that are lucky enough to face this ideal scenario. This is the case in digital advertising where a model attempts to predict which ad a consumer is most likely to engage with. Almost immediately after the prediction is made, the ground truth, whether they clicked or not, is determined.

For another example of this ideal scenario we can take a look at predicting food delivery estimates. As soon as the Pizza has arrived at the hungry customer's house, you know how well your model did.

Once you have this latent ground truth linked back to your prediction event, model performance metrics can easily be calculated and tracked. The best model metric to use primarily depends on the type of the type of model and the distribution of the data it's predicting over. Here are a few common model performance metrics:

Accuracy: General overall accuracy is a common statistic useful when classes are balanced.

Recall: Useful for unbalanced classes. What fraction of overall positives did I get correct.

Precision: Useful for unbalanced classes. What fraction of positive identifications were correct.

F1: Useful for unbalanced classes and allows for analysis of trade off between Recall and Precision.

MAE or MAPE: Regression or Numeric Metric performance analysis. MAPE can be good when the percent of how far off you are matters.

Once a model metric is determined, tracking this metric on a daily or a weekly cadence allows you to make certain that performance has not degraded drastically from when it was trained or when it was initially promoted to production.

However as you have surely predicted by now, this ideal scenario is the exception and not the rule. In many real world environments access to ground truth can vary greatly, and with it the way the tools you have at your disposal to monitor your models.

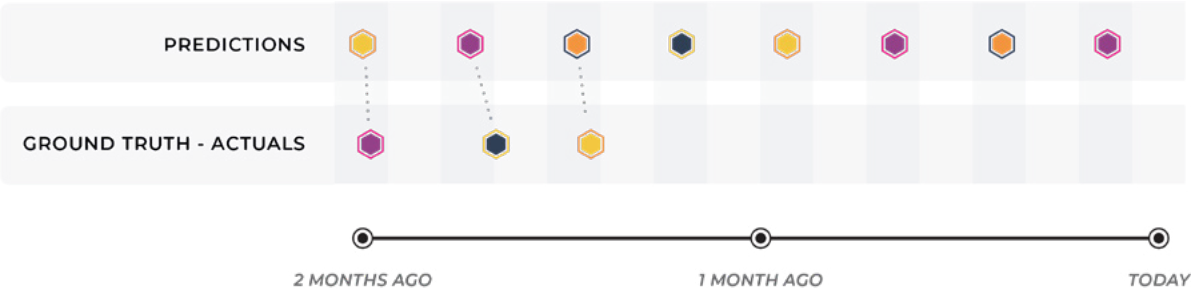
While many applications enjoy realtime ground truth for their model's predictions, many model application scenarios have to wait a while to know how their model should have behaved in production.

Imagine you are trying to predict which of your customers are creditworthy and which ones of them are likely to default on a loan. You likely won't know if you made a good decision until the loan is paid off or the customer defaults. This makes it tricky to ensure that your model is behaving as expected.

This delay in receiving ground truth can also not have a fixed time scale. Take, for example, trying to classify which credit card transactions are fraudulent. You likely won't know if a transaction was truly fraudulent until you get a customer report claiming that their card was stolen. This can happen a couple of days, weeks, or even months after the transaction cleared.

In these cases, and a number of others, the model owner has a significant time horizon for receiving ground truth results for their model's predictions.

Ground Truth - 2 Month Lag



In this above diagram, while we do see that ground truth for the model is eventually determined, the model's predictions over the last month have not received their corresponding outcomes.

When this ground truth delay is small enough, this scenario doesn't differ too substantially from real time ground truth, as there is still a reasonable cadence for the model owner to measure performance metrics and update the model accordingly as one would do in the realtime ground truth scenario.

However, in systems where there is a significant delay in receiving ground truth, teams may need to turn to proxy metrics. Proxy metrics are alternative signals that are correlated with the ground truth that you're trying to approximate. For example, imagine you using a model to which consumers are most likely to default on their credit card debt. A potential proxy metric for success in this

scenario might be the percentage of consumers you have lent credit to that make a late payment.

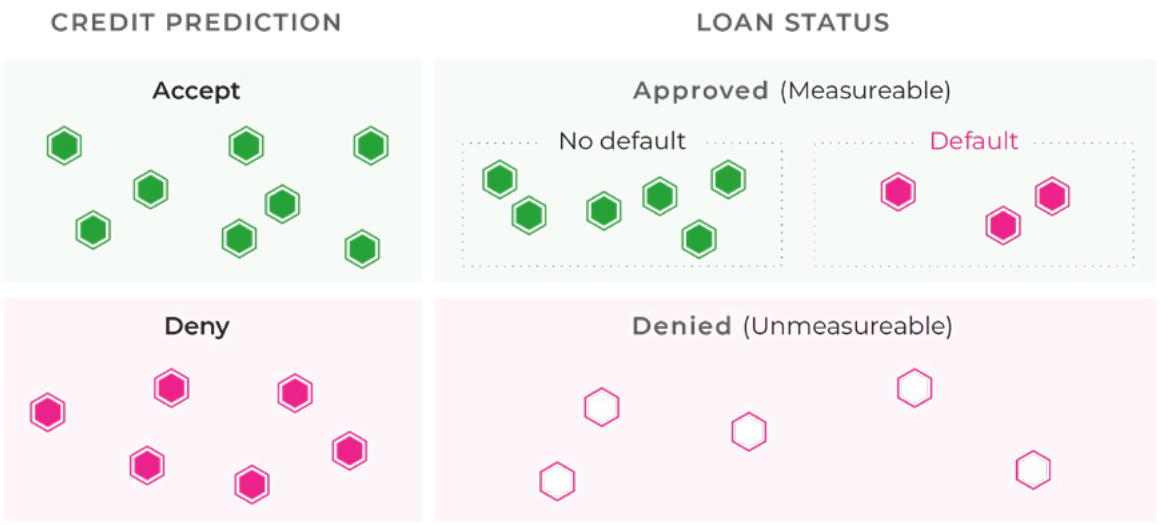
So while you don't have access to ground truth yet, you can start to see how the proxy metrics that you can compute in the meantime change over time to measure how your model is performing.

Proxy metrics serve as a powerful tool in the face of delayed ground truth as they give a more up to date indicator of how your model is performing.

Case 3: Causal Influence on Ground Truth (Biased Ground Truth)

One important thing to note is that not all ground truth is created equal. There are some cases where teams receive real time ground truth; however, the model's decisions substantially affect the outcome.

Prediction of Credit Cohort



In this example, we are trying to predict who is credit worthy enough to receive a loan. This becomes tricky because when you decline someone's credit, you no longer have any information about whether they could have paid you back.

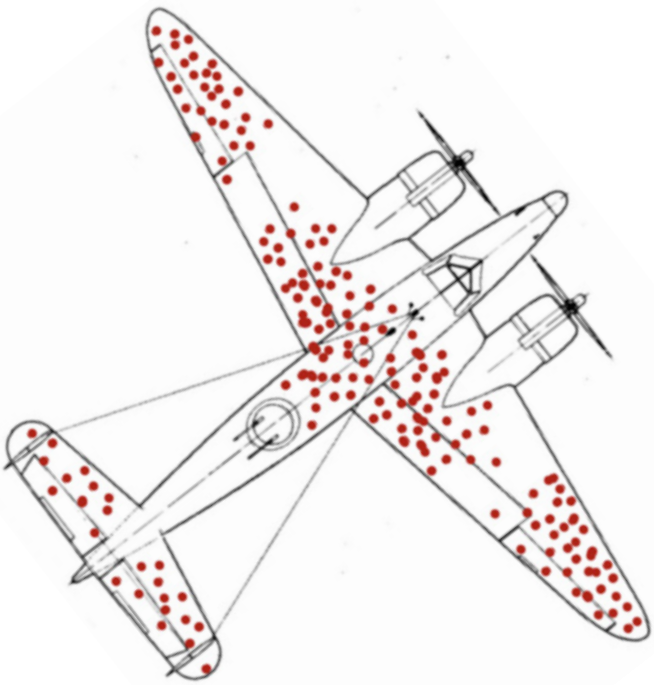
In other words, only the people you decide to give a loan to will result in outcomes that you can use to train future models on. As a result, we will never know whether someone the model predicted will default could have actually paid the loan back in full.

This may lead you to throw your hands in the air and accept that your ground truth is just going to be biased; however, you do have some tools at your disposal.

Something you can do is create a hold-out set where you don't follow your model's predictions and compare the difference in prediction performance between this hold-out set and the set using the model's predictions.

You can use these two sets that received different treatments to look for validate your models predictions and, in our example's case, ensure that you're not potentially missing a potential set of credit worthy people that your model was missing.

As someone working with data, you will most likely see this picture pop up often. No discussion of bias in data is complete without a reference to it.



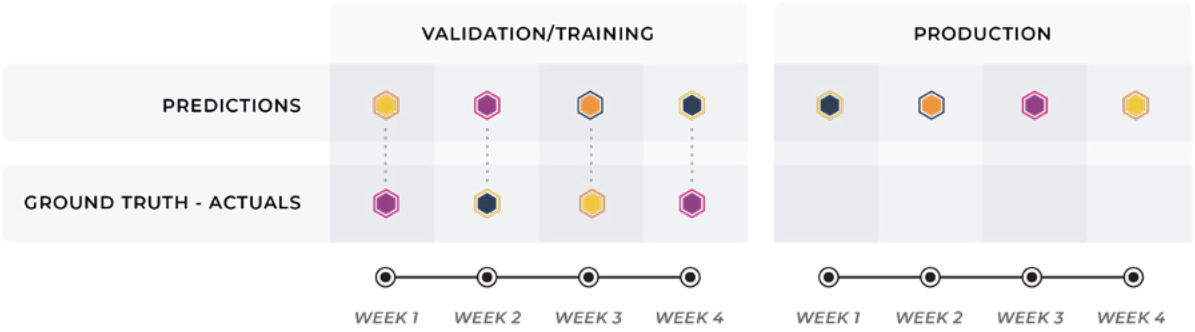
As the story goes, engineers in WW2 created a heatmap of locations where bullet holes pierced the planes that came back from battle with the intention of determining where to fortify the armor. One day a statistician named Ahbrahm Wald noted that the heatmap was only created from planes that had made it back from their missions. So, in fact, the best place to put armor was probably where the empty space existed, such as the engines, as the planes that were hit in these locations never made it home.

In short: always be conscious of the bias in your ground truth data.

Case 4: No Ground Truth

This brings us to the worse case scenario for a modeling team: having no ground truth feedback to connect back to model performance.

No Ground Truth in Production



In the above example we have ground truth in our validation and training sets to link back to our model's predictions; however in production, we have little to no feedback on how our model is performing.

Again in this scenario proxy metrics for ground truth can be extremely useful. In the absence of ground truth, if you can find something else that correlates to ground truth, you can still get a sense of how your model is performing over time.

Outside of proxy ground truth metrics, even when ground truth is hard to collect, it's important for teams to find a way to collect a sample of ground truth data.

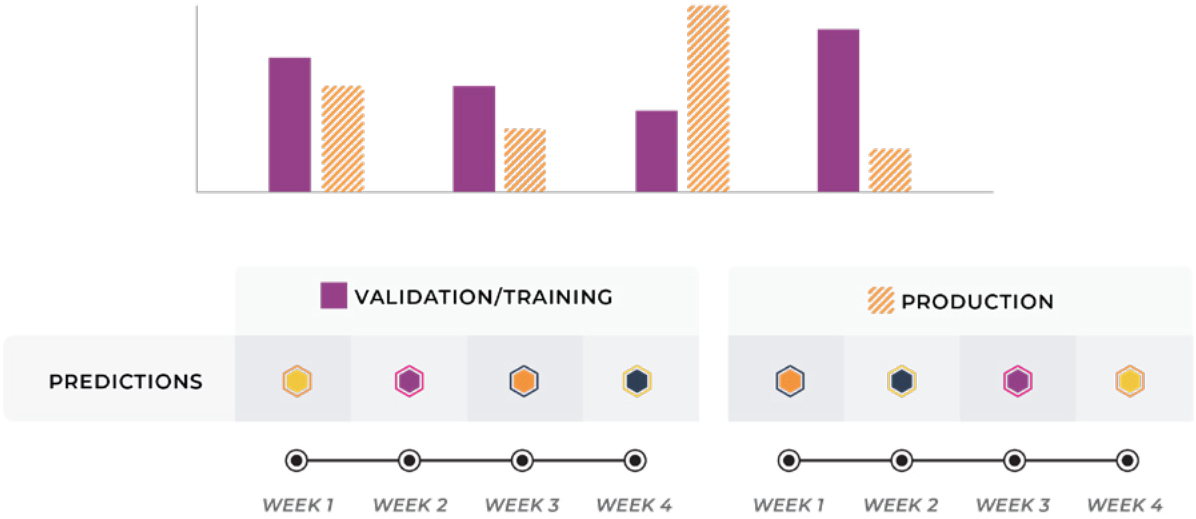
One way to acquire this ground truth data is to hire human annotators or labelers to provide feedback on their model's performance. This approach can be expensive and time consuming; however, the reward for having a set of high-quality ground truth data is immense.

In the periods where ground truth is available or has been collected through manual annotation, performance or lagging performance metrics can be used. Though these lagging performance metrics are not quite as good at signaling a sudden model performance regression in a real time application, they still provide meaningful feedback to ensure that the models performance is moving in the right direction over time.

Drift is a Proxy for Performance

While these lagging performance metrics can't immediately signal a change in a model's performance, measuring the shift in the distribution of prediction outputs potentially can. A drift occurring in the output prediction can be used to alert the team of aberrant model behavior even when no ground truth is present.

No Ground Truth in Production



Some metrics you can use to quantify your prediction drift are distribution distance metrics, such as: Kullback-Leibler Divergence, Population Stability Index (PSI), Jensen-Shannon Divergence, etc. **For a full guide on using statistical distances in machine learning, see Arize's separate white paper on the subject.**

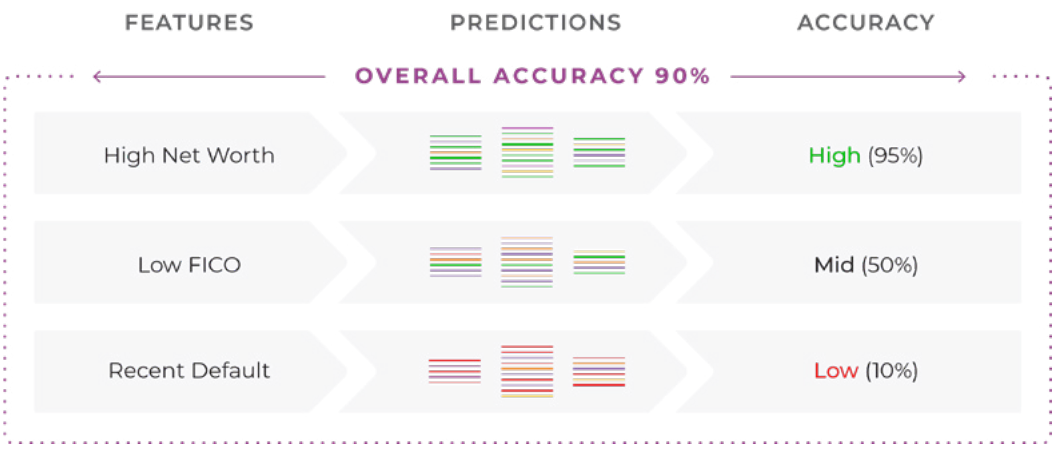
Model Metric by Cohort

As much as data scientists like to deal with aggregate optimization statistics the reality is that models affect different people, customer segments, and business decisions differently.

Two candidate models with the same accuracies could affect particular sets of individuals dramatically differently, and these differences can be very important to your business.

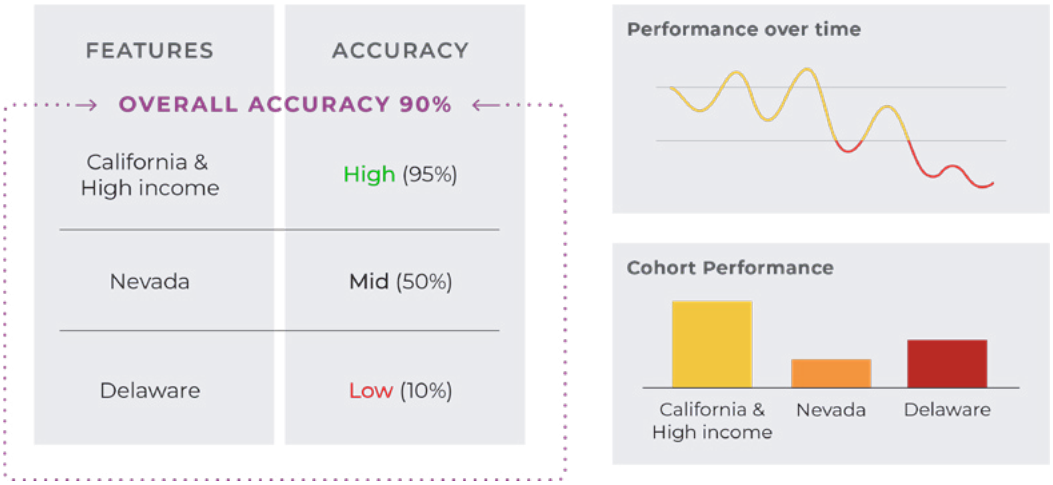
Teams typically divide their data into "slices" or "cohorts". These cohorts can be discovered over time or they are built on the fly to debug where a model is making a disproportionate amount of mistakes.

Cohort Prediction: Facets/ Slices of Groups



In this example above, we see that grouping data based on features such as high net worth, low FICO scores, and the presence of a recent default provides important information about how the model is performing for these particular cohorts.

Above the overall accuracy of 90% is really hiding the fact that the model is performing terribly at predicting credit worthiness for people who have recently defaulted.



Measuring model performance across cohorts is similar to measuring model performance in aggregate. Performance should be measured for each cohort that is important to the business and an alert should be issued when performance drops below a threshold defined by training or initial model launch.

Measuring Business Outcomes

Now that we have talked about measure model performance by selecting the right model metrics for your application scenario, let's briefly talk about measuring business metrics.

Business metrics go hand in hand with the model metrics and when properly defined, they should be linked. At the end of the day, you aren't shipping a F1 score to your customers, and as a result it's important to keep in touch with how your models are affecting how each customer is experiencing your product.

Business metrics tend to not be well suited for traditional optimization, yet they provide key insights into a business goal. Since these metrics are not easy for a model to optimize for, the optimization problem will be set up using a parallel metric.

Turning to the credit worthiness example again, while your model might be optimizing for a traditional model metric such as accuracy or F1, a business metric you might want to be monitoring is the percentage of people you turn away from credit.

At the end of the day a product manager on your team might not care about MAPE, but they will care about how your users are experiencing your product. Measuring model evaluation metrics doesn't usually capture how many angry customers you might have.

As you may have picked up, identifying and measuring business metrics is an extremely important process for ensuring your effort is being well spent in improving your product.

In summary, measuring model performance is not one size fits all, and your business application may require some or all of these measurement techniques that we discussed.

While the path toward measuring your model's performance is not always clear cut, what is clear is that correctly measuring your model's performance is essential to ensuring you are shipping a consistent and effective product to your customers.

Explainability

As models have increased in complexity, the ability to introspect and understand why a model made a particular prediction has become more and more difficult. As a result, it has become even more important that we have the ability to explain how they make their predictions.
In this section, we will lay out the different levels of ML explainability, and how each of these can be used across the ML Lifecycle. Lastly we will cover some common methods that are being used to obtain these levels of explainability.

Explainability in Training

To start, let's cover how explainability can be used to help during the model training phase of the machine learning life-cycle. In particular, let's begin by starting to tease apart the different flavors of ML explainability.

Of particular importance in training is **global explainability**. We consider an ML engineer to have access to global model explainability if across all predictions they are able to attribute which features contributed the most to the model's decisions. The key term here is "all predictions," unlike cohort or local explainability—global is an average across all predictions.

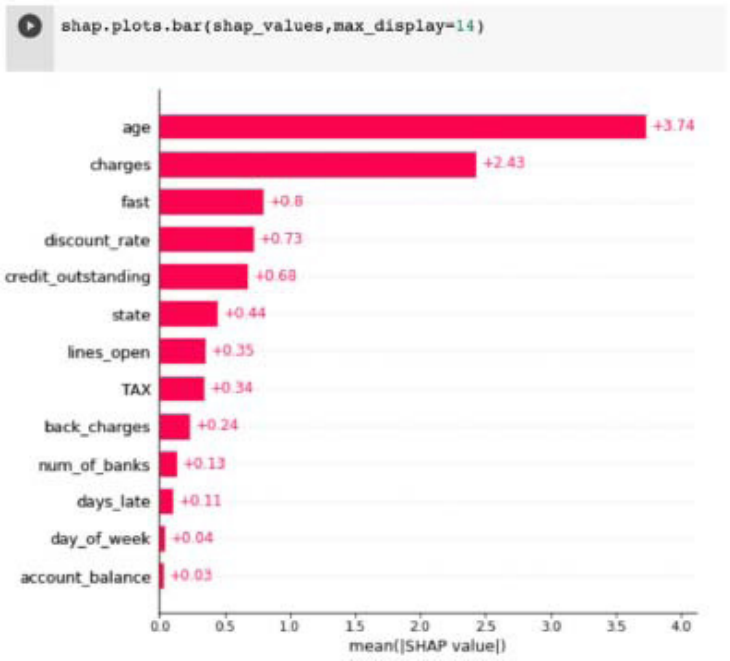
In other words, global explainability lets the model owner determine to what extent each feature contributes to how the model makes its predictions over all of the data.

In practice, one area global explainability is used is on an ad hoc basis to provide information to non-data science teams about what the model, on average, uses to make decisions. Example: marketing teams, looking at a churn model, might want to know what are the most important features to predict customer churn.

In this use case, the model provides global insights to stakeholders outside of data science, and how often it is needed and for whom can vary—it's more of an analytical function to provide business stakeholders more insights.

There is also a model builder use case for global explainability, namely ensuring a model is doing what you expect and hasn't changed version to version. There may be unforeseen ways your model can "learn" the function you set out to teach it, and without the ability to understand what features it's relying on it's hard to say which function it chose to approximate.

Let's take a model that predicts the credit limit that a new customer should get for your bank's new credit card. There are a number of ways that this model can accomplish its job of assigning a credit limit to this new customer, but not all are created equal. Explainability tools can help build confidence that your model is learning a function that isn't over-indexing on particular features that may cause problems in production.



For example, let's say a model is relying extremely heavily on age to make its prediction of what credit limit to assign. While this may be a strong predictor for ability to repay large credit bills, it may under-predict credit for some younger customers who have an ability to support a larger credit limit, or over-predict for some older customers who perhaps no longer have the income to support a high credit limit.

Explainability tools can help you find potential issues like this before shipping a model to production. Global explainability helps spot check how features in the model are contributing to the overall predictions of the model. In this case, global explainability would be able to highlight age as a primary feature that the model is relying on, and allow the model owner to understand how this might impact cohorts of their users and take action.

During training, global explainability is instrumental in gaining confidence in the features you choose to provide your model for its predictions. Often, a model builder will add a large number of features and observe a positive shift in a key model performance metric. Understanding which features or interactions between features drove this improvement can help you build a leaner, quicker, and even a more generalizable model.

Explainability in Validation

Onto the next phase of the ML lifecycle: model validation. Let's take a look at how explainability can help model builders validate models before shipping them to production.

To this end, it's important to define our next class of model explainability: **Cohort Explainability**. Sometimes you need to understand how a model is making its decisions for a particular subset of your data, also known as a cohort. Cohort explainability is the process of understanding to what degree your model's features contribute to its predictions over a subset of your data.

Cohort explainability can be incredibly useful for a model owner to help explain why a model is not performing as well for a particular subset of its inputs. It can help discover bias in your model and help you uncover places where you might need to shore up your datasets.

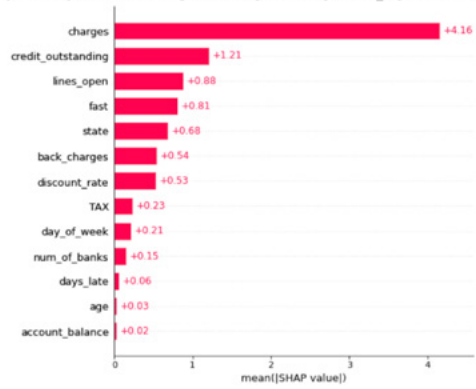
Taking a step back, during validation, our primary objective is to see how our model would hold up in a serving context. To be more specific, validation is about probing how well your model has generalized from the data that it was exposed to in training.

A key component in assessing generalization performance is uncovering subsets of your data where your model may be underperforming, or relying on information that might be too specific to the data that it was trained on, also commonly known as overfitting.

```
X_age = X[X.age < 30]
y_age = y[X_age.index.to_numpy()]
model = xgboost.XGBRegressor().fit(X_age, y_age)

# explain the model's predictions using SHAP
# (same syntax works for LightGBM, CatBoost, scikit-learn, transformers, Spark, etc.)
explainer = shap.Explainer(model)
shap_values = explainer(X_age)
shap.plots.bar(shap_values,max_display=14)
```

```
[14:13:26] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now
```



Cohort explainability can serve as a helpful tool in this model validation process by helping to explain the differences in how a model is predicting between a cohort where the model is performing well versus a cohort where the model is performing poorly.

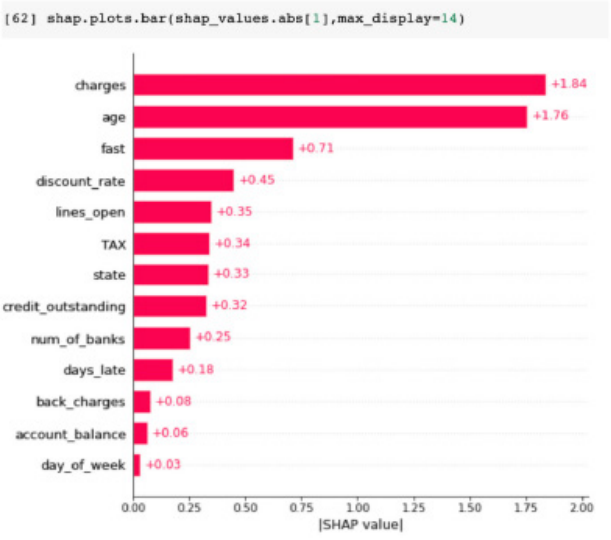
For example, let's say that you have a model for predicting credit scores for a particular user and you want to know how the model is making its prediction for your users that are below the age of 30. It's worth noting the most important feature for this group, "charges," is different from the global feature importance. Cohort explainability can help investigate how your model is treating these users in comparison to some other cohort of your data, such as users above the age of 50. This can help you uncover somewhat unexpected or undesirable relationships between input features and model outcome.

Another important benefit that cohort explainability buys you is the ability to understand when to split your model into a collection of models and federate their predictions. It's very possible that the function you are trying to learn is best handled by a federation of models, and cohort explainability can help you discover the cohorts for which a different feature set or model architecture might make more sense.

Explainability in Production

Lastly, let's take a look at how explainability can help a model owner in the production context. Once the model has left the research lab and has been served into production, the needs of a model owner change a bit.

In production, model owners need to be able to answer all-of-the-above Cohort and Global explainability, in addition to questions about very specific examples to help with customer support, enable model auditability, or even just provide feedback to the user about what happened.



This leads us into our last class of ML explainability: Individual, also known as **Local Explainability**. This is somewhat self explanatory, but local explainability helps answer the question, "for this particular example, why did the model make this particular decision?"

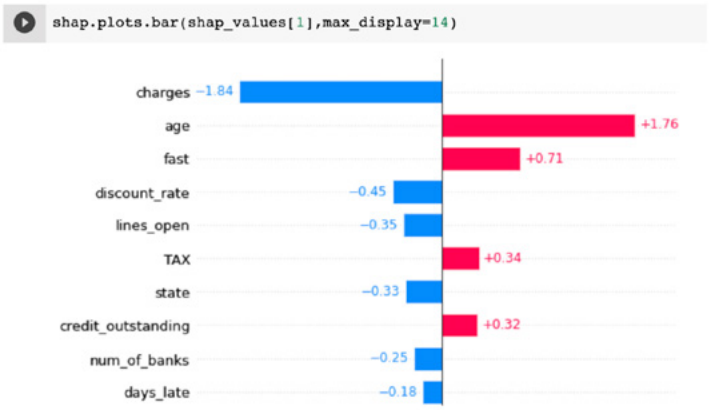
The level of specificity is an incredibly useful tool in the toolbox for an ML engineer, but it's important to note that having local explainability in your system does not imply that you have access to global and cohort explainability.

Local explainability is indispensable for getting to the root cause of a particular issue in production. Imagine you just saw that your model has rejected an applicant for a loan and you need to know why this decision was made. Local explainability would help you get to the bottom of which features were most impactful in making this loan rejection.

Especially in regulated industries, such as lending in our previous example, local explainability is paramount. Companies in regulated industries need to be able to justify decisions made by their model and prove that the decision was not made using features or derivatives of protected class information.

Note: In all our examples we used the absolute values of Shap for visualization; these hide the directional information of the individual features.

The prediction-level shap with directional information can illustrate how the feature influences an outcome—in this case, "charges" decreases the probability of default where "age" increases.



As more and more regulation impacts technology, companies are not going to be able to hide behind ML black boxes to make decisions in their products. If they would like to reap the benefits of machine learning, they are going to have to invest in solutions that provide local explainability.

When the model is served and making predictions for your users, individual inference level explainability can act as a fantastic entry point into understanding the dynamics of your model. As soon as you see a model prediction that may have gone awry, either from a bug report or monitoring thresholds that you set up, this can start an investigation into why this happened.

It may lead you to discover which features contributed most to the decision, and even cause you to dive deeper into a cohort that may also be affected by this relationship you uncovered. This can help close the ML lifecycle loop, allowing you to go from "I see a problem with an individual example" to "I have discovered a way to improve my model."

Common Methods In ML Explainability:

Now that we have defined some broad classes of ML explainability and how they might be used in different stages of the ML Lifecycle, let's turn our attention toward a few techniques that are powering ML explainability solutions today.

SHAP – SHapley Additive exPlanations

To start, let's take a look at SHAP which stands for SHapley Additive exPlanations. SHAP is an explainability technique that developed from concepts in cooperative game theory. SHAP attempts to explain why a particular example differs from the global expectation from a model.

For each feature in your model, a Shapley value is computed which explains how this feature contributed to the difference between the model's prediction for this example as compared to the "Average" or expected model prediction.

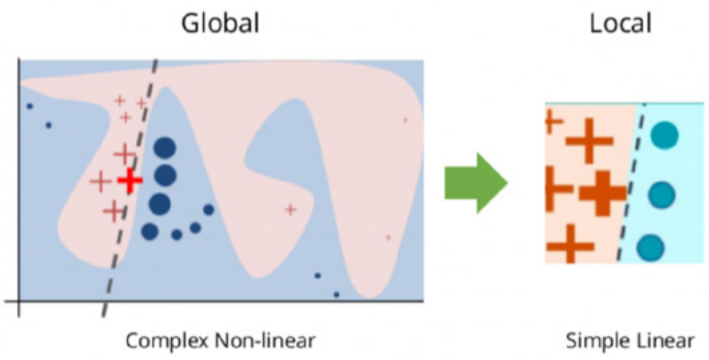
The SHAP values of all the input features will always sum up to the difference between the observed model output for this example and the baseline (expected) model output, hence the additive part of the name in SH**A**P.

SHAP can also provide global explainability using the Shapley values computed for each data point, and even allows you to condition over a particular cohort to gain some insight on how feature contributions differ between cohorts.

Using SHAP in the model-agnostic context is simple enough for local explainability, though global and cohort computations can be costly without particular assumptions about your model.

LIME – Local Interpretable Model-Agnostic

LIME, or Local Interpretable Model-Agnostic Explanations, is an explainability method that attempts to provide local ML explainability. At a high level, LIME attempts to understand how perturbations in a model's inputs affect the end-prediction of the model. Since it makes no assumptions about how the model reaches the prediction, it can be used with any model architecture, hence the "model-agnostic" part of LI**M**E.



LIME attempts to understand this relationship between a particular example's features and the model's prediction by training a more explainable model such as a linear model with examples derived from small changes to the original input.

At the end of training, the explanation can be found from the features for which the linear model learned coefficients above particular thresholds (after accounting for some normalization). The intuition behind this is that the linear model found these features to be most important in explaining the model's prediction, and therefore for this local example you can reason about the contributions that each feature made in explaining the prediction that your model made.

Implications

One of the most common critiques of modern machine learning is the absence of explainability tools to build confidence in, provide auditibability for, and enable continuous improvement of machine learned models. Today, there is a keen interest in surmounting this next hurdle. As evidenced by some of the modern explainability techniques covered, we are well on our way.

Data Quality

In practice today, a model is often only as good as the data it is trained on. Data quality doesn't stop being important after the model is trained, but continues to remain important as the model is deployed in production. The quality of the model's predictions is highly dependent on the quality of the data sources powering the model's features.

What Do We Mean by Data Quality?

Data quality is a broad term and can cover a wide variety of issues in your data. For the purposes of this section, data quality refers to hard failures in data pipelines. Other important issues around data quality—such as "slow bleed" failures or gradual drift in your data over time—will be covered in a subsequent section titled "What Can Go Wrong: Model Failure Modes."

To dig deeper, it's necessary to define and break out categorical data streams versus numerical data streams.

Categorical Data

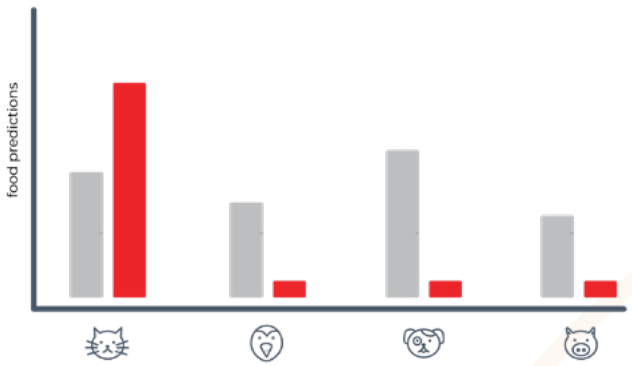
Categorical data is just what it sounds like: a stream of categories, like the type of pet someone owns (dog, cat, bird, pig, etc.).

Cardinality Shifts

To start, something that can go wrong with a categorical data stream is a sudden shift in the distribution of categories. To take it to an extreme, let's say your hypothetical model predicting which pet food to buy for your pet supply store starts seeing data saying that people only own cats now. This might cause your model to only purchase cat food, and all your potential customers with dogs will have to go to the pet supply store down the street instead.

Data Type Mismatch

In addition to a sudden cardinality shift in your categorical data, your data stream might start returning values that are not valid to the category. This is, quite simply, a bug in your data stream, and a violation of the contract you have set up between the data and the model. This could happen for a variety of reasons: your data source being unreliable, your data



processing code going awry, some downstream schema change, etc. At this point, whatever comes out of your model is undefined behavior, and you need to make sure to protect yourself against type mismatches like this in categorical data streams.

Missing Data

One incredibly common scenario that practitioners run into is the problem of missing data. With the rising number of data streams used to compute large feature vectors for modern ML models, the likelihood that some of these values will be nil is higher than ever. So what can you do about it?

One thing you certainly can do is throw your hands up in the air and discard the row in a training context, or throw an error in your application in a production context. While this will help you avoid this problem, it's possibly not the most practical. If you have hundreds, thousands, or tens of thousands of data streams used to compute one feature vector for your model, the chance that one of these streams is missing can be very high!

This brings us next to how you might fill this missing value, commonly referred to as imputation. For categorical data, you could choose the most common category that you have historically seen in your data, or you could use the values that are present to predict what this missing value likely is.

Numerical Data

A numerical data stream is also pretty self-explanatory. Numerical data is data that is represented by numbers, such as the amount of money in your bank account, or the temperature outside in Fahrenheit or Celsius.

Out of Range Violations

To start things off, something that can go wrong with numerical data streams is out of range violations. For example, if age was an input to the model and you are expecting the age to be between 0–120, but suddenly receive a value in the 300s, this would be considered out of range.

Type Mismatch

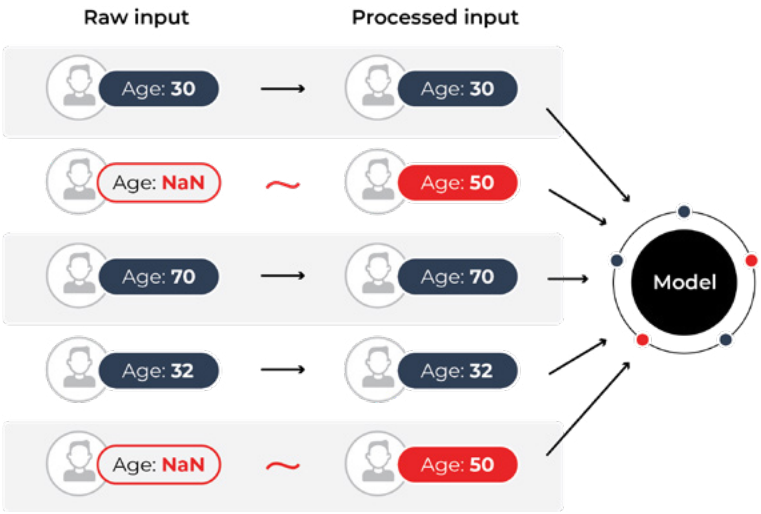
Type mismatch can also affect numerical data. It's in the realm of possibility that for a particular data stream where you are expecting a temperature reading that you are returned a categorical data point, and you have to handle this appropriately. It's possible that the default behavior may be to cast this categorical value to a number that, although now valid, has entirely lost its semantic meaning and is now an error in your data that is incredibly hard to track down.

Missing Data

For numerical data, you have a few more options for imputations, such as taking the average, median, or some other distribution metric for this particular value. The complexity of your solution to this problem is entirely up to your application scenario, but it's important to know that no solution is perfect here.

Challenges with Monitoring Data Quality Today

Now that we have gotten a better idea of what possible data quality issues you may run into, let's now briefly dive into some common challenges that practitioners run into when attempting to keep tabs on the quality of their data.



Before we start here, it's important to note that this is different from the broad product space of data observability. Data observability tools are mostly focused on monitoring the quality of tables and data warehouses, while ML Observability is focused on monitoring the inputs and outputs of models. These models are consistency evolving, features are being added and changed, and so the data quality monitoring of models must be able to evolve with the schema of the model.

Too Much Data to Keep Tabs on

It's not surprising to many current ML practitioners that many models these days rely on tons of features to perform their tasks. One rule of thumb, guided by recent advances in statistical learning theory, suggests that a model can effectively learn approximately a feature for every 100 examples you have in a training set. With training set sizes exploding into the hundreds of millions and even billions, models with feature vector lengths in the tens and hundreds of thousands are not uncommon.

This leads us to a major challenge that practitioners face today. To support these incredibly large feature vectors, teams have poured larger and larger data streams into feature generation. Writing code to monitor the quality of each of these data streams is fundamentally untenable, and the reality is that this data schema will inevitably change often as the team experiments to improve the model.

At the end of the day, no one wants to sit there and hand configure thresholds, baselines and set up a custom data monitoring system for each of these data streams that are feeding into the model. It's common to add a feature, drop a feature, change how it is computed, and adding more work into the ML development loop will only slow you and your team down.

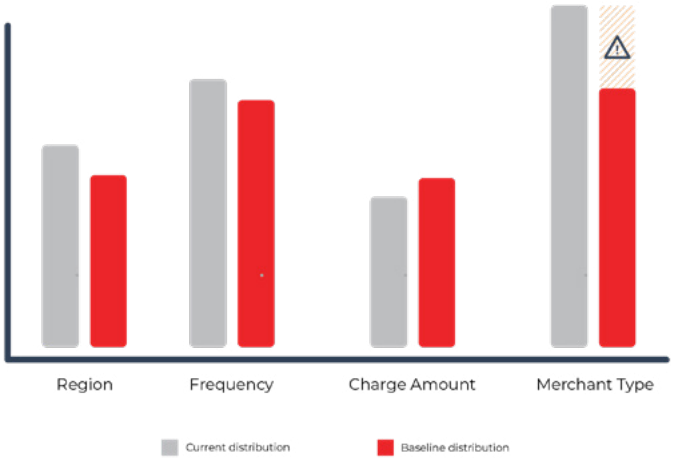
What now?

Now that we understand some of the current challenges around monitoring and fixing data quality issues, what can we do about it? To start, teams need to start keeping track of how the quality of their data affects the end performance of their model.

Leverage Historical Information

Ultimately the model's performance is what we care about, and it's very possible that the quality of some data is worth more than that of others. To avoid manually creating baselines and thresholds for each data stream, teams need to have a history of data to look at either from training sets or from historical production data.

Once these historical distributions have been determined, your monitoring system can have a better idea about what it should consider an outlier in a numerical stream, and generate alerts when a categorical stream has strongly deviated from its historical distribution. From these distributions, intelligent baselines and thresholds can be created to balance how "noisy" or likely to fire these alerts are, giving power to the model team to balance risk vs reward.



On top of setting up automatic alerting systems for all of your data streams, your data quality monitoring system should also allow you to enforce type checks to protect against downstream errors in your model and avoid potential typecasting issues.

Hill Climb using Model Performance

Lastly, by keeping track of your model's end performance, your monitoring system should also allow you to test out different imputation methods for your data and give you the performance impact for this new imputation strategy. This provides confidence that the choices you are making are positively impacting the end performance of the model.

As fast as machine learning has progressed and made its way into some of our most crucial products and services, the tooling to support these experiences has lagged behind. These core features of a modern data quality monitoring system bring back control to the ML engineer and remove a large amount of guesswork, which unfortunately has crept very deeply in the art of productionizing machine learning.

Model and Feature Drift[2]

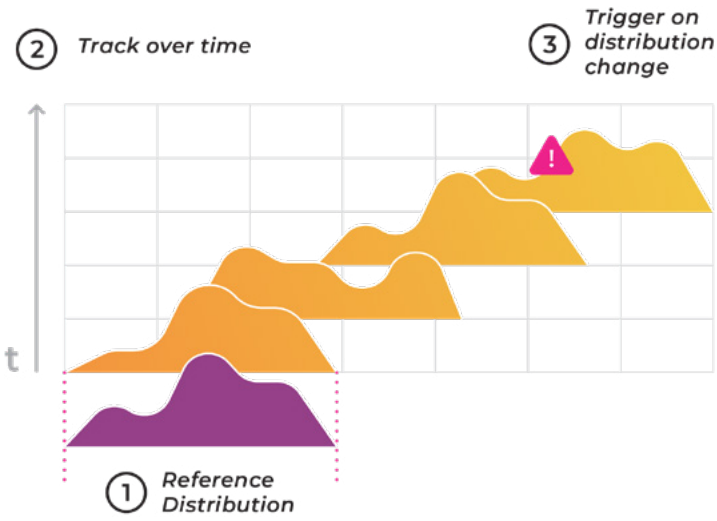
As an ML practitioner, you probably have heard of drift. In this piece, we will dive into what drift is, why it's important to keep track of, and how to troubleshoot and resolve the underlying issue when drift occurs.

What Is Drift?

First things first, what is drift? **Drift** is a change in distribution over time. It can be measured for model inputs, outputs, and actuals. Drift can occur because your models have grown stale, bad data is flowing into your model, or even because of adversarial inputs.

Now that we know what drift is, how can we keep track of it? Essentially, tracking drift in your models amounts to keeping tabs on what had changed between your reference distribution, like when you were training your model, and your current distribution (production).

Models are not static. They are highly dependent on the data they are trained on. Especially in hyper-growth businesses where data is constantly evolving, accounting for drift is important to ensure your models stay relevant.



Change in the input to the model is almost inevitable, and your model can't always handle this change gracefully. Some models are resilient to minor changes in input distributions; however, as these distributions stray far from what the model saw in training, performance on the task at hand will suffer. This kind of drift is known as **feature drift** or **data drift**.

It would be amazing if the only things that could change were the inputs to your model, but unfortunately, that's not the case. Assuming your model is deterministic, and nothing in your feature pipelines has changed, it should give the same results if it sees the same inputs.

---

[2] NOTE: this section is adapted from a previous article written in collaboration with *Hua Ai, Data Science Manager at Delta Air Lines.*

While this is reassuring, what would happen if the distribution of the correct answers, the actuals, change? Even if your model is making the same predictions as yesterday, it can make mistakes today! This drift in actuals can cause a regression in your model's performance and is commonly referred to as **concept drift** or **model drift**.

How Do I Measure Drift?

As we talked about previously, we measure drift by comparing the distributions of the inputs, outputs, and actuals between training and production.

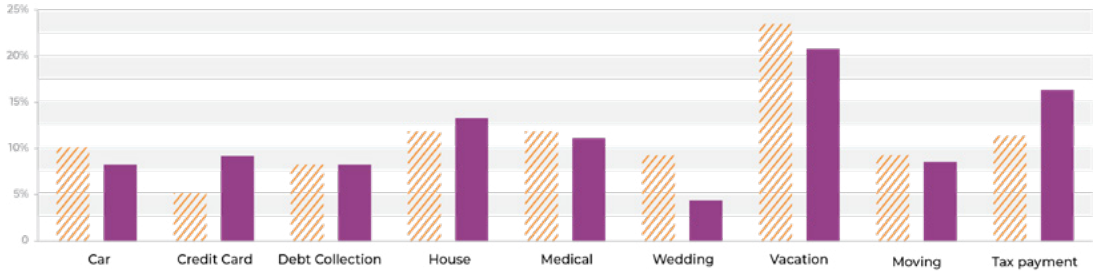
But how do you actually quantify the distance between these distributions? For that, we have distribution distance measures. To name a few, we have

1. Population Stability Index (PSI)
2. Kullback—Leibler divergence (KL divergence)
3. Wasserstein's Distance

While each of these distribution distance measures differs in how they compute distance, they fundamentally provide a way to quantify how different two statistical distributions are.

This is useful because you can't build a drift monitoring system by looking at squiggles on charts. It would be best if you had an objective, quantifiable ways of measuring how the distribution of your inputs, outputs, and actuals are changing over time.

PSI Calc



	Reference (A)	Actual (B)	A - B	Ln (A/B)	PSI
Car	10	8	2.00%	0.223	0.004
Credit Card	5	9	-4.00%	-0.588	0.024
Debt Collection	8	8	0.00%	0.000	0.000
House	12	13	-1.00%	-0.080	0.001
Medical	12	11	1.00%	0.087	0.001
Wedding	9	4	5.00%	0.811	0.041
Vacation	23	21	2.00%	0.091	0.002
Moving	9	8	1.00%	0.118	0.001
Tax Payment	12	18	-6.00%	-0.405	0.024
Total					0.098

For example, in the above figure, we see a comparison in the distributions of how I spent my money this last year as compared to the year prior. The Y-axis represents the percentage of the total money I spent in each category, as denoted on the x-axis. To see if my allocation of money has changed significantly over the last year, we can calculate the population stability index (PSI) between these two distributions.

For each category in the budget, we calculate the difference in percentage between the reference distribution A (my budget last year) and the actual distribution B (my budget this year) and multiply this by the natural log of (A %/ B%). For each of these categories, take the sum of this value, which gives us our PSI.

For example, in the above figure, we see a comparison in the distributions of how I spent my money this last year as compared to the year prior. The Y-axis represents the percentage of the total money I spent in each category, as denoted on the x-axis. To see if my allocation of money has changed significantly over the last year, we can calculate the population stability index (PSI) between these two distributions.

For each category in the budget, we calculate the difference in percentage between the reference distribution A (my budget last year) and the actual distribution B (my budget this year) and multiply this by the natural log of (A %/ B%). For each of these categories, take the sum of this value, which gives us our PSI.

The larger the PSI, the less similar your distributions are, which allows you to set up thresholding alerts on the drift in your distributions.

Regardless of the distribution distance metric you are using, it's important to not just measure the drift in your distributions but also to measure how these distance metrics relate to important business KPIs and metrics. By doing so, you can start to understand how the drift can actually impact your customers and help you understand what drift thresholds trigger alerts to your team.

Can I Retrain My Model?

What should we do when we notice a trained model has drifted? A first thought could be, “let’s retrain it”! While retraining is usually necessary, how to retrain requires some more thoughts. Simply adding the most recent data into your training set and redeploying the same model architecture may not solve the problem.

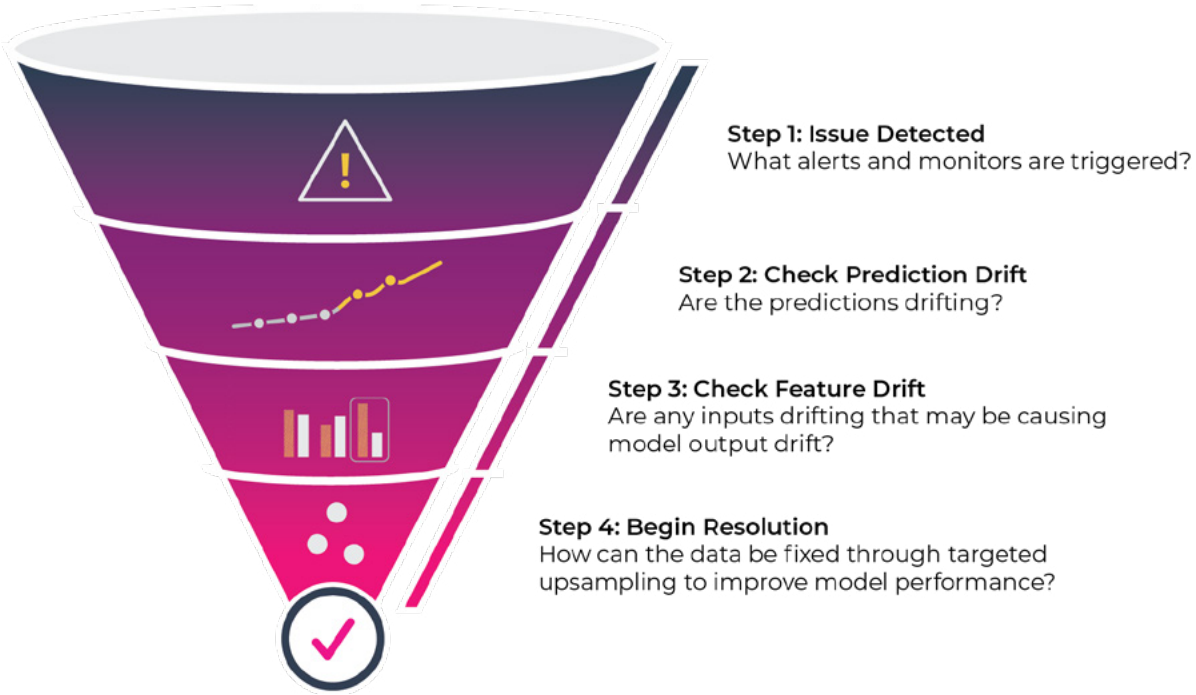
To start, you have to be careful about how you sample this newer data and represent it in your model. If you add too much new data and cause an overrepresentation in your training set, you risk overfitting this newer data. By doing so, your model might not generalize well in the future, and it may impact its performance on inputs that it previously had no trouble with.

On the other hand, if you only add a few examples, your model likely won’t change much at all, and your model might still make the mistakes that you set out to resolve.

You can adjust this tradeoff by weighting the examples in your loss function to strike the right balance between these two competing forces. One way you might measure how to balance this tradeoff is by measuring your performance on one globally sampled hold-out set to approximate your generalization performance and on another hold-out set sampled just from the population of the newer data. If you are performing really well on the global hold outset but not the newer data, you can try upping the weight you assign to the new data in your training set and vice-versa.

While, in many cases, retraining your model is the right solution, some changes are so fundamental that a simple retrain won’t solve anything. If you cannot achieve an acceptable validation performance on your retrained model, it may be time to go back to the drawing board. If something in your business has fundamentally changed, your models may need to as well.

How to Troubleshoot Drift



It all starts with an alert—an email or a notification that something is off. Usually, it is a good practice to keep track of the change in performance and changes in the input data since those changes may provide answers to the change in performance. It is also important to understand where drift has happened on a specific slice of data or certain dates. That will help to diagnose the model and to come up with solutions.

A systematic evaluation is needed when a significant drift alert has been triggered for a period of time. It's usually an art instead of science to decide how significant a drift becomes concerning since it depends on how the predictions are being used and the business value of the prediction. But here are some steps to get you started in resolving drift.

1. Repull Training Data

Identify what input features or outcome variables have drifted and understand how their distributions have changed. Carefully consider what time period should be included in the retraining. Resampling or weighting observations can be used to reconstruct a more balanced training data set.

2. Feature Engineering

Sometimes we'll notice some features have drifted significantly but have not caused model performance issues. That is not something to be overlooked since that indicates the relationship between those features and the outcome variable. Re-construct and select features to adapt to the new dynamics in this data set.

It is also a good time to connect with the end-users of the model to understand if their business processes have changed. New features are often needed to capture the change.

3. Model Structure

Sometimes, the model structure should be revisited as well. For example, if only a slice of predictions has been impacted, a hierarchical model can be helpful to address this without changing the entire model.

How To Get Ahead of This?

Of course, doing all of these amidst a fire drill or crisis amplifies an already stressful situation. Imagine: you get a call from your business partners asking you to explain why the model significantly underperforms in one day since they need to explain an undesirable business outcome to their stakeholders. It is never easy to explain a model's performance on a set of specific data points, especially in pressured situations. It can also cause people to make nearsighted decisions and over-adjust the model to catch the most recent trend.

A good practice is to set up a cadence to review model performance periodically instead of relying entirely on alarms to indicate when things have gone wrong. Regular reviews help to keep track of changing business dynamics and of thinking about model adjustments proactively. Also, it is important to set up a regular channel to communicate with end-users, hear about their feedback on the model, or learn about upcoming process changes. At the end of the day, the models are there to support the end-users. Therefore, user perceptions are equally important as model performance metrics.

ML Service Health & Reliability[3]

One particular challenge that ML practitioners face when deploying models into production environments is ensuring a reliable experience for their users. Just imagine, it's 3 am and you awake to a frantic phone call. You hop into a meeting and the CTO is on the line, asking questions. The number of purchases has suddenly plummeted in the newly launched market, resulting in a massive loss of revenue every minute. Social media has suddenly filled with an explosion of unsavory user reports. The clock is ticking. Your team is scrambling, but it's unclear where to even start. Did a model start to fail in production? As the industry attempts to turn machine learning into an engineering practice, we need to start talking about solving this ML reliability problem.

An important part of engineering is ensuring reliability in our products, and those that incorporate machine learning should be no exception. At the end of the day, your users aren't going to give you a pass because you are using the latest and greatest machine learning models in your product. They are simply going to expect things to work.

To frame our discussion about reliability in ML, let's first take a look at what the field of software engineering has learned about shipping reliable software.

Reliability in Software

The Why

Virtually any modern technological enterprise needs a robust Reliability Engineering program. The scope and shape of such a program will depend on the nature of the business, and the choices will involve the trade-offs around complexity, velocity, cost, etc.

A particularly important trade-off is between velocity ("moving fast") and reliability ("not breaking things"). Some domains, such as fraud detection, require both.

Adding ML into the mix makes things even more interesting.

Consider setting the goal of 99.95% availability. This gives us an outage budget of 5 minutes per week. The vast majority of outages (well over 90% in our experience) are triggered by human-introduced changes to code and/or configuration. This now also increasingly includes changes to production ML models and data pipelines.

[3] This section is adapted from an earlier piece written in collaboration with Bob Nugman, ML Engineer at Doordash.

It is common to have changes to production systems' code and configuration to occur nearly continuously, with each change having the potential for creating an outage-inducing incident. Similarly, with increased reliance on ML, there's an increasing appetite for high-velocity production delivery of ML systems, again with a risk of making a change that introduces a regression or an outage.

Allowing ourselves just one incident per week, the challenge then becomes to detect *and fully mitigate* an incident within five minutes, if we are to meet this goal. How?

There needs to be a systematic Reliability Program.

The Three Pillars of Reliability

A successful reliability program will have the following elements. Each will be covered in more detail below.

- **Observability**: Capability to detect, explore, and make sense of the regressions.
- **Management of Change**: Tooling and practices to ensure that every change introduced (code, configuration, business rules, infrastructure, etc) is discoverable, observable, rolled out gradually, mitigable, revertible.

- **Incident Response**: When (not if) an incident occurs, a pre-existing plan and capability is in place, to first mitigate and then revert the impact of the incident. The process of incident response includes the initiation of the post-incident phase, including blameless post mortems, the findings of which feedback into improvement of all of the three pillars.

These three pillars exert pressure on the entire engineering process, technological stack, as well as the organization's culture.

Let us explore the goals and some of the properties of each.

Observability

A successful observability solution will enable us to:

- Detect a regression quickly;
- Inform a path to rapid and efficient mitigation; and
- Once the issue has been mitigated, inform the causes of the issue, so that the problem can be fully analyzed, understood, and addressed, usually through the post mortem process.

To be efficient, observability tools and practices need to be standardized across the org, while enabling the flexibility to meet the needs of every team. An observability team should formulate best practices and implement tools to enable developers to meet their observability needs, consistently and with minimum effort.

Management of Change

As noted above, most outages are triggered by one of the many changes to code and configuration. The goal of a Management of Change system is to ensure the changes are introduced in a centralized, systematic fashion which supports our reliability goals.

Similar to Observability, management of change (code, configuration, infra, ML models, etc) should be standardized across the org, while accommodating varying needs between teams. This is best achieved with a dedicated owner(s) for the management of change tooling and practices.

For additional reference, here is an example of well-constructed management-of-change capability.

Incident Response

Despite our best preparation, truly unimaginable things will happen. At that time, a response should not look like an engineering activity with brainstorming, problem-solving, etc. It should look like an incident response, with a predefined structure, rehearsed roles, sharp specialized tools, and a mandate to operate.

Importantly, the other two pillars, Observability and Management of Change, are crucial for mounting a successful Incident Response capability.

Reliability in Machine Learning

Now that we have taken a look into what reliability means in the broad world of software engineering, let's take our learnings to understand what problems the field of ML Ops needs to solve to help companies deploy reliable applications with machine-learned components.

To do so, let's turn back our story about the late-night call from your CTO that we discussed a bit earlier. To give some more context, let's say that the model that ranks your search results for your e-commerce company is returning strange results and is severely impacting customer conversions. Let's take what happens here step by step.

The first step in the response to the problem has happened even before you got invited to the call with your CTO. The problem has been discovered and the relevant people have been alerted. This is likely the result of a metric monitoring system that is responsible for ensuring important business metrics don't go off track.

Next, using your ML observability tooling, you are able to determine that the problem is happening in your search model since the proportion of users who are engaging with your top n-links returned has dropped significantly.

After learning this you rely on your model management system to either roll back to your previous search ranking model or deploy a naive model that can hold you over in the interim. This mitigation is what stops your company from losing (as much) money every minute since every second counts for users being served incorrect products.

Now that things are somewhat working again, you need to look back to your model observability tools to understand what happened with your model. There are a number of things that could have gone wrong here, some of which could inform your mitigation strategy, so it's important to quickly start understanding what went wrong.

Lastly, once you have discovered the root cause of the issue, you have to come up with a solution to it, ranging from fixing a data source, retraining your model, to going back to the drawing board to devise a new model architecture.

Here is a deeper dive into each of these pieces that enable ML reliability in production products.

Observability

The key ingredient in making any system reliable is the ability to introspect the inner workings of the system. In the same way that a mechanic needs to peer under the hood of a car to see if your engine is running smoothly, an ML engineer needs to be able to peer under the hood of their model to understand how their model is fairing in production. While this seems obvious, many companies have been flying blind when it comes to deploying machine learning. Measuring your model's performance via aggregate performance metrics is not observability.

The best way to think about ML observability is how effectively your team can detect a problem with your model's performance, perform mitigation to the problem to stop the bleeding, identify the root cause of the regression, and perform remediation or solution to the problem. It's important to note that having the ability to detect a problem does not constitute full observability into an ML system. Without the ability to introspect to find the root cause or weight the sum of contributing factors, any resolution is going to be some form of guesswork.

To better illustrate what kind of things your tooling should be looking for, we first need to understand what are some things that can go wrong?

So what can go wrong?

What you should observe really depends on what can go wrong.

There are many different model failure modes and production challenges when working with ML models, each of which requires you to observe additional information in your system.

To start, the first step in the battle is detecting that an issue has occurred. How this is typically done is to measure a model performance metric such as running accuracy, RMSE, f1, etc. One catch is that this isn't as easy as it sounds. In the ideal case, you know the ground truth of your model's prediction pretty quickly after the model has made the decision, making it easy to determine how well your model is doing in production. Take, for example, predicting which ad a user might click on. You have a result around how well you did almost immediately after the model makes the decision. The user either clicked on it or they didn't!

Many applications of ML don't have this luxury of real-time ground truth, in which case proxy performance metrics such as relevant business metrics might be used instead. On top of model performance metrics, you may want to monitor service health metrics such as prediction latency, to ensure your service is providing a good experience for your users.

Once a regression has been detected by monitoring model performance or service health metrics, you need more information to understand what might be going on with your model. Some things that are important to keep tabs on to help with incident response:

Service:

- Latency of model predictions and user-facing latency
- Service downtime (pretty similar to software)

Data:

- New values in production unseen before in training
- Noisy or missing values in the data can have a big impact on the features consumed by a model.

Model:

- The underlying task that the model is performing can drift slowly or quickly change overnight!
- Your model may be biased in a way that was not designed (are some unexpected subsets of your users getting measurably different outcomes)
- Your model may be performing particularly poorly on some subsets of data (need to store and make sense of your model errors)

For each of these potential production challenges, your ML Observability toolset should enable your team to detect regressions and drill into them to best understand why they happened and what you can do about it.

Next, here are some tips on how to best manage shipping updates to your model in production.

Management of Change

Every time you push new changes into production, you risk introducing your users to issues that your team did not foresee and protect against.

In fact, let's say for the sake of it that your search model is regressing on your hypothetical e-commerce platform due to a new model rollout. Now that your business metrics caught that something was going wrong, and your observability tooling pinpointed the search model, what do we do about it? We alluded previously to the difference between mitigation and remediation. Here, since we are rapidly losing the company money, it's likely that the best course of action is to stop the bleeding as quickly as possible (mitigate the issue).

One option we may have is to revert back to the previous model we had deployed. Alternatively, we could ship our naive model, a model that may not have as good of performance but works consistently pretty well. In our case, this might just be displaying the exact results returned from elastic search.

To best protect against these potential issues from occurring rapidly and dramatically for the users of your product, ML systems should follow similar rollout procedures to those of software deployments.

In the same way that software is typically tested using static test cases to ensure that the code is not regressing any behavior, ML systems should also undergo static validation tests before deployment. For example, if you are shipping an autonomous driving service, running your new model through some standardized deterministic simulator routes might allow you to catch some obvious regressions.

While static validation is exceedingly important for improving the quality of your shipping product, there is no replacement for what you learn about a model in production. Let's talk about how you can get these learnings from your production model without risking a full outage or a degraded experience for all of your users.

You may want to ship your model to a subset of your users first to detect issues early and before all of your users catch a whiff of the issue. This technique is commonly referred to as a **canary deployment**.

As you gradually roll out your changes if a problem is detected via your ML monitoring systems, you should be able to easily and quickly revert back to a previous model version along with its corresponding software version.

Another topic that is closely related is the idea of **shadow deployment**. In a shadow deployment, you would start to feed the inputs that your existing model is seeing in production to your new model before you ship it. So while your users are still experiencing the predictions and user experience provided by the existing model, you can start to measure how your new model is performing and make any necessary changes to get it ready for prime-time.

One additional benefit of a shadow deployment is that you can perform experiments with multiple candidate models in a shadow deployment and choose the one that is able to perform best on your current production data. Now that we have some techniques to help us improve the quality of our deployments, let's talk about what you can do when you find an issue with your production model after you have deployed it into production.

Incident Response

Okay so we discovered an issue with our model in production, what should we do about it? This very much depends on your model application, but here we will talk about some general strategies about how to handle an issue in the short term (mitigation) and work towards the real fix (remediation).

Mitigation

To start, just as with software, you may be able to roll back to a previous model version and corresponding software/configuration. This mitigation strategy might help you if you promoted a bad model that got past your validation procedures; however, this will not always solve your issue. It's possible that your input data distribution or the underlying task of the model has changed, making the older model also a poor choice to have in production.

Another strategy that can work in some cases is to deploy a naive version of your model. This may generally have lower performance than your more complex model, but it may do better in the face of change in input and expected output distributions. The model doesn't need to be machine-learned and can just be a simple heuristic-based model. This strategy may help you buy time while you rework your more complex, but more performant model.

Remediation

This brings us to the most common advice that is given to resolve an incident caused by an ML model in production: Just retrain it! This advice is common because it covers a lot of potential failure modes for a model. If the input data has shifted or the underlying task has changed, retraining on newer production data might be able to solve your issue. The world changes over time, and it's possible your model needs to be regularly retrained to stay relevant.

Retraining strategies could encompass a whole other technical article, so let's skip to the abridged version. You have some options when you retrain your model:

You can choose to **upsample certain subsets of your data**, potentially to fix issues regarding unintended bias or underperformance for a category of your data.

You also can **sample the newer production data** to build new training sets to use if you think that the shift in your input/output distributions is here to stay.

If you think that this shift in distributions is temporary and potentially seasonal, you can train a new version of your model on the data from this seasonal period and deploy it, or turn to engineering features to help your model understand this seasonal indicator in the function it is trying to approximate.

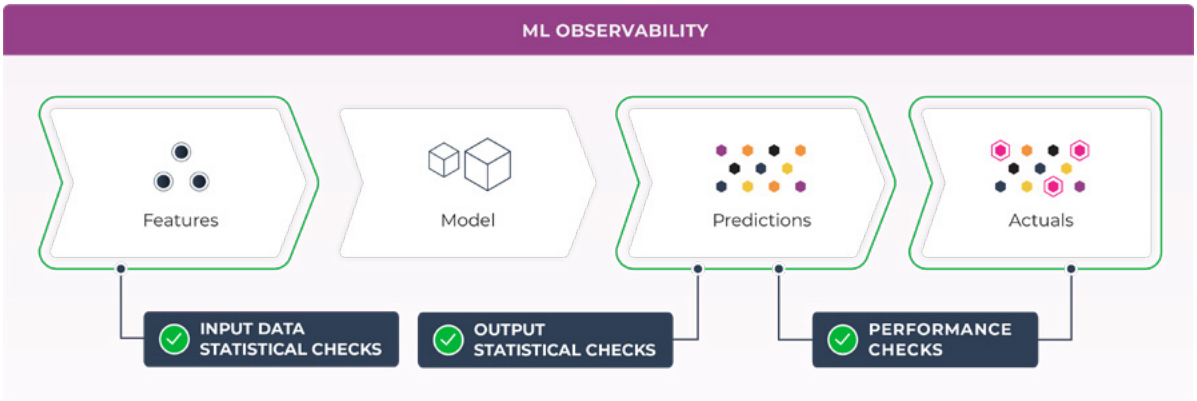
It's possible that your model's performance may have dipped due to the introduction of a new category of examples that it had not seen in it's training. If this category of examples is sufficiently different you may need an entirely new model to handle these particular examples. The process of training a separate model and employing a higher-level model to determine which to use for a particular example is commonly referred to as a **federation**.

The last option is going back to the drawing board. If retraining hasn't helped restore performance and your older models also fail to do the job, it's possible that the task has changed significantly enough to require some of the following: a new model architecture, new features, and new data processing steps.

It took years for the software world to get behind the reliability framework outlined above. With the three pillars of observability, management of change, and incidence response, teams can translate the reliability gains from the world of software to the world of ML applications. It's now up to the ML Ops space to provide the tools that we desperately need to make ML applications reliable.

ML SERVICE-LEVEL PERFORMANCE MONITORING

This section covers the often overlooked field of service-level ML performance in additional detail by breaking down how it can be measured and improved.



Let's start off by breaking down what we mean by service-level performance of an ML system. In essence, there really are two important measurements that we are going to talk about: service performance and model performance.

Service performance is the time it takes to load the model into memory, gather the requisite data, and compute the features the model needs to make its prediction. Service performance also includes the time it takes for the user to be made aware of the decision that the model has made.

Model performance is the time it takes for your model to make its prediction once it is fed its input.

In a real-time system, both of these metrics contribute to the user-perceived latencies of your application, which can be the difference between a customer choosing to use your product or service over another. As a result, it's important not just to monitor these service-level performance metrics, but also make real progress in reducing these latencies in your application.

Let's start by taking a look at what you might want to monitor and improve upon to hone in on making your service more performant.

Optimizing Performance of the Service

Input Feature Lookup

Before the model can even make a prediction, all of the input features must be gathered or computed by the service layer of the ML system. Some of the features will be passed in by the caller, while other features might be collected from a datastore or calculated in real-time.

For example, a model predicting the likelihood of a customer responding to an ad might take in the historical purchase information of this customer. The customer wouldn't provide this when they view the page themselves, but the model service would query a data warehouse to fetch this information. Gathering input features can generally be classified into two groups:

1. **Static Features**: Features that are less likely to change quickly and can be stored or calculated ahead of time. For example, a customer's historical purchase patterns or preferences can be calculated ahead of time.

2. **Real-Time Calculated Features**: Features that require being calculated over a dynamic time window. For example, when predicting ETAs (estimated time of arrival) for food delivery, you might need to know how many other orders have been made in the last hour.

In practice, a model typically uses a mix of static and real-time calculated features. Monitoring the lookup and transformations needed for each feature is important to trace where the latency is coming from in the ML system. It's important to remember that your service level performance in the input feature lookup stage is only as good as your slowest feature.

Pre-Computing Predictions

In some use cases, it is possible to reduce prediction latency by precomputing predictions, storing them, and serving them using a low-latency read datastore. For example, a streaming service might compute and store ahead of time the recommendations for a new user of their service.

This type of offline batch-scoring job can vastly reduce latencies in the serving environment because the brunt of the work has been done before the model has even been called.

For example, recommendation systems used by a streaming service like Netflix can pre-compute the movies or TV shows that you are likely to enjoy when you are not using the service so that the next time you login you are quickly greeted with some personalized content without the long loading screen.

Optimizing Performance of the Model

Reduce Complexity

Now that we have looked at service performance, let's turn our attention to how you might monitor and improve your model performance.

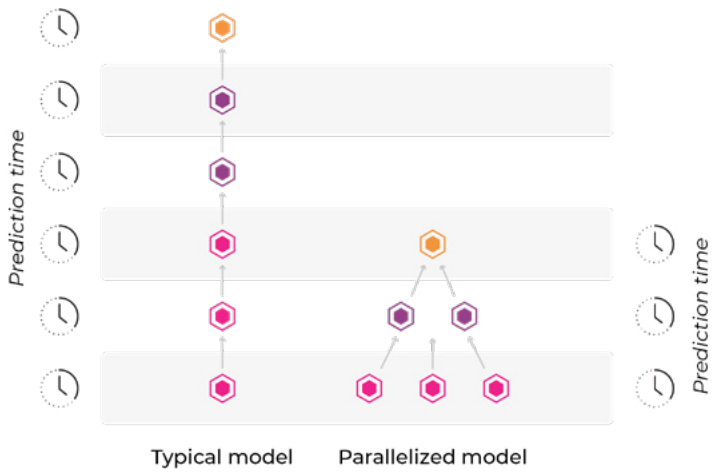
One approach to optimize model prediction latency is to reduce the complexity of the model. Some examples of reducing complexity could be reducing the number of layers in a neural network, reducing levels in decision trees, or pruning any irrelevant or unused part of a model.

In some cases, this might be a direct tradeoff to the model efficacy. For example, if there are more levels in a decision tree, there are more complex relationships that can be captured from the data and therefore increase the overall effectiveness of the model. However, fewer levels in a decision tree can reduce prediction latency.

Balancing the efficacy of the model (accuracy, precision, AUC, etc.) with its required operational constraints is important to strive for any model to be deployed. This becomes especially relevant for models that are embedded on more constrained mobile devices.

Parallelize

Aside from reducing the complexity of the model, something you can do to improve your model performance in production is to re-architect your model to be more parallelizable. If a part of your model doesn't depend on the output of another part of your model, why not run both of these sections *at the same time*.



The cloud-ML industry is moving to highly scalable on-demand clouds that allow you to leverage football fields of specialized computers to run your model on. In a similar vein, mobile processors are dedicating significant portions of their chips to machine learning accelerators, which allow developers to exploit the parallel nature of their model inference pipelines.

If you have the ability to throw more cores at the problem at prediction time, you can leverage a parallel nature of your model to speed up your prediction time. You may be leaving performance on the table if you don't look into how your model can be reimagined in a more parallel way.

Takeaway

While traditional ML performance monitoring is of the utmost importance for measuring and improving your application of machine learning, it doesn't capture the full picture of how a user is experiencing your application. *Service level performance metrics matter*, and a change that increases your accuracy by one percent that causes a 500 millisecond regression might not be worth it for your use case!

If you don't see the trade-offs you are making with every change you make to your system, you are going to slowly bury your model in a pile of small performance regressions that add up to a slow, unwieldy product. Have no fear: there are a number of techniques to diagnose performance issues and ultimately improve your model's service-level performance, but first you have to start paying attention to the milliseconds.

Conclusion

While this ebook contains a wealth of best practices and resources, ultimately it's teams that put these insights into practice. Hopefully by collaborating through a common platform, everyone who touches machine learning—from the data scientists who build models to the ML engineers who deploy them and even the executives overseeing efforts and ROI—can begin to develop a common culture and strategy around observability that moves beyond mere monitoring or compliance.

To learn more about Arize AI's leading observability platform and use-cases specific to your industry, **Request a Demo.**

For the latest on ML observability best practices and tips, **Sign up** for our monthly newsletter The Drift.