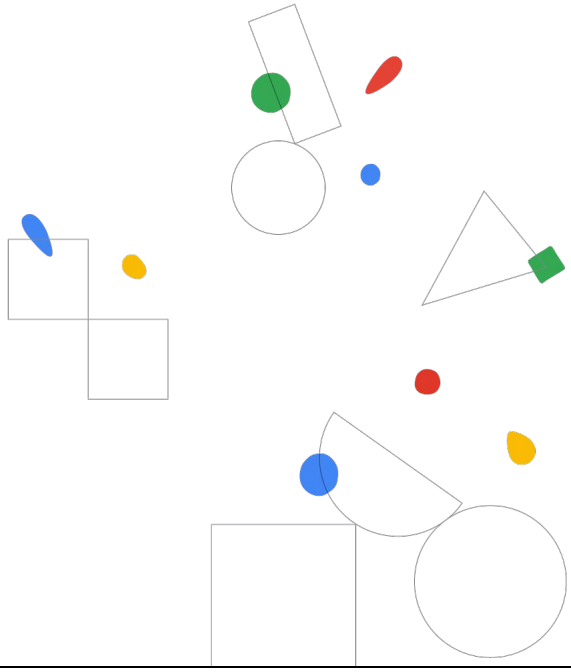
Google Cloud

Introduction



Module 02

Designing adaptable ML systems

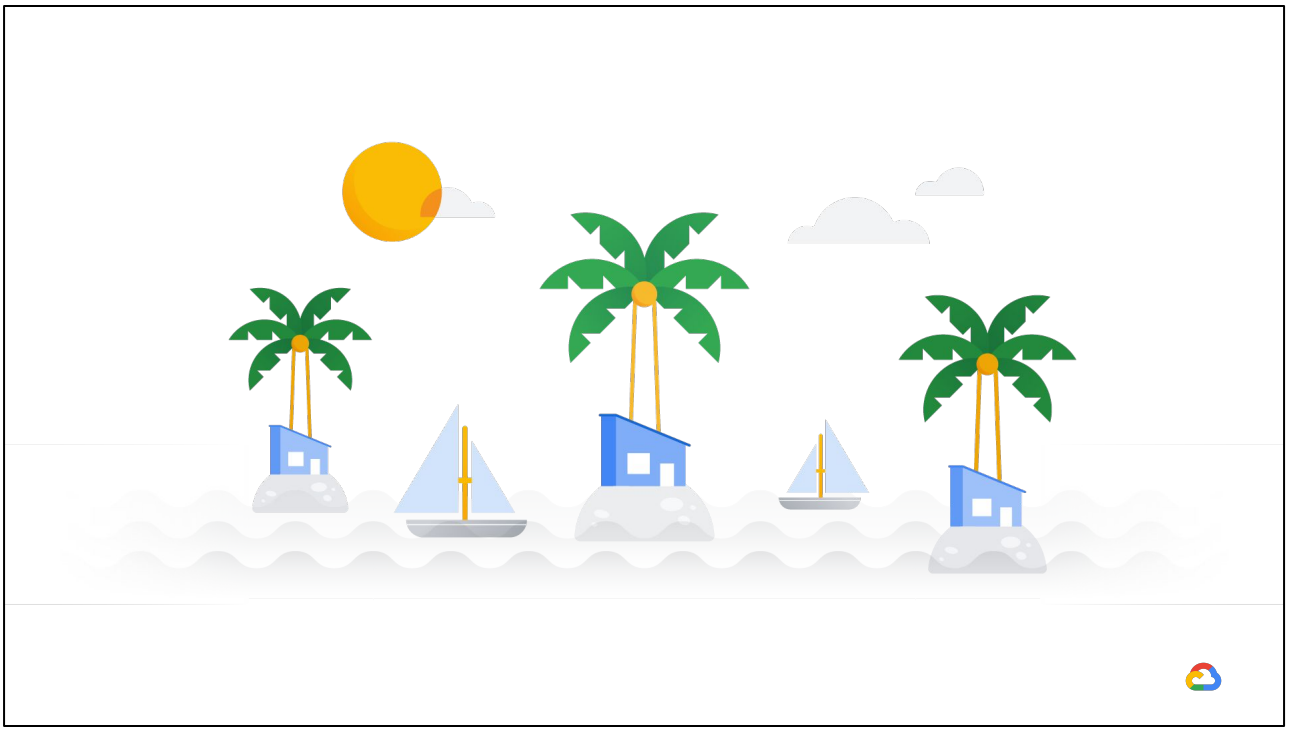
Welcome to Designing Adaptable ML systems.

Objectives

1. Recognize the ways that a model is dependent on data
2. Make cost-conscious engineering decisions
3. Know when to roll back a model to an earlier version
4. Debug the causes of observed model behavior
5. Implement a pipeline that is immune to one type of dependency

In this module, we'll explore how to:

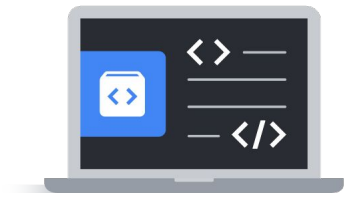
- Recognize the ways that a model is dependent on data
- Make cost-conscious engineering decisions
- Know when to roll back a model to an earlier version
- Debug the causes of observed model behavior
- Implement a pipeline that is immune to one type of dependency



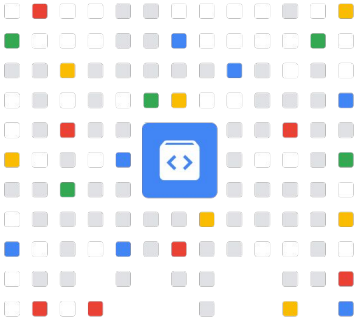
In the 16th century, John Donne famously wrote in one of his poems that no man is an island.

He meant that human beings need to be part of a community to thrive.

Software today is modular



Monolithic design

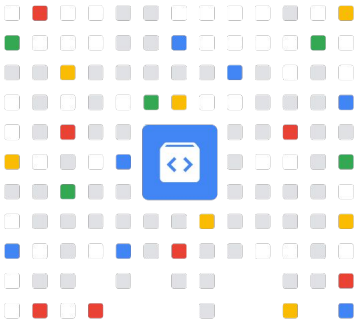


Modular design

In software engineering terms, we would say that few software programs adopt a monolithic island-like design.

Instead, most software today is modular, and depends on other software.

Modular programs are more maintainable

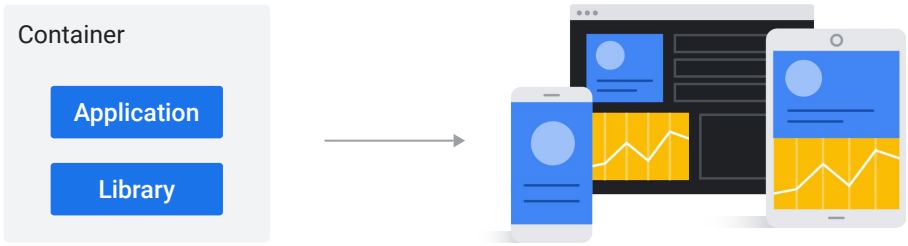


Modular design

- More maintainable
- Easier to reuse
- Easier to test
- Easier to fix

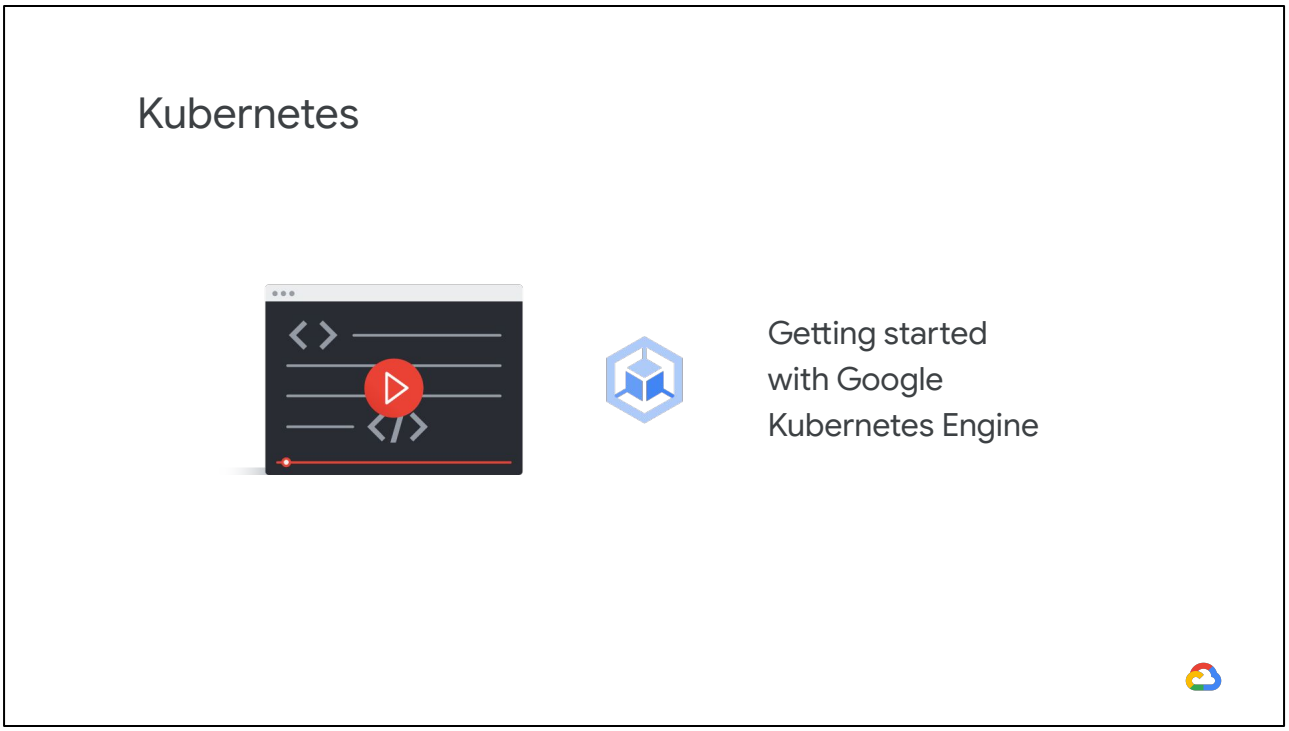
Modular programs are more maintainable, as well as easier to reuse, test, and fix because they allow engineers to focus on small pieces of code rather than the entire program.

Containers



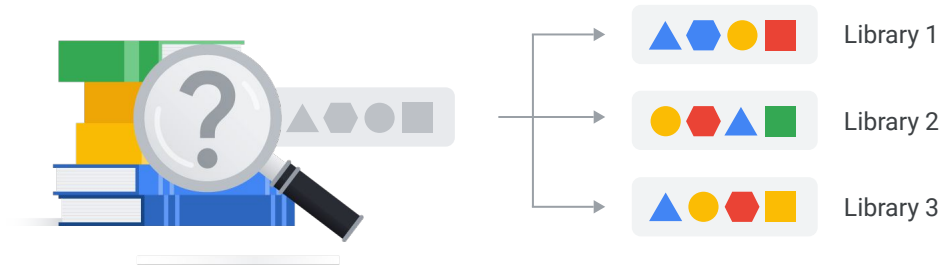
Containers make it easier to manage modular programs.

A container is an abstraction that packages applications and libraries together so that the applications can run on a greater variety of hardware and operating systems. This ultimately makes hosting large applications better.



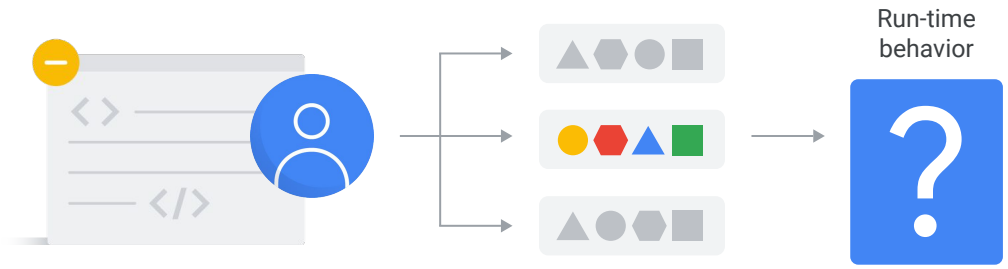
To learn more about Kubernetes, Google's open source container orchestration software, check out the getting started with Google Kubernetes engine course.

Identify a specific version of a library



But what if there was no way to identify a specific version of a library, and you had to rely on finding similar libraries at run-time?

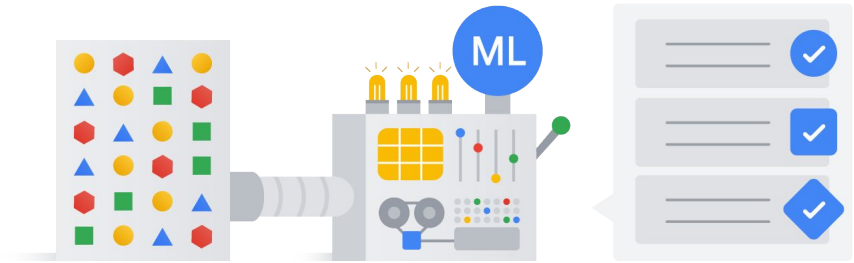
Identify a specific version of a library



Furthermore, what if someone else got to choose which version got run and they didn't know or really care about your program?

There would be no way of knowing what the run-time behavior would look like.

This is precisely the case for ML



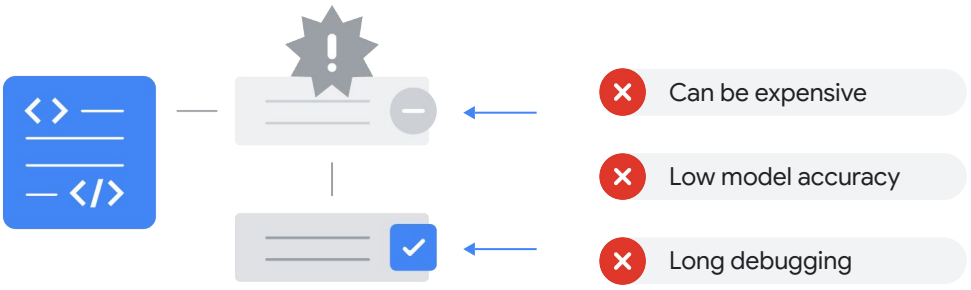
Unfortunately, this is precisely the case for machine learning, because the run-time instructions, for example, the model weights, depend on the data that the model was trained on.

This is precisely the case for ML

Additionally, similar data will yield similar instructions.

And finally other people including other teams and our users create our data.

Mismanaged dependencies



Just like in traditional software engineering, mismanaged dependencies say, code that assumes one set of instructions, will be called when another end up being called instead can be expensive.

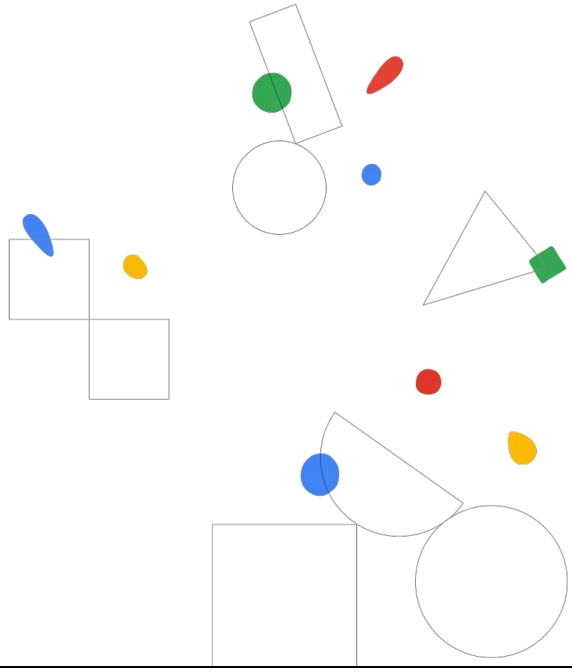
Your models' accuracy might go down or become unstable.

Sometimes, the errors are subtle and your team may end up spending a large proportion of its time debugging.

With a better understanding of how to manage data dependencies, many problems can be either detected quickly or circumvented entirely

Google Cloud

Adapting to data



Module 02

Designing adaptable ML systems

Adapting to change

When it comes to adapting to change, consider which of these four is most likely to change?

A An upstream model

B A data source maintained by another team

C The relationship between features and labels

D The distributions of inputs

When it comes to adapting to change, consider which of these four is more likely to change?

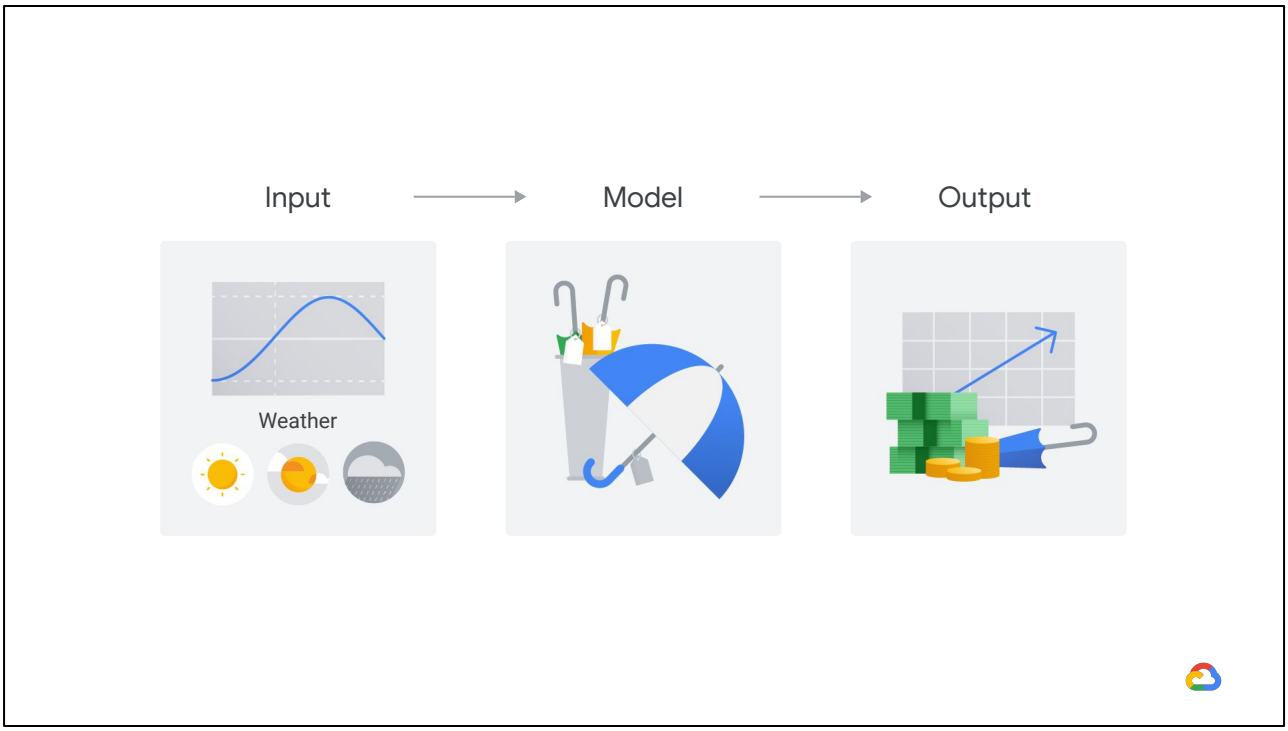
An upstream model, a data source maintained by another team, the relationship between features and labels, or the distributions of inputs. The answer is

Adapting to change

When it comes to adapting to change, consider which of these four is most likely to change?

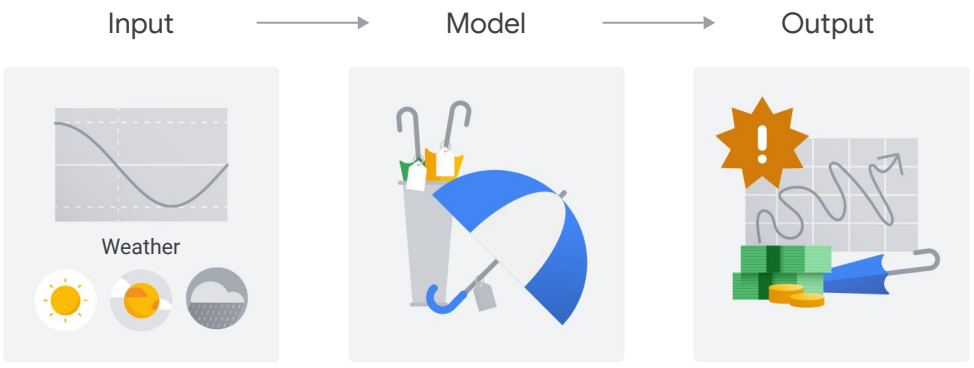
- An upstream model
- A data source maintained by another team
- The relationship between features and labels
- The distributions of inputs

that all of them can, and often do, change. Let's see how this happens, and what to do about it with a couple example scenarios.



Let’s say that you've created a model to predict demand for umbrellas that accepts as input an output from a more specialized weather prediction model.

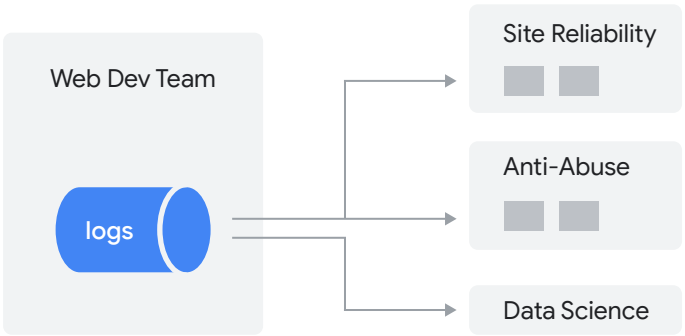
Unbeknownst to you and the owners, this model has been trained on the wrong years of data. Your model, however, is fit to the upstream model’s outputs. What could go wrong?



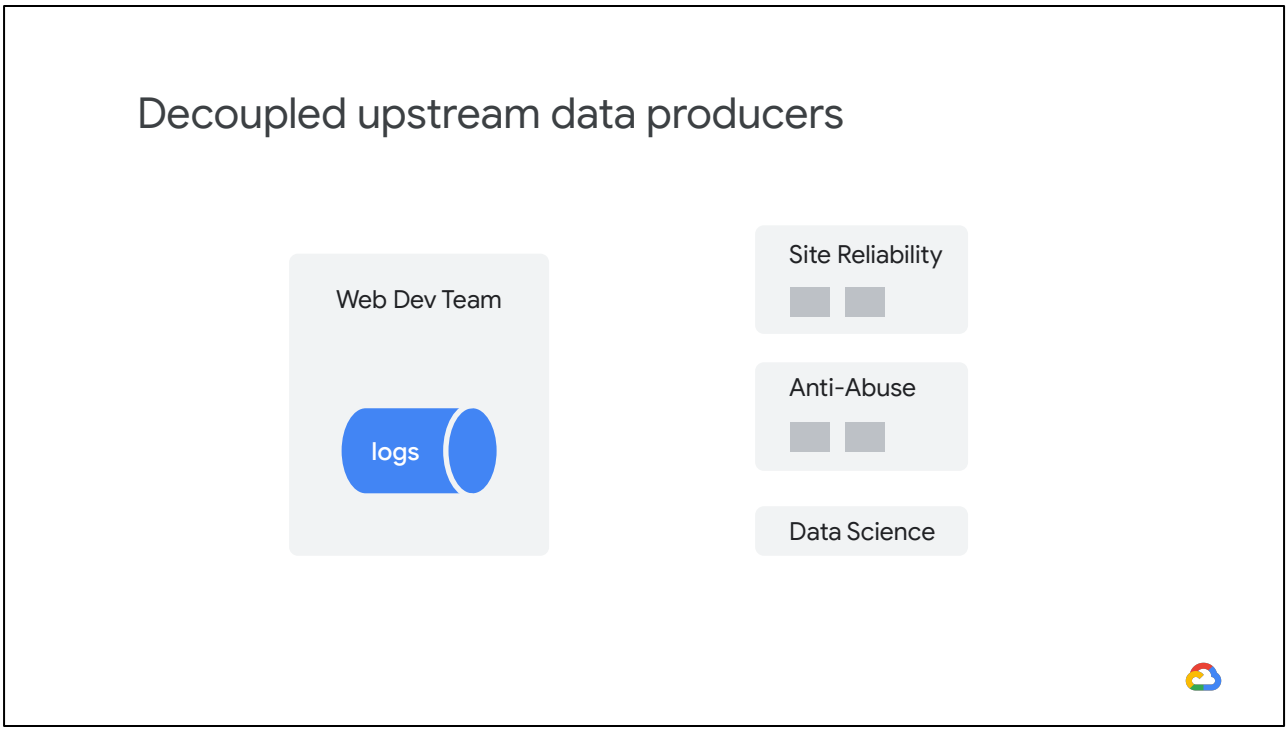
One day, the model owners silently push a fix and the performance of your model, which expected the old model's distribution of data, drops.

The old data had below-average rainfall and now you're under-predicting the days when you need an umbrella.

Decoupled upstream data producers

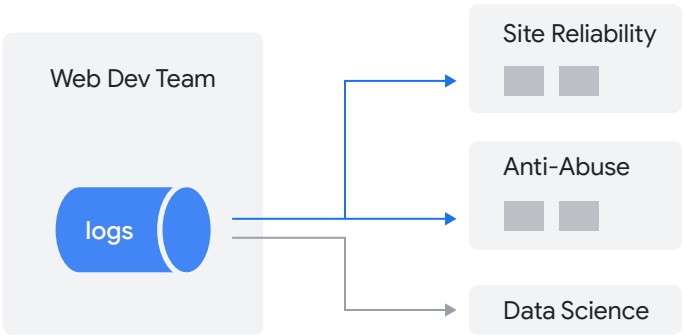


Here's another scenario. Let's say your small data science team has convinced the web development team to let you ingest their traffic logs.



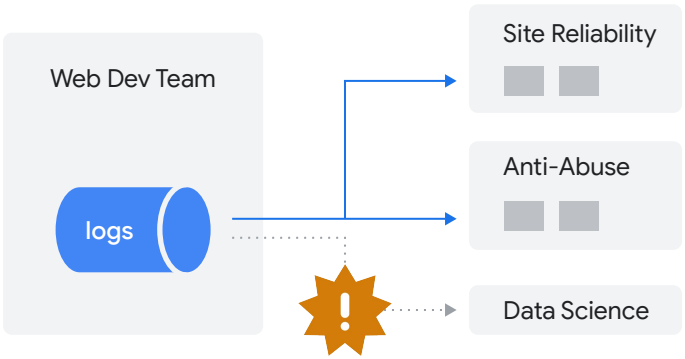
Later, the web development team refactors their code and changes their logging format,

Decoupled upstream data producers



but continues publishing the old format.

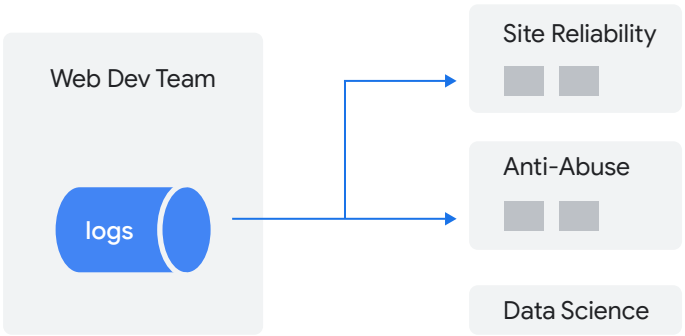
Decoupled upstream data producers



At some point, they stop publishing in the old format but they forget to tell your team.

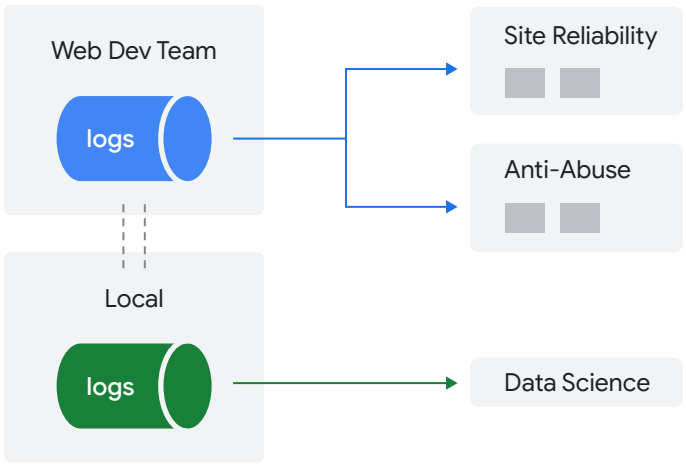
Your model's performance degrades after getting an unexpectedly high number of null features.

Decoupled upstream data producers

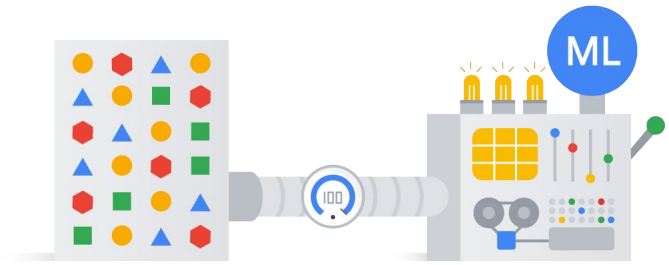


To fix this problem, first, you should stop consuming data from a source that doesn't notify downstream consumers.

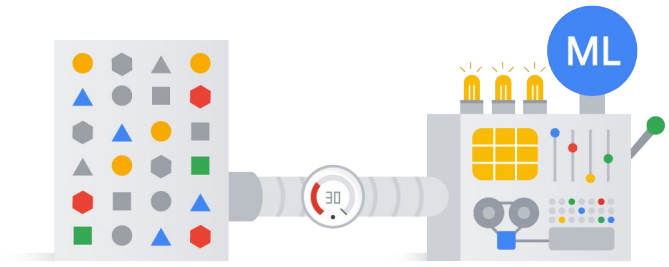
Decoupled upstream data producers



Second, you should consider making a local version of the upstream model and keeping it updated.



Sometimes, the set of features that the model has been trained on include

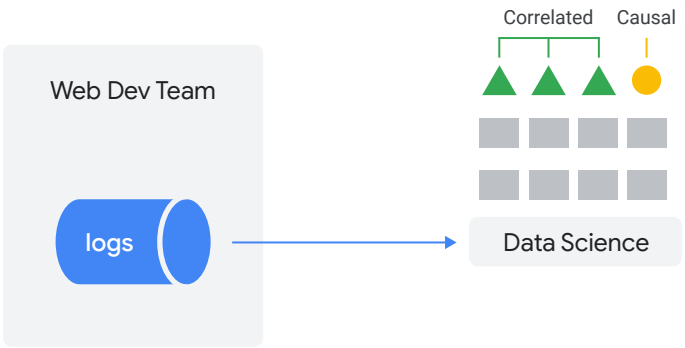


many that were added indiscriminately, which may worsen performance at times.

Underutilized data dependencies

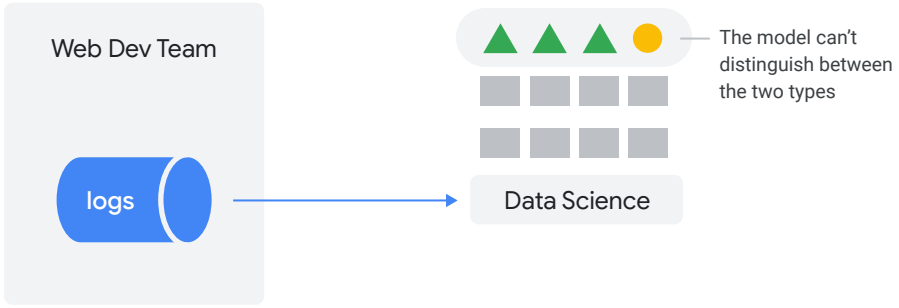
For example, under pressure during a sprint, your team decided to include a number of new features without understanding their relationship to the label.

Underutilized data dependencies



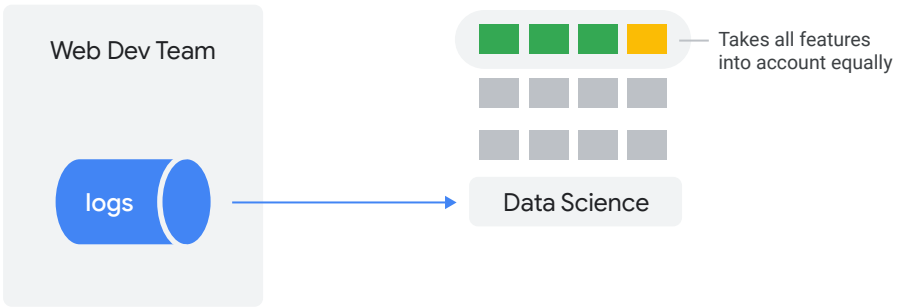
One of them is causal, while the others are merely correlated with the causal one.

Underutilized data dependencies



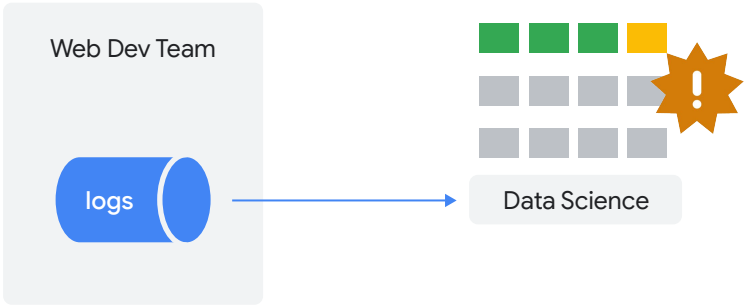
The model can't distinguish between the two types, and takes all features into account equally.

Underutilized data dependencies



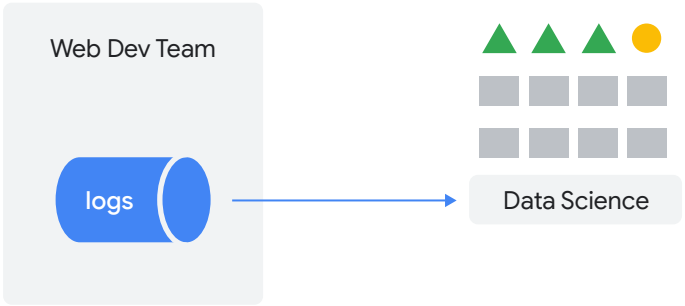
and takes all features into account equally.

Underutilized data dependencies



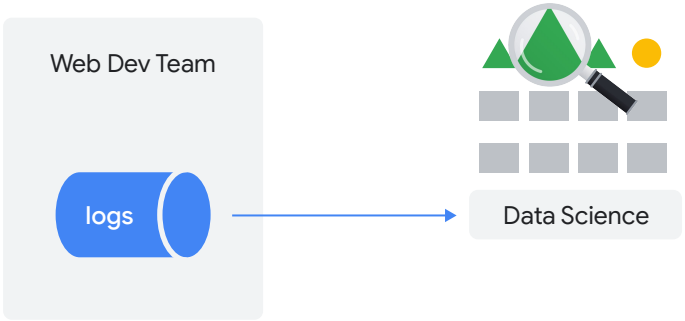
Months later, the correlated feature becomes decorrelated with the label and is thus no longer predictive. The model's performance suffers.

Underutilized data dependencies



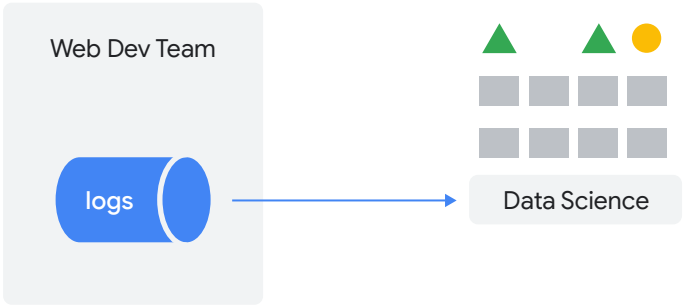
To address this,

Underutilized data dependencies



features should always be scrutinized before being added, and all features should be subjected to leave-one-out evaluations, to assess their importance.

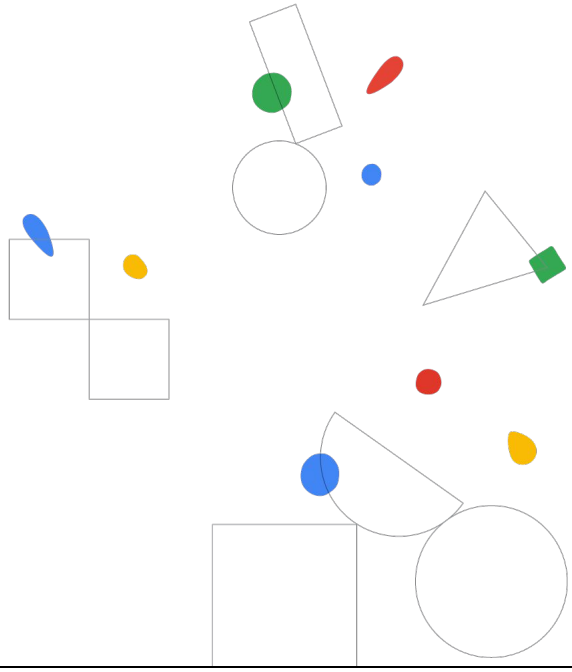
Underutilized data dependencies



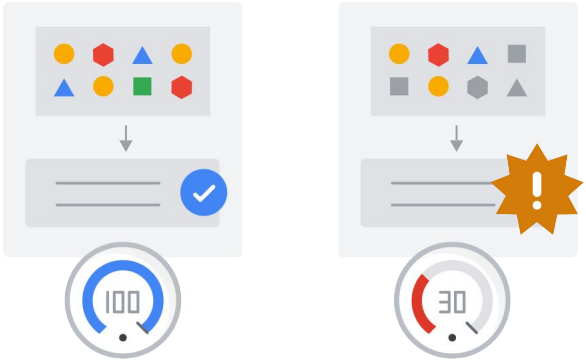
and all features should be subjected to leave-one-out evaluations, to assess their importance.

Google Cloud

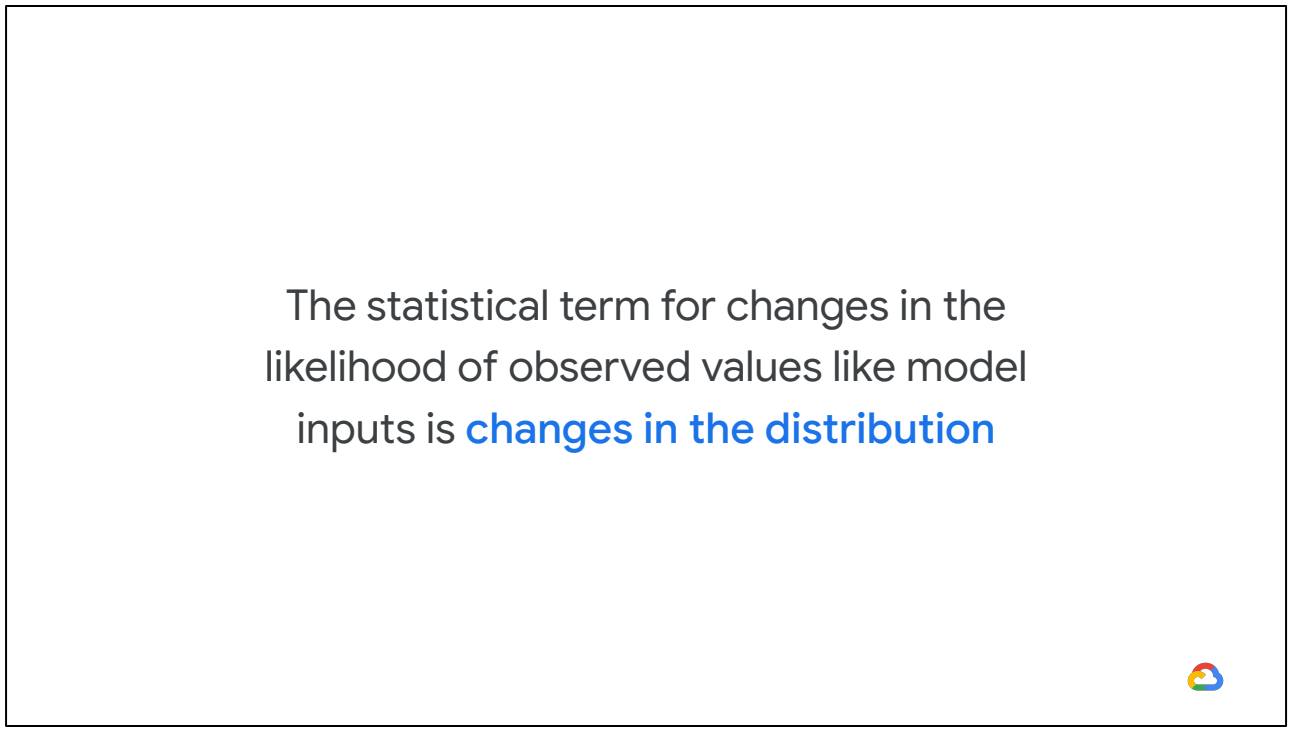
Changing distributions



Module 02
Designing adaptable ML systems



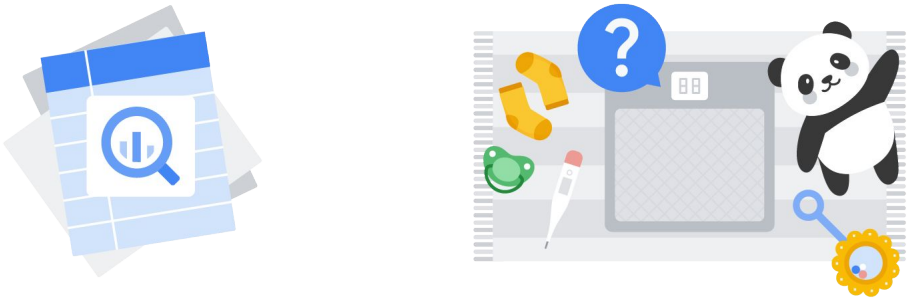
Earlier you saw how, in the context of ingesting an upstream model, our model's performance would degrade if it expected one input but ingested another.



The statistical term for changes in the likelihood of observed values like model inputs is *changes in the distribution*.

Changes in label distribution

Natality database → Baby weight prediction

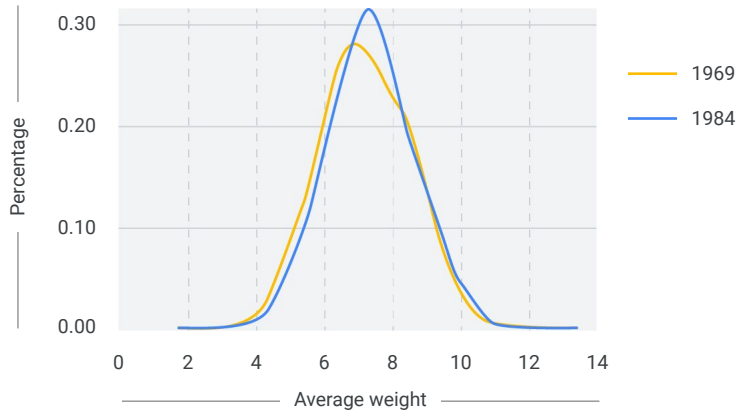


For example, sometimes the distribution of the label changes.

We've looked at the natality dataset in BigQuery and tried to predict baby weight.

Baby weight has actually changed over time. It peaked in the 1980s and has since been declining.

Changes in label distribution



In 1969, babies weighed significantly less than they did in 1984. When the distribution of the label changes, it could mean that the relationship between features and labels is changing as well.

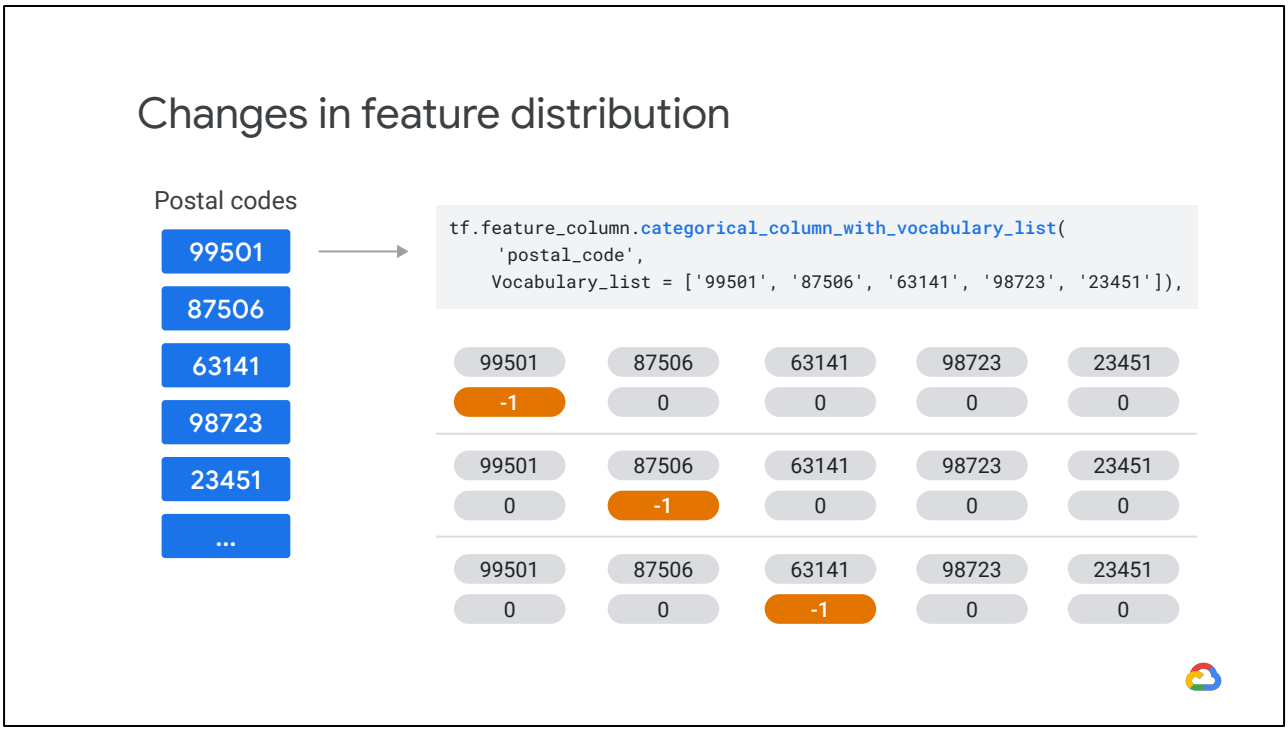
At the very least, it's likely that our model's predictions, which will typically match the distribution of the labels in the training set, will be significantly less accurate.

Changes in feature distribution

Postal code → Population movement patterns

However, sometimes it's not the labels, but the features, that change their distribution.

For example, say you've trained your model to predict population movement patterns using postal code as a feature. Surprisingly, postal codes aren't fixed. Every year, governments release new ones and deprecate old ones.



Now as a ML practitioner, you know that postal codes aren't really numbers. So you've chosen to represent them as categorical feature columns, but this might lead to problems.

If you chose to specify a vocabulary, but set the number of out of vocab buckets to 0, and didn’t specify a default, then the distribution may become skewed toward the default value, which is -1.

Changes in feature distribution

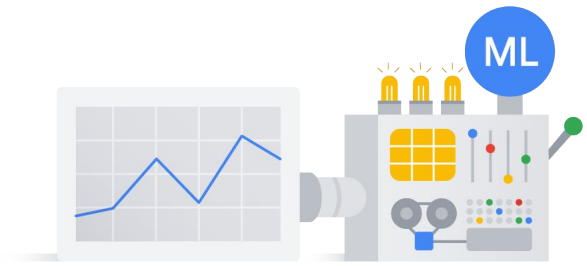
And this might be problematic because the model may be forced to make predictions in regions of the feature space which were not well represented in the training data.

There's another name for when models are asked to make predictions on points in feature space that are far away from the training data, and that’s extrapolation.

Extrapolation means to generalize outside the bounds of what we’ve previously seen.

Protect from changing distributions

- Monitoring

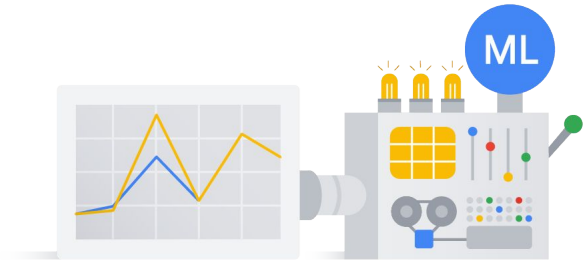


You can protect yourself from changing distributions using a few different methods.

The first thing you can do is be vigilant through monitoring. You can look at the descriptive summaries of your inputs,

Protect from changing distributions

- Monitoring

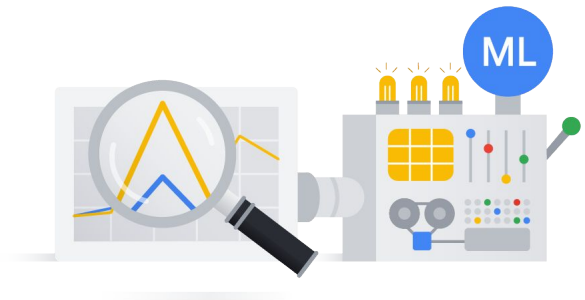


and compare them to what the model has seen.

If, for example, the mean or the variance has changed substantially,

Protect from changing distributions

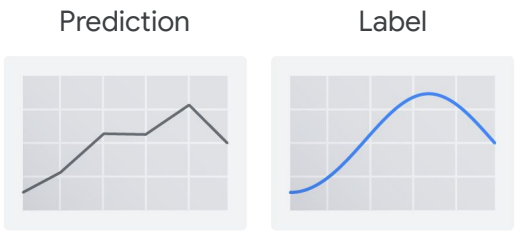
- Monitoring



then you can analyze this new segment of the input space, to see if the relationships learned still hold.

Protect from changing distributions

- Monitoring
- Check residuals



You can also look to see whether the model's residuals, that is the

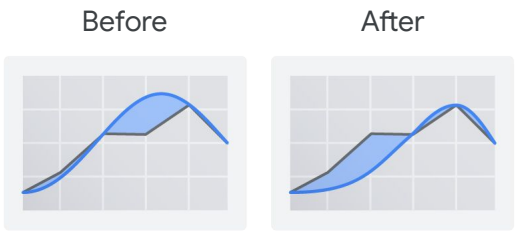
Protect from changing distributions

- Monitoring
- Check residuals

difference between its predictions and the labels, has changed as a function of your inputs.

Protect from changing distributions

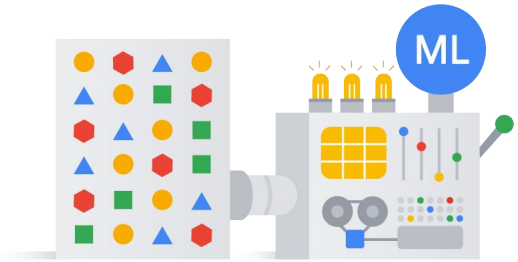
- Monitoring
- Check residuals



If, for example, you used to have small errors at one slice of the input and large in another, and now it's switched, this could be evidence of a change in the relationship.

Protect from changing distributions

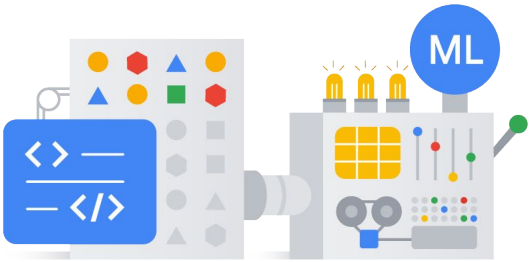
- Monitoring
- Check residuals
- Emphasize data recency



Finally, if you have reason to believe that the relationship is changing over time,

Protect from changing distributions

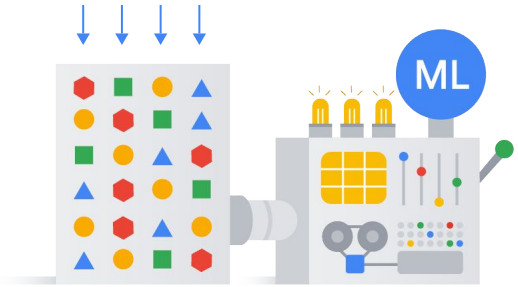
- Monitoring
- Check residuals
- Emphasize data recency



you can force the model to treat more recent observations as more important by writing a custom loss function,

Protect from changing distributions

- Monitoring
- Check residuals
- Emphasize data recency
- Regularly retrain your model

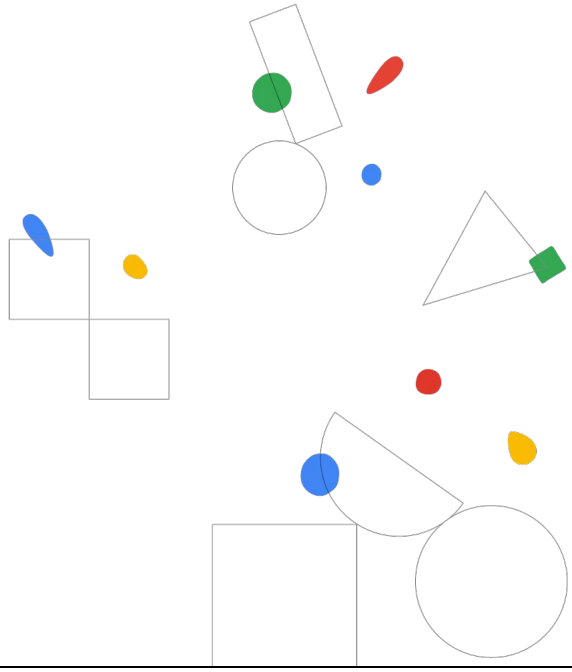


or by retraining the model on the most recent data.

Google Cloud

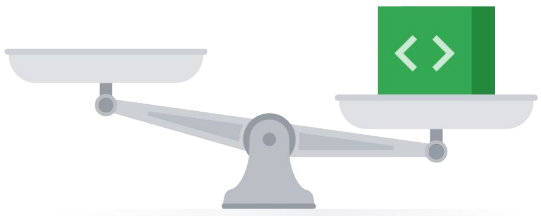
Lab

Adapting to Data

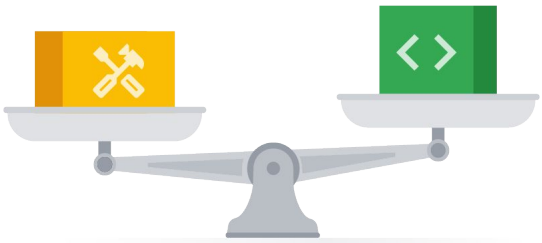


Module 02

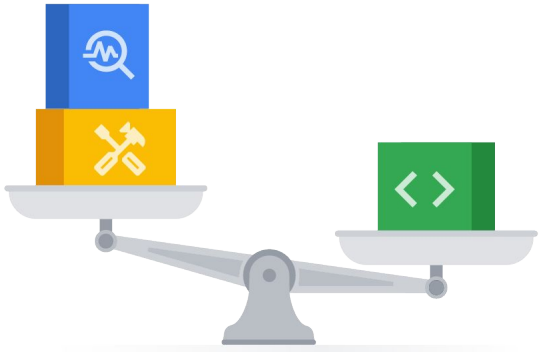
Designing adaptable ML systems



When leading a team of engineers, many decisions are informed by



technical debt and other sorts of



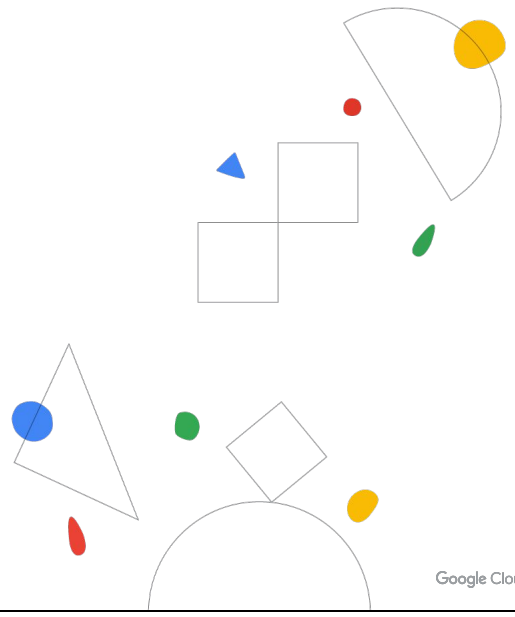
cost-benefit analyses.

Very high rates of return

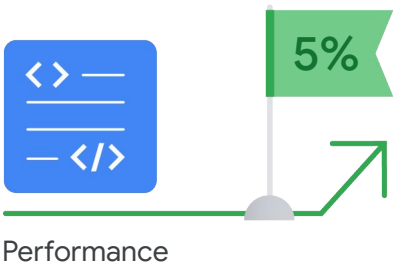
The best teams get very high rates of return on their investments. With that in mind, let's consider a few scenarios.

Scenario 1

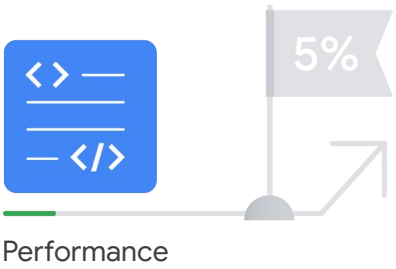
Code Sprint



Let’s imagine that you’re the leader of a team of engineers and you are nearing the end of a code sprint.



One of the team’s goals for the sprint is to increase performance on the model by 5%.

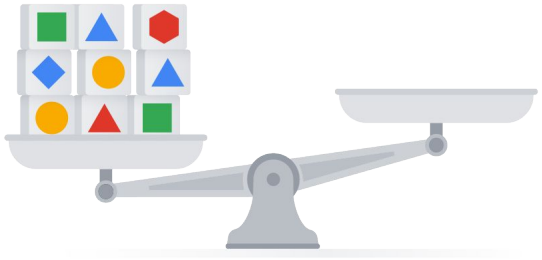


Currently, however, the best performing model is only marginally better than what was around before.

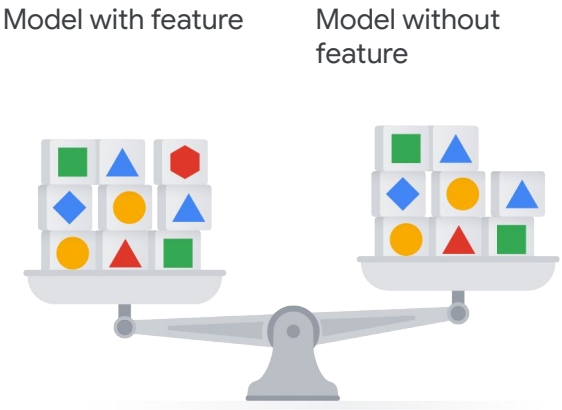
Ablation analysis

One of the engineers acknowledges this but still insists that it’s worth spending time doing an extensive ablation analysis

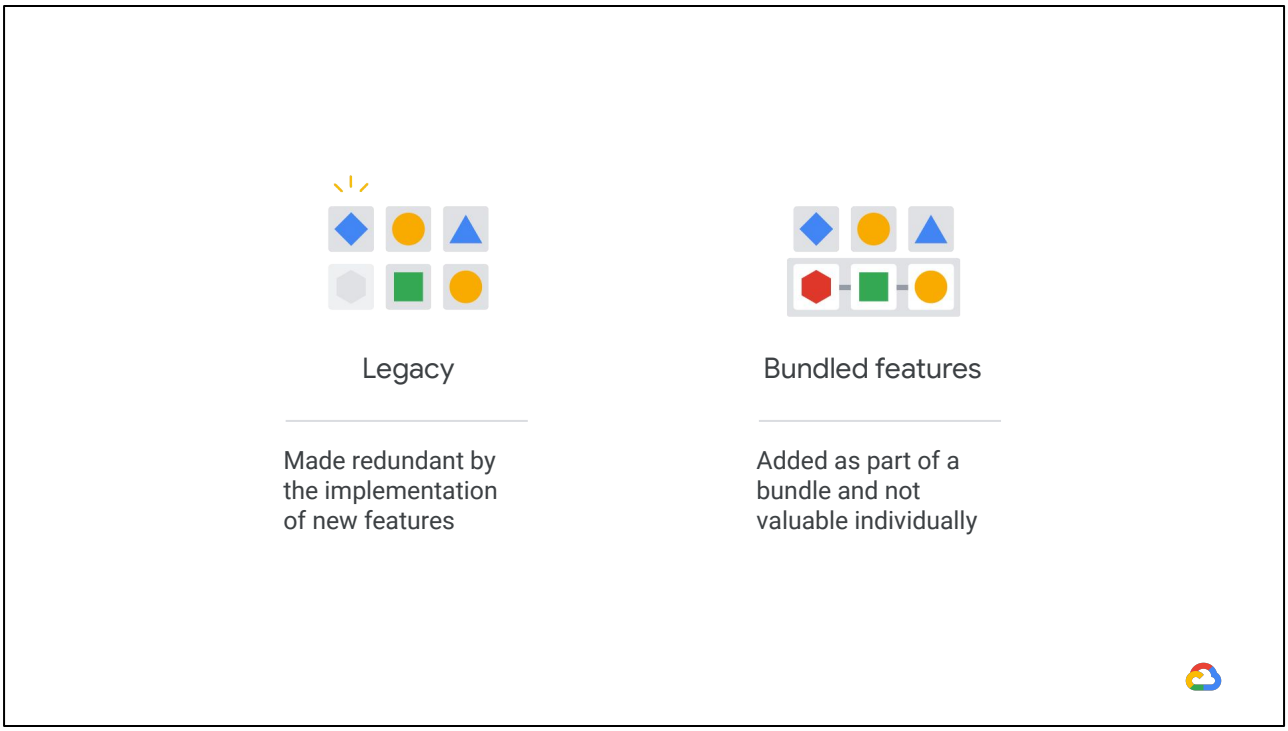
Model with feature



where the value of an individual feature is computed by comparing it



to a model trained without it. What might this engineer be concerned about?



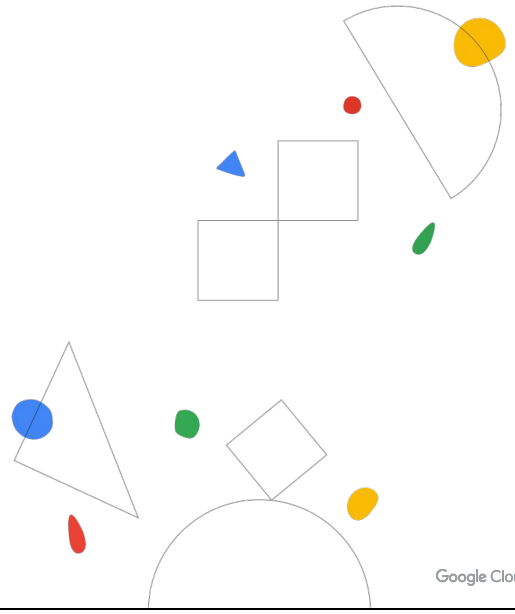
The engineer might be concerned about legacy and bundled features. Legacy features are older features that were added, because they were valuable at the time. But since then, better features have been added, which have made them redundant without our knowledge.

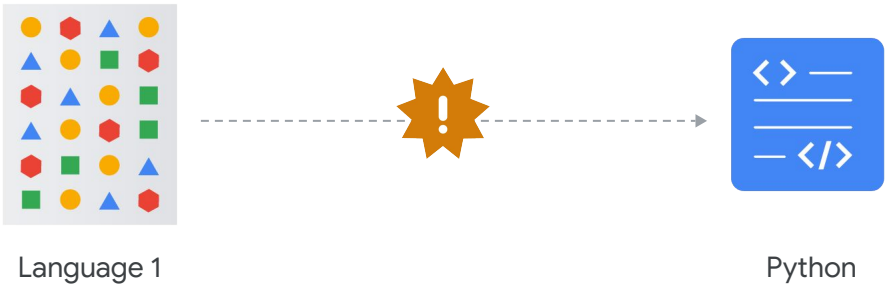
Bundled features on the other hand, are features that were added as part of a bundle, which collectively are valuable but individually may not be.

Both of these features represent additional unnecessary data dependencies.

Scenario 2

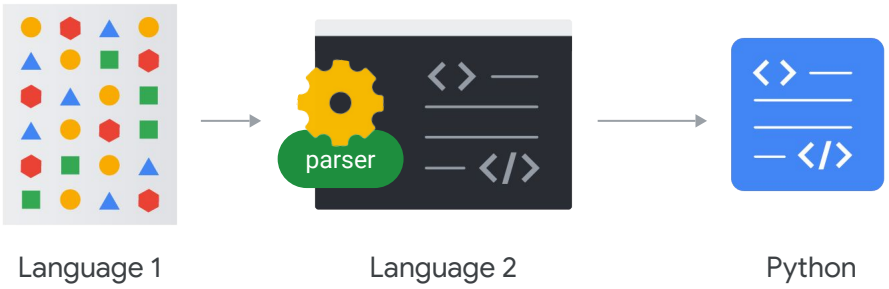
A Gift Horse





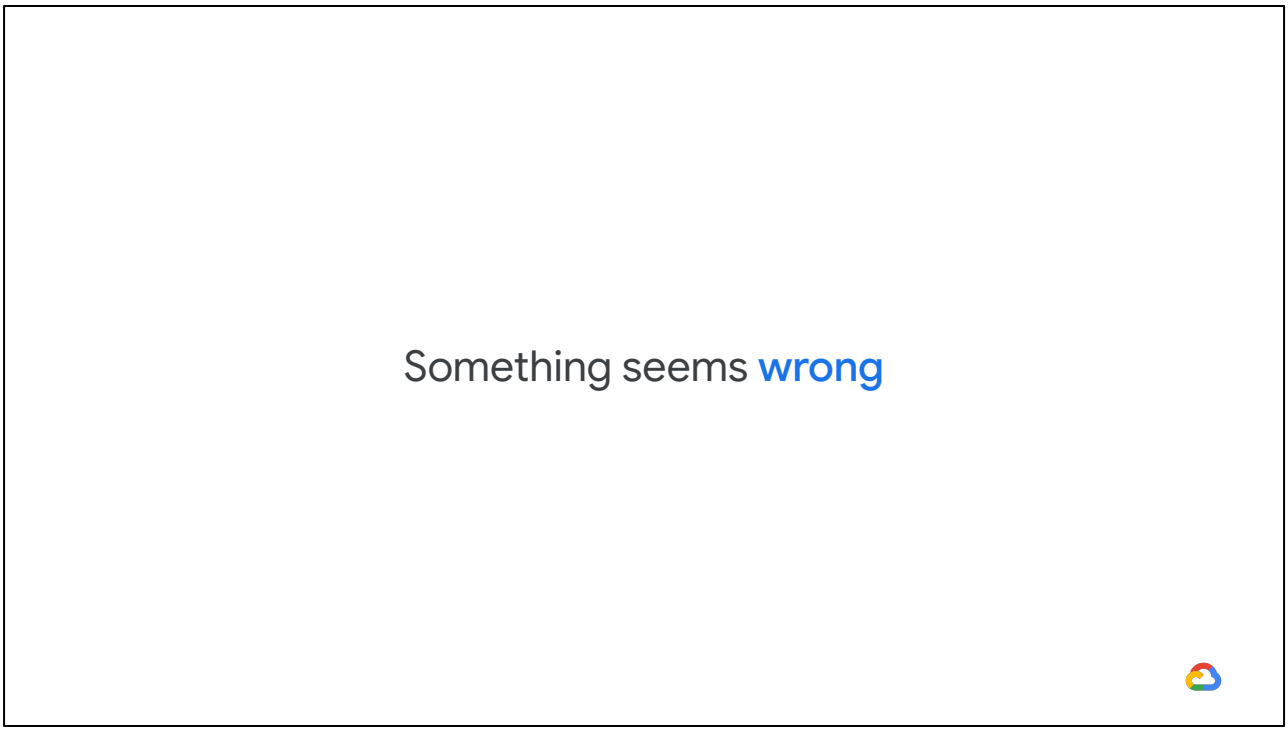
In another scenario, another engineer has found a new data source that is very much related to the label.

The problem is that it's in a unique format and there's no parser written in Python, which is what the codebase is composed of.

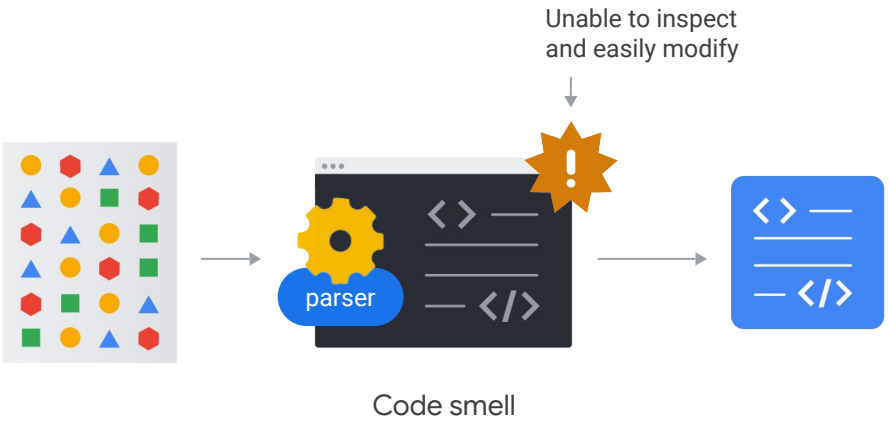


Thankfully, there is a parser on the web but it's closed source and written in a different language.

The engineer is thinking about the model performance.



Something in the back of your mind seems wrong.



Code smell

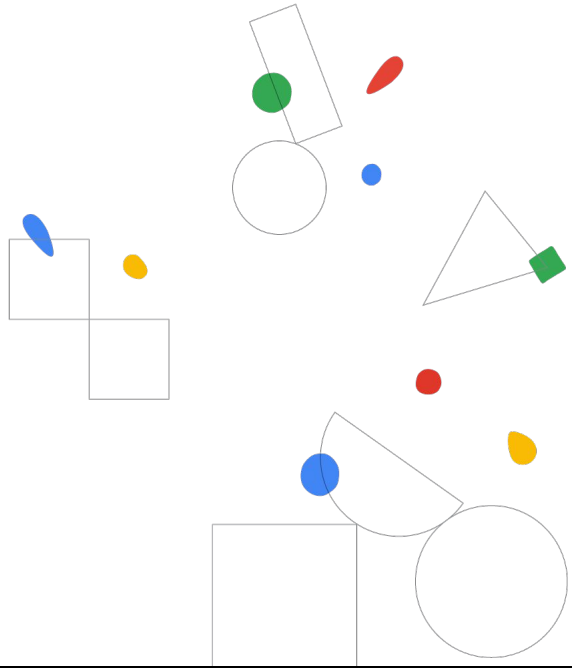
What is it? It's the smell.

No, really! There's a concept called code smell and it applies in ML as well.

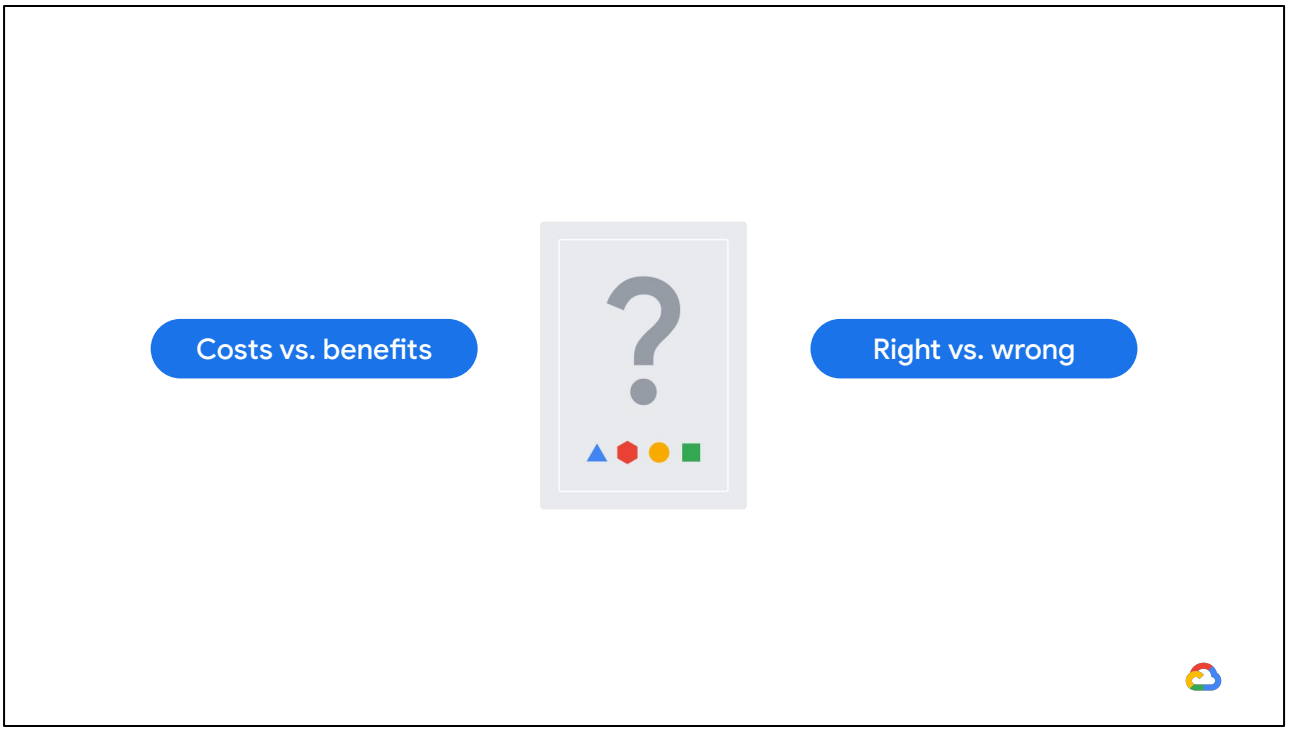
In this case, you might be thinking, "I wonder what introducing code that we can't inspect and are unable to easily modify into our testing in production frameworks will do."

Google Cloud

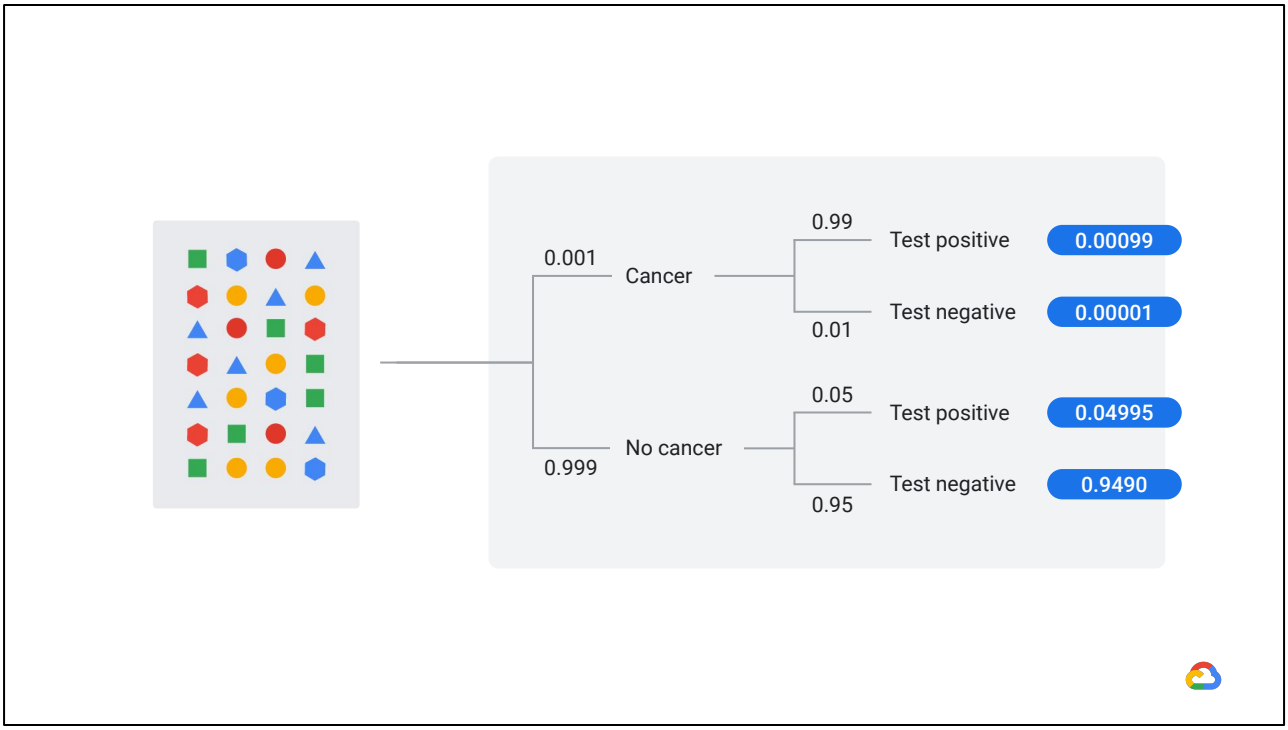
Right and wrong decisions



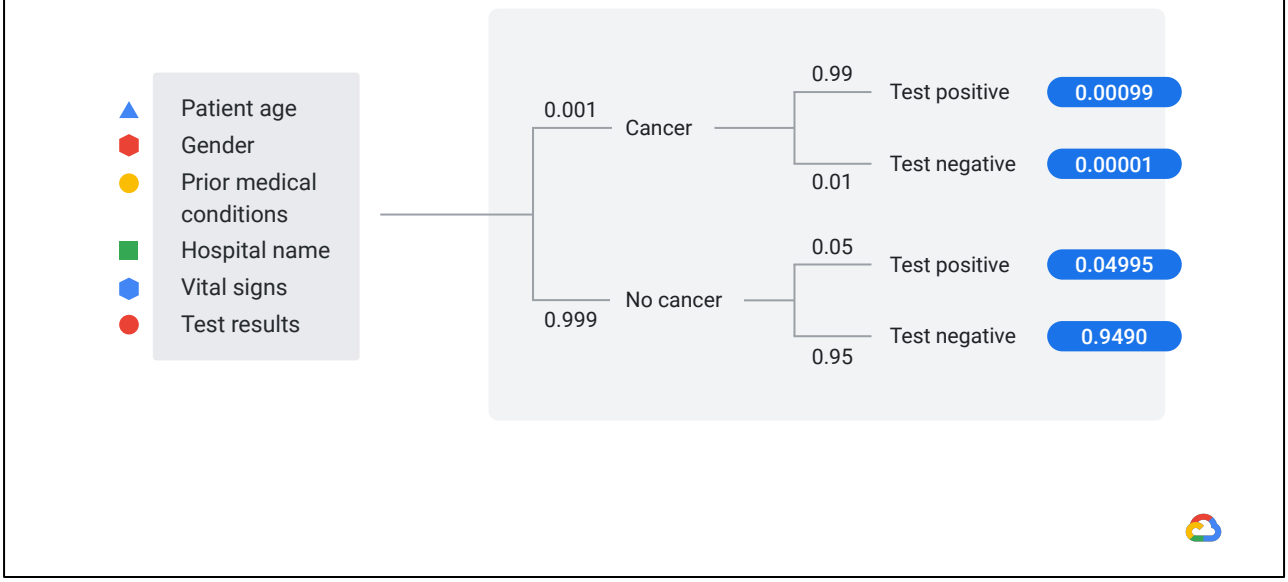
Module 02
Designing adaptable ML systems



Some decisions about data are a matter of weighing cost vs. benefit, like short-term performance goals against long-term maintainability. Others, though, are about right and wrong.

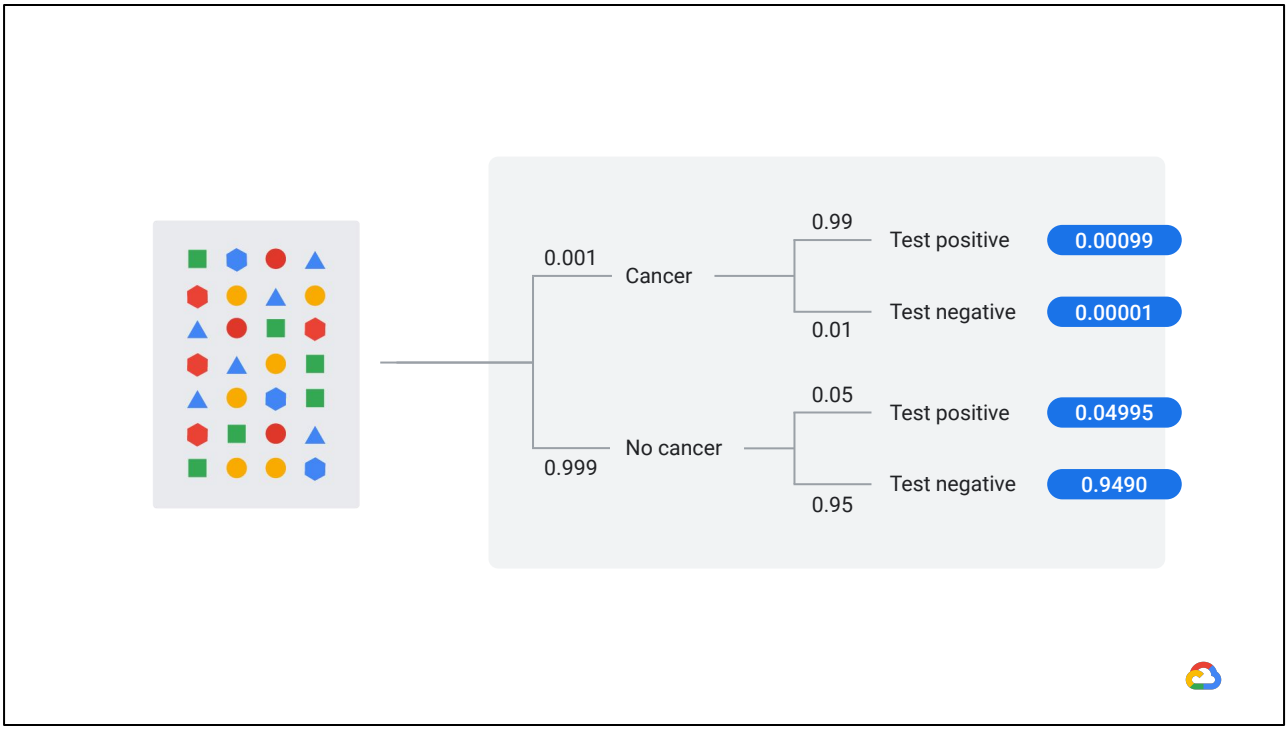


For example, let's say that you've trained a model to predict "probability a patient has cancer" from medical records and

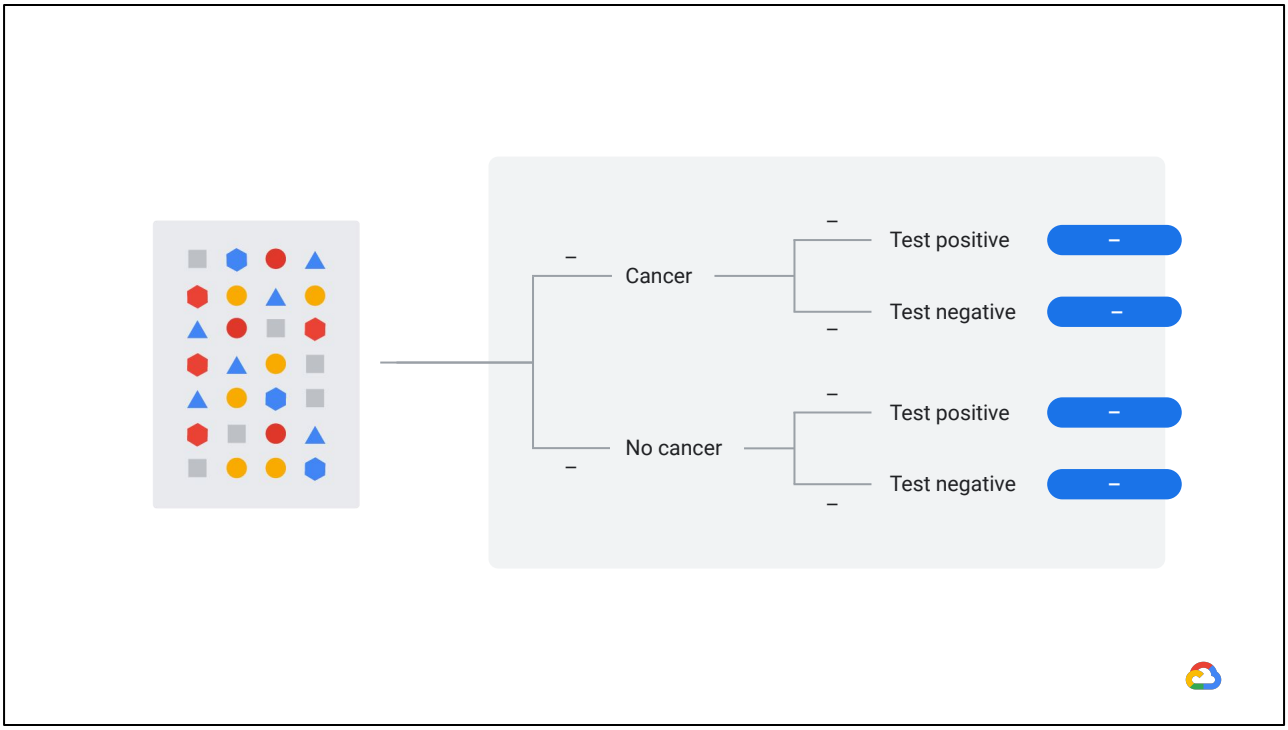


that you've selected patient age, gender, prior medical conditions, hospital name, vital signs, and test results as features. Your model had excellent performance on held-out test data but performed terribly on new patients.

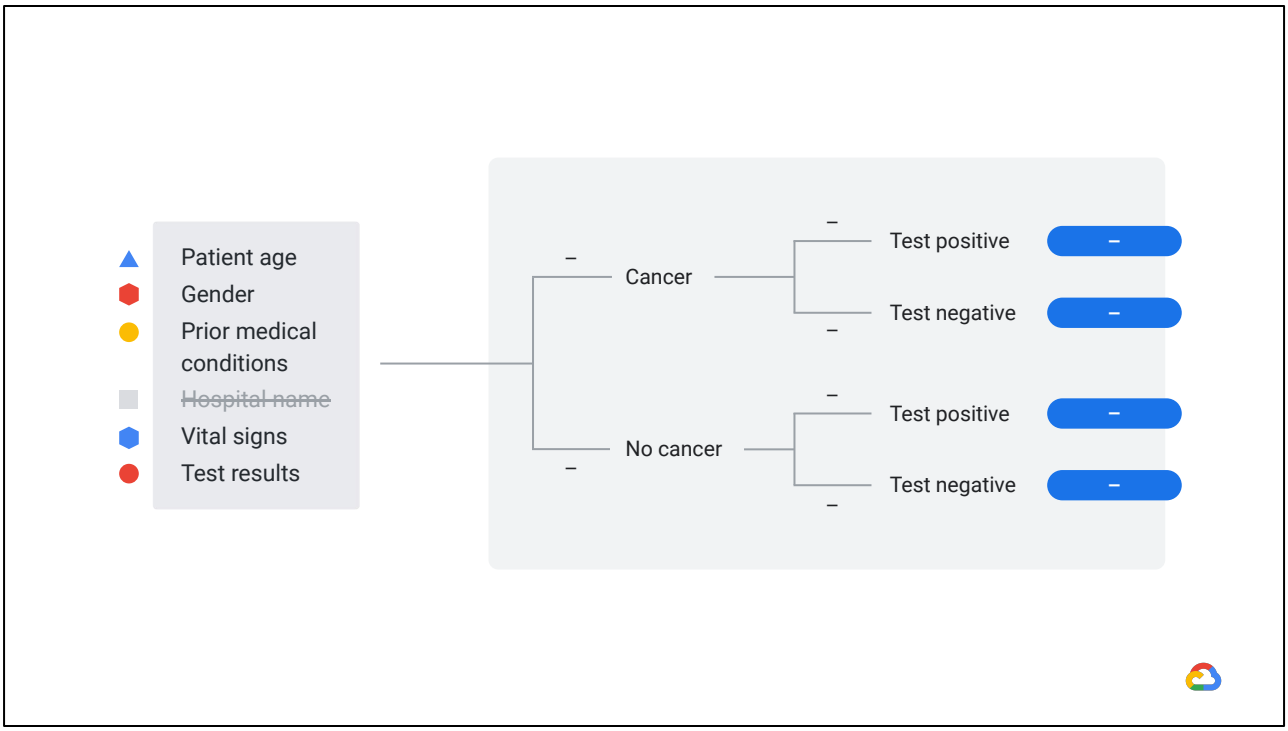
Any guesses as to why?



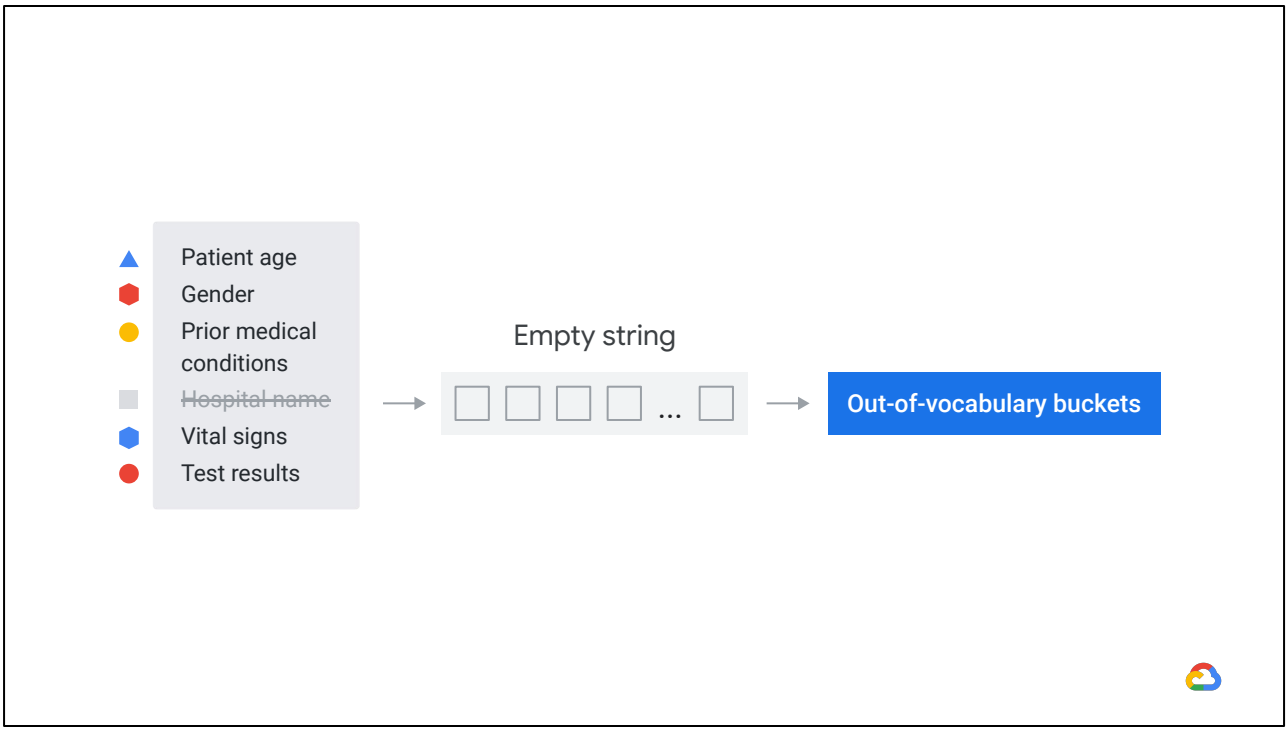
It turns out the model was trained using a feature



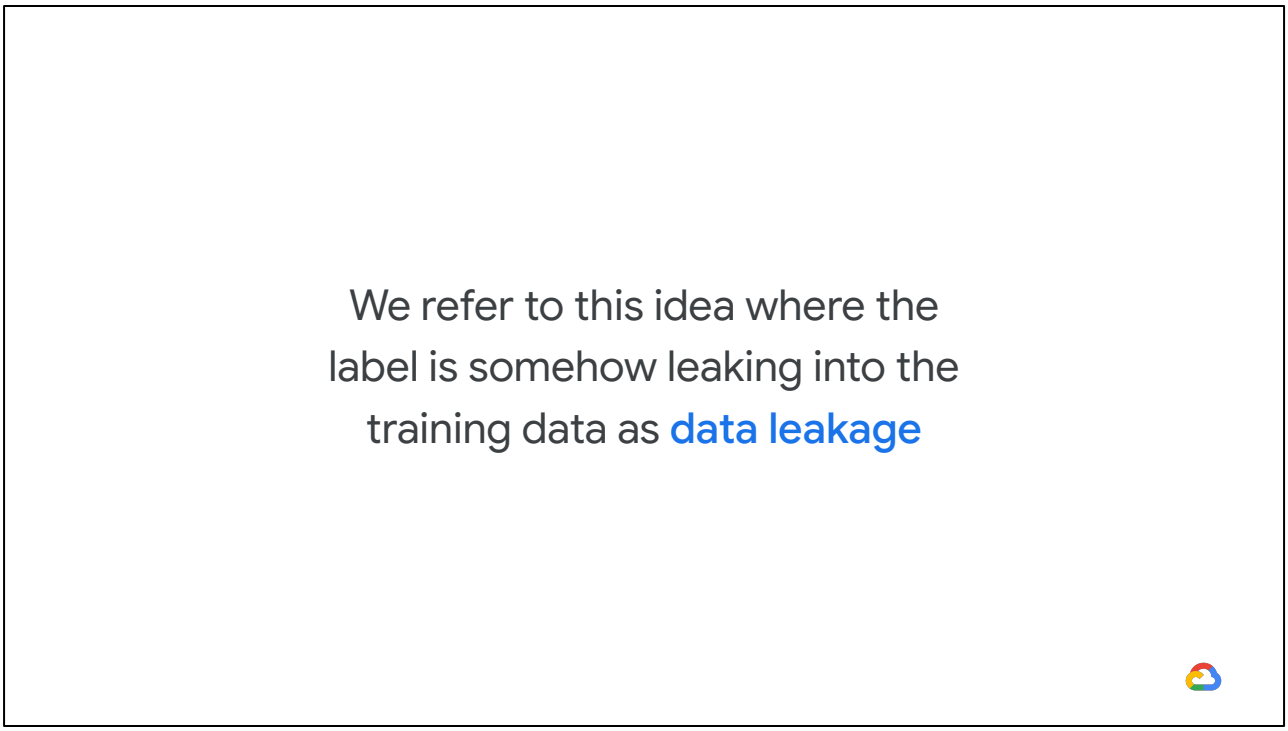
that wasn't legitimately available at decision time, and so, when the model was deployed into production, the distribution of this feature changed and it was no longer a reliable predictor.



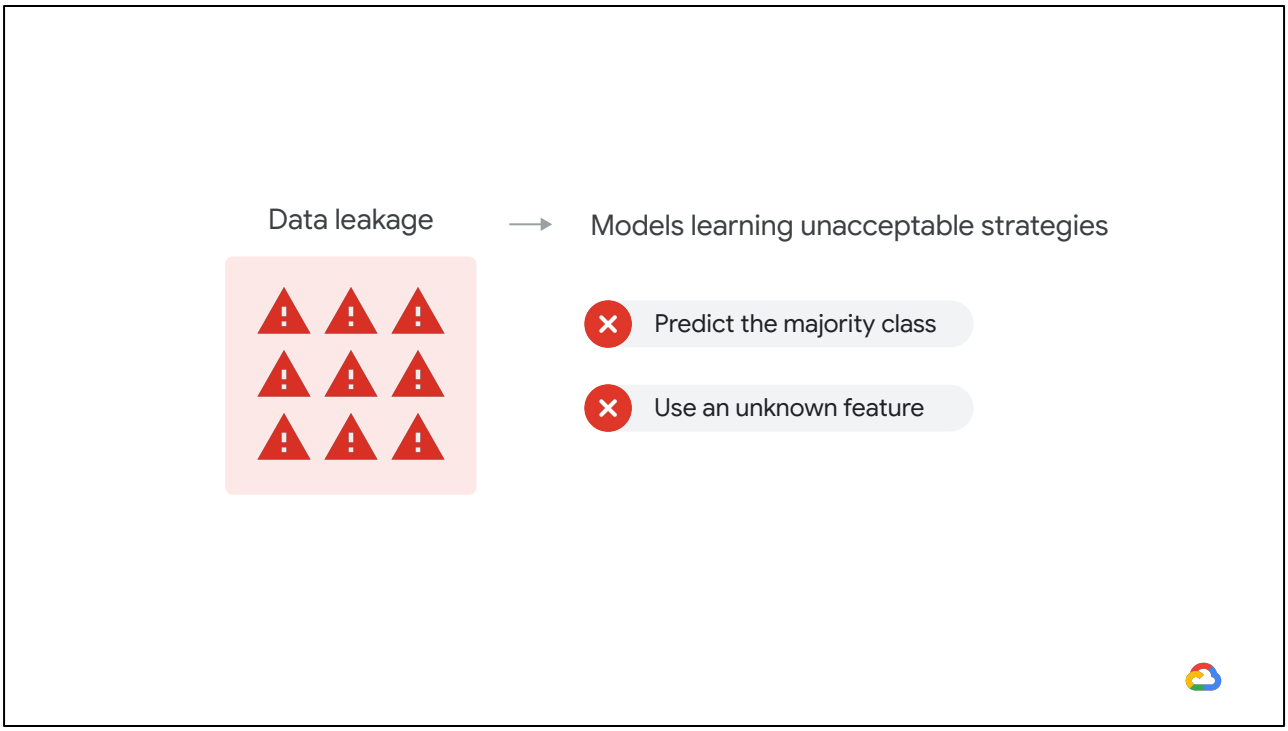
In this case, that feature was ‘hospital name’. You might think, ‘hospital name’... How could that be predictive? Well, remember that there are some hospitals that focus on diseases like cancer. So, the model learned that ‘hospital name’ was very important.



However, at decision time, this feature wasn't available to the model, because patients hadn't yet been assigned to a hospital, but rather than throwing an error, the model simply interpreted the hospital name as an empty string, which it was still capable of handling thanks to out-of-vocabulary buckets in its representations of words.



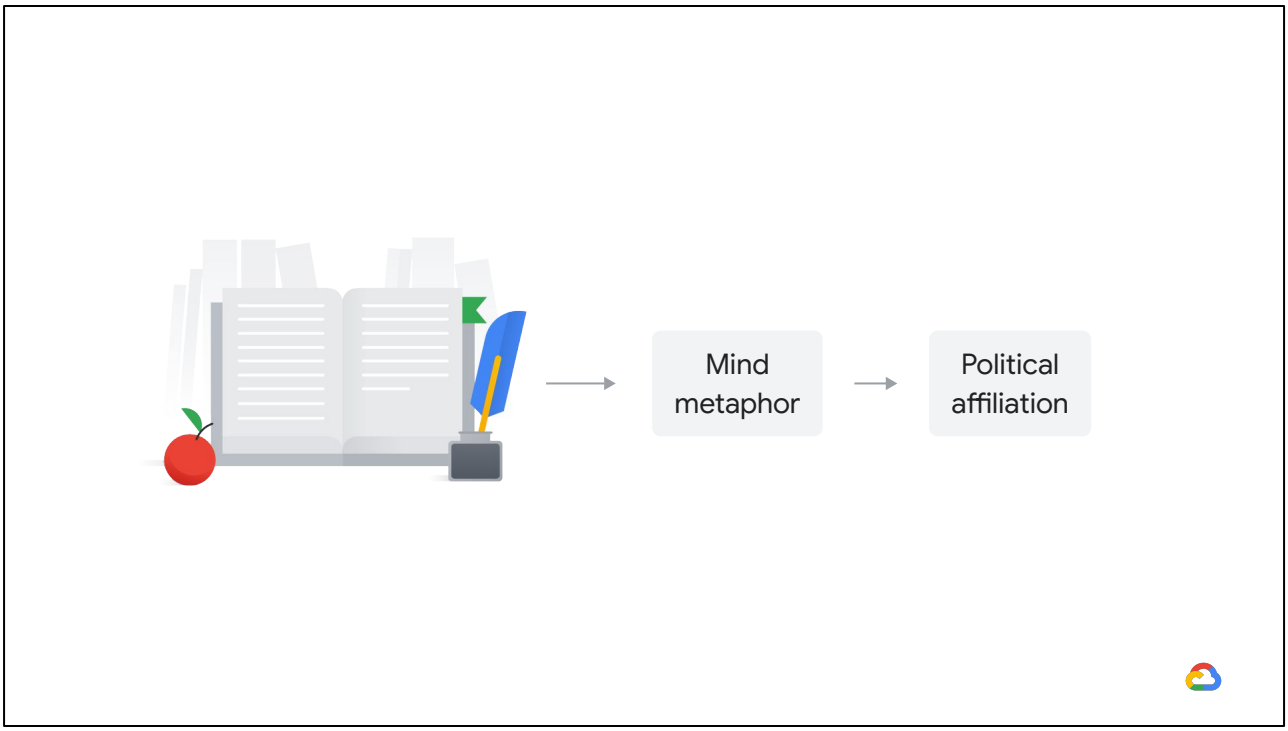
We refer to this idea where the label is somehow leaking into the training data as data leakage.



Data leakage is related to a broader class of problems we've seen before in the last specialization where we talked about models learning unacceptable strategies.

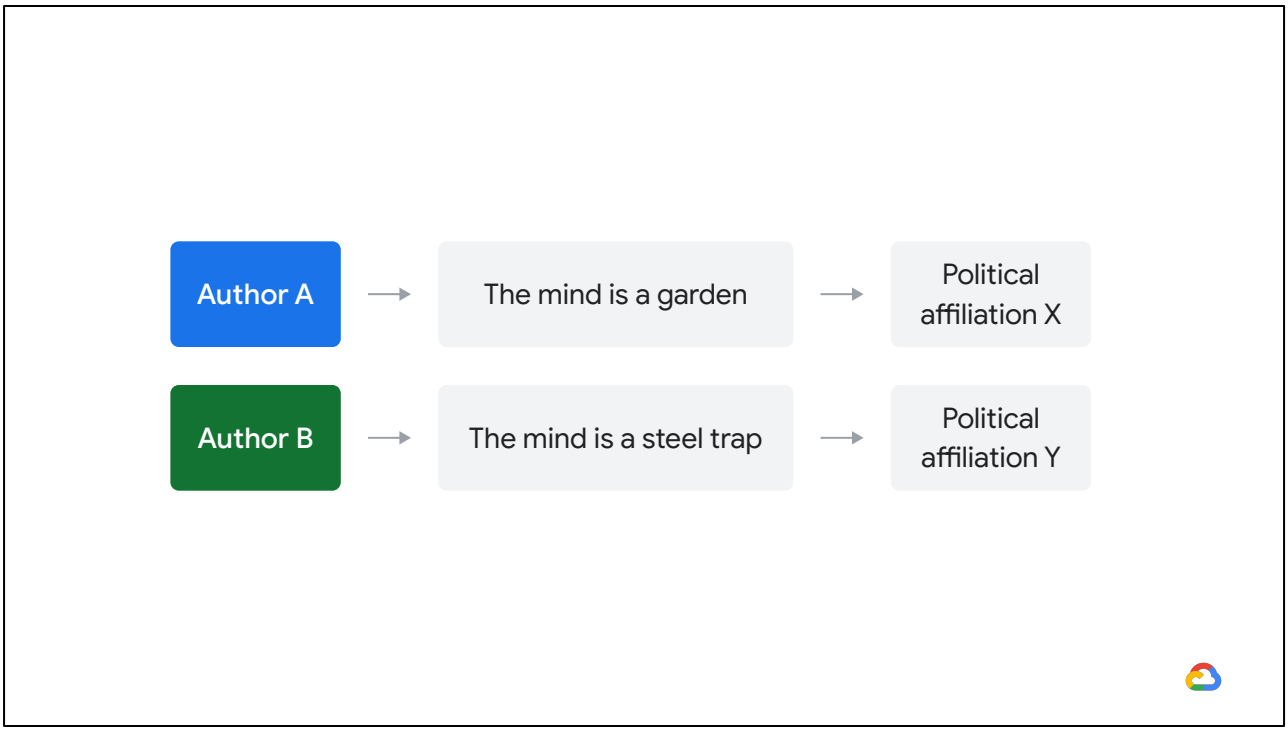
Previously, we learned that when there's class imbalances, a model might learn to predict the majority class.

In this case, the model has learned to use a feature that wouldn't actually be known and which cannot be plausibly causally related to the label.

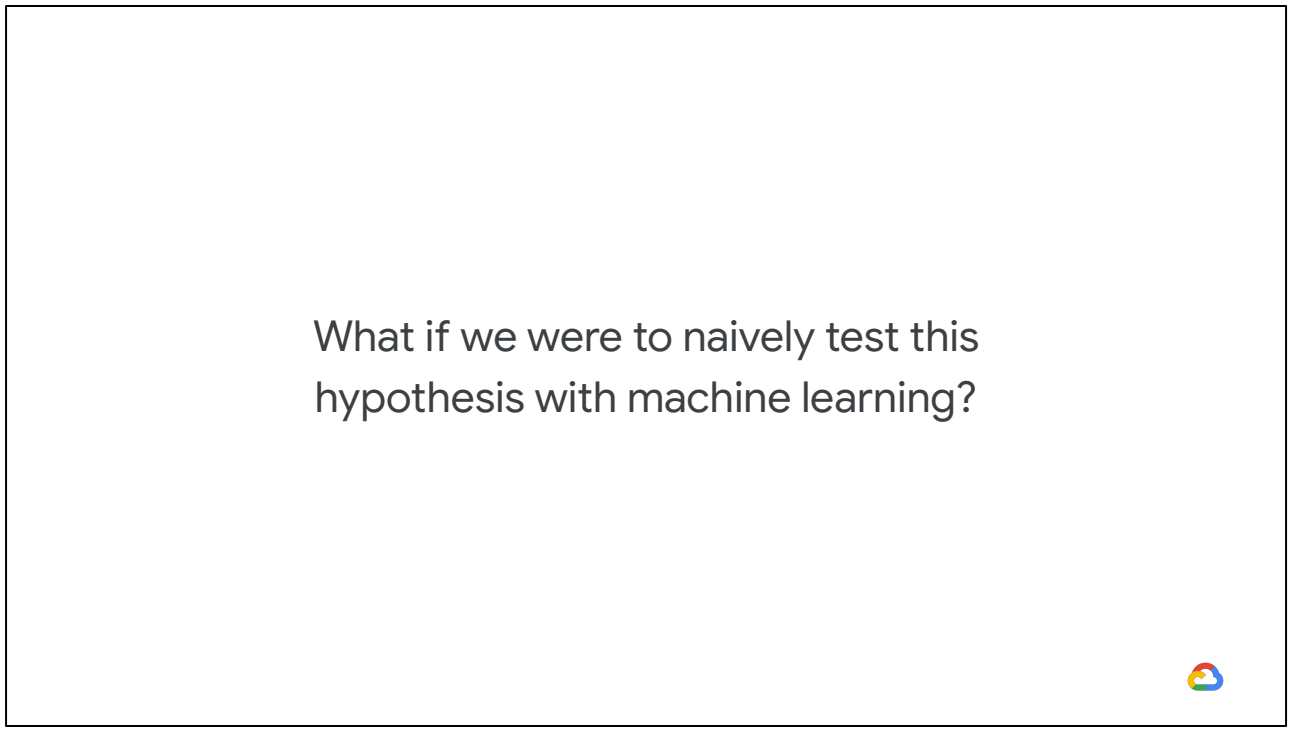


Here’s a similar case.

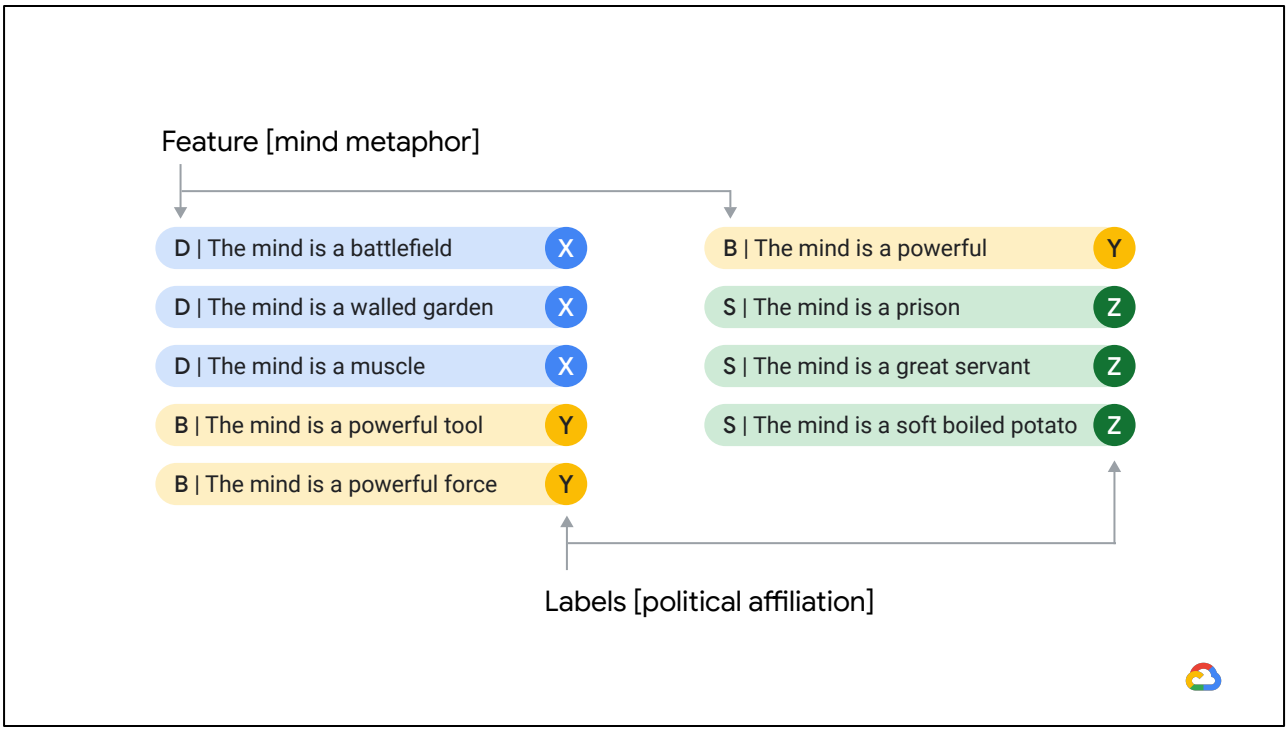
A professor of 18th century literature believed that there was a relationship between how an author thought about the mind and their political affiliation.



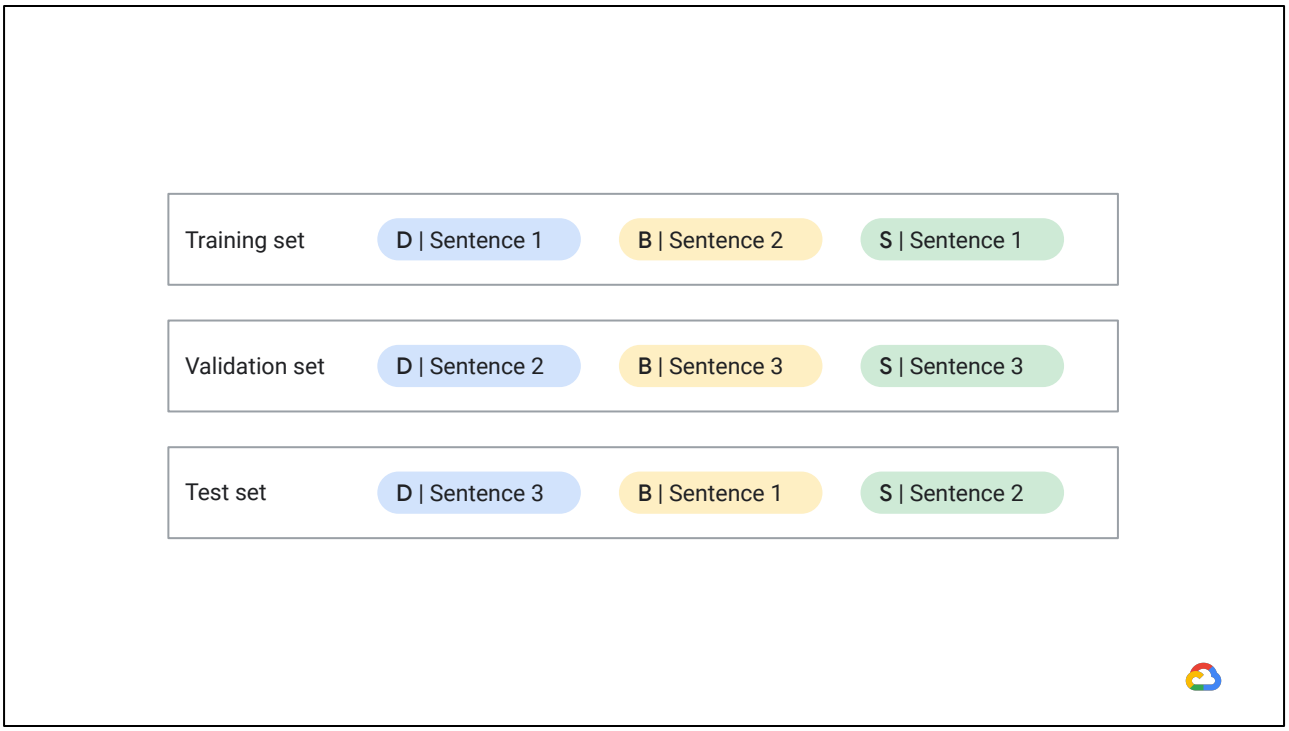
So, for example, perhaps authors who used language like “the mind is a garden” had one political affiliation and authors who used language like “the mind is a steel trap” another. Here’s what they did.



What if we were to naively test this hypothesis with machine learning? Some people tried that and they got some unexpected results.

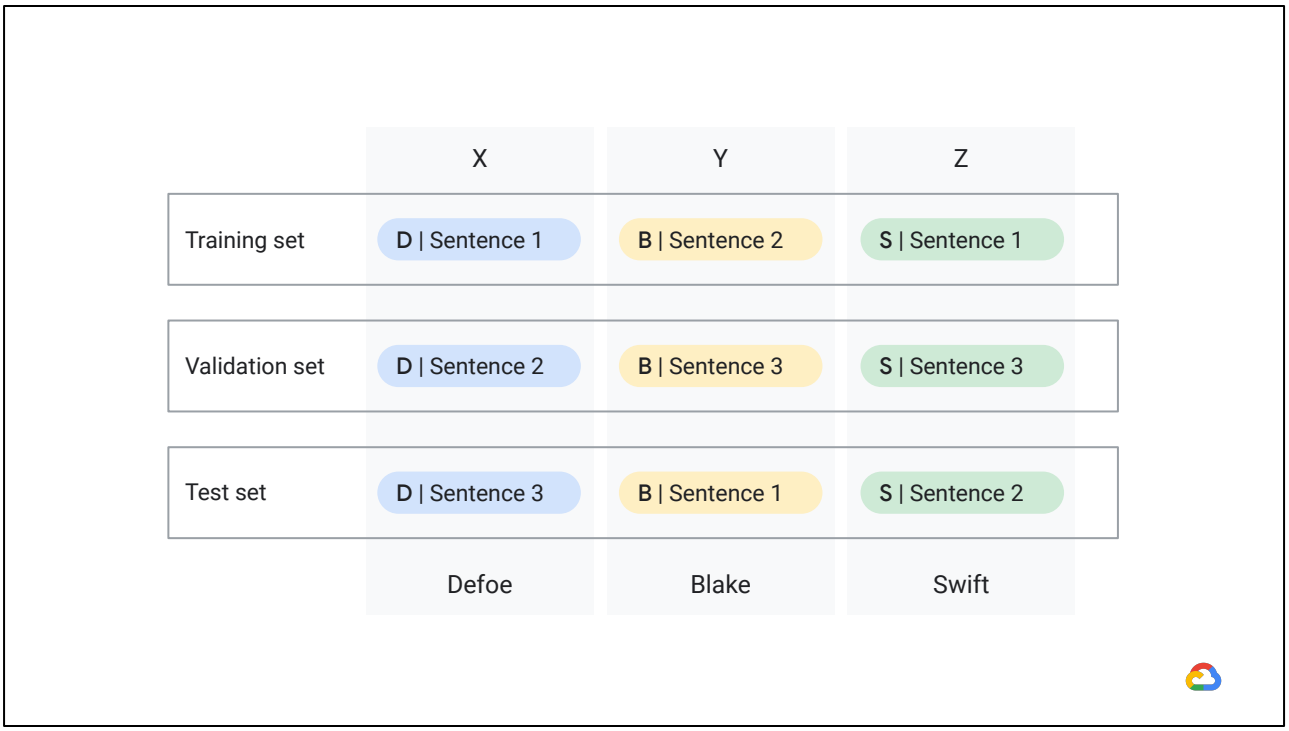


They took all of the sentences in all of the works by a number of 18th century authors, extracted just the mind metaphors and set those as their features and set those as their features and then used the political affiliations of the authors who wrote them as labels.

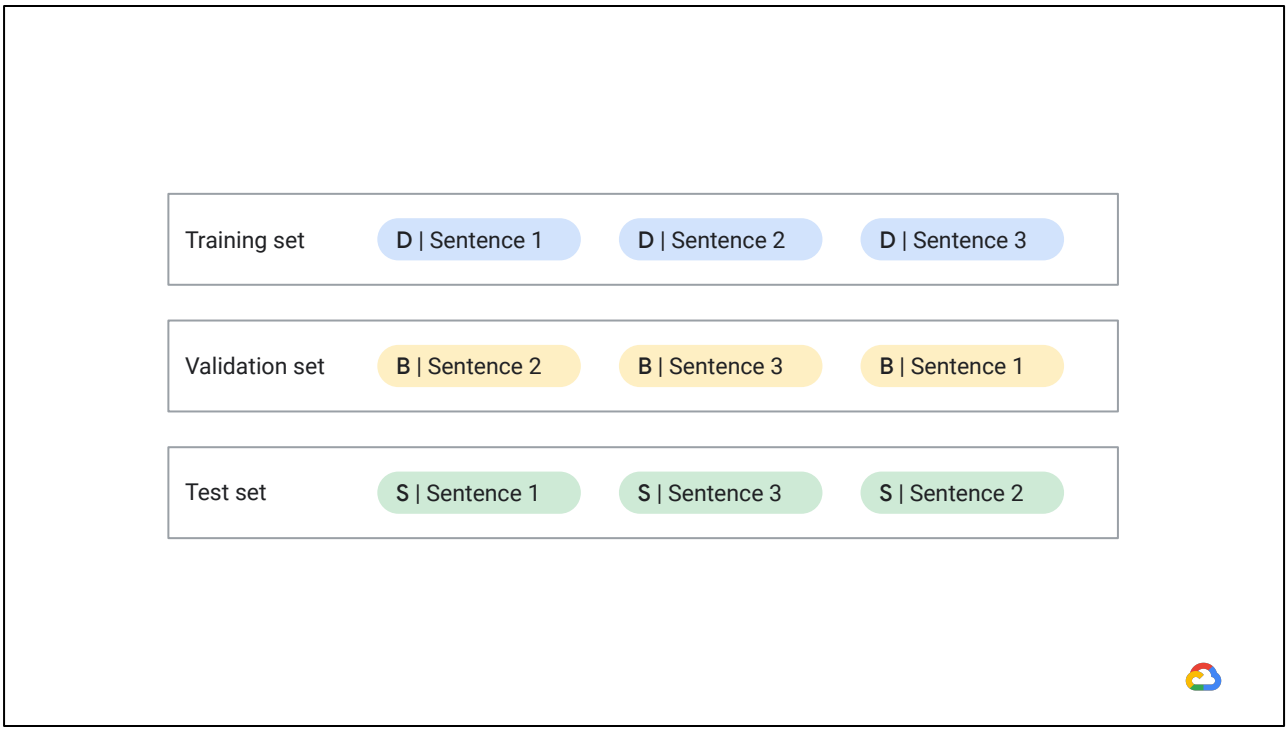


Then, they randomly assigned sentences to each of the training, validation, and test sets. And because they divided the data in this way, some sentences from each author were distributed to each of those three sets. And the resulting model was amazing! … But suspiciously amazing.

What might have gone wrong?



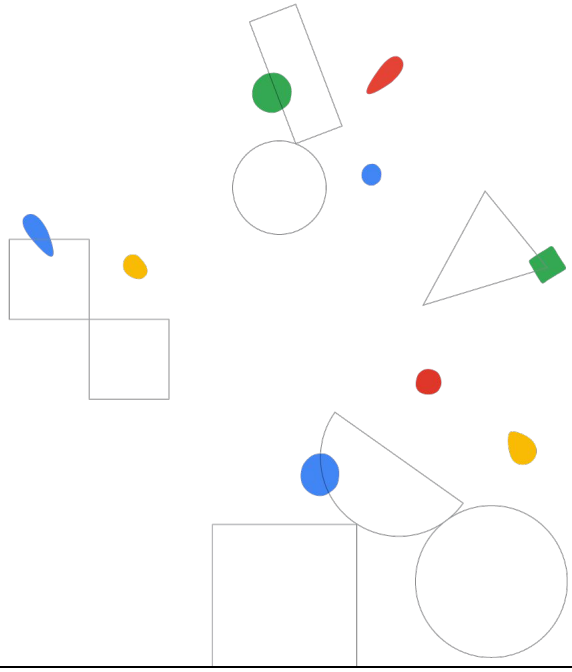
One way to think about it is that political affiliation is linked to that person. And if we wouldn't include '*person name'* in the feature set, we should not include it implicitly either.



When the researchers changed the way they partition the data and instead partitioned it by author instead of by sentence, the model's accuracy dropped to something more reasonable.

Google Cloud

System failure

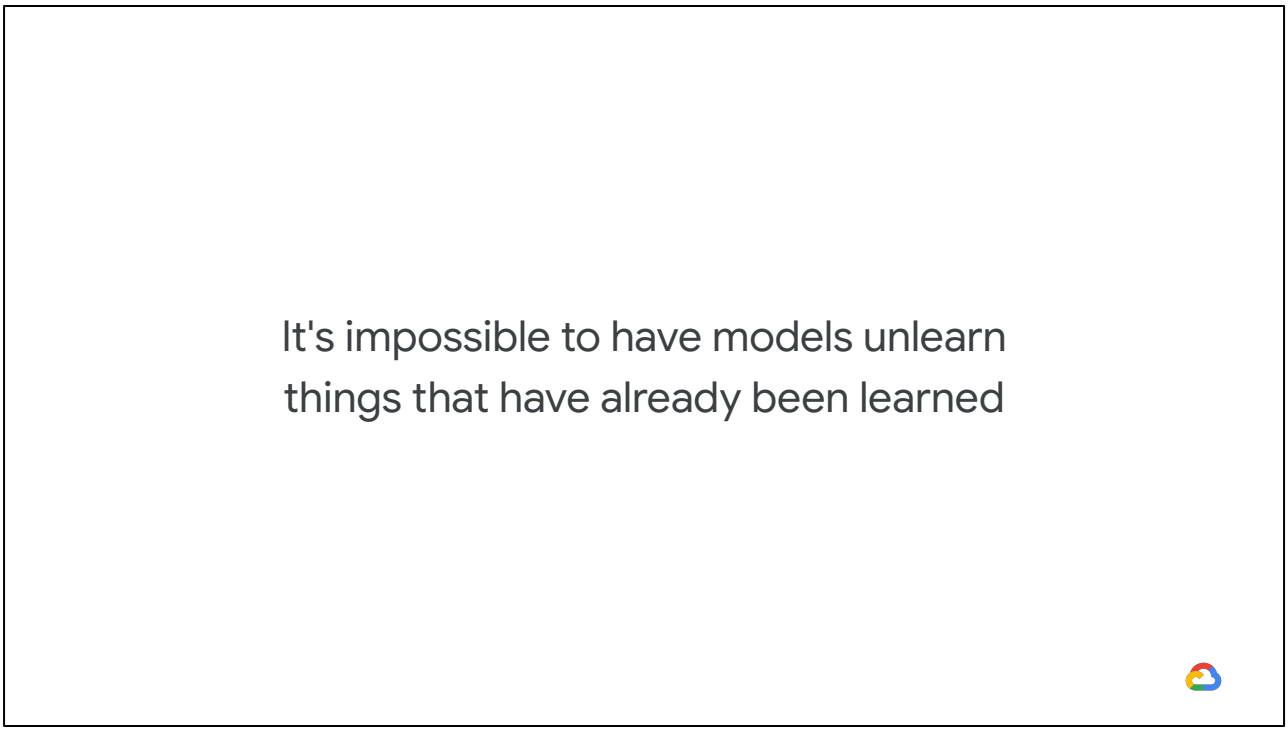


Module 02
Designing adaptable ML systems

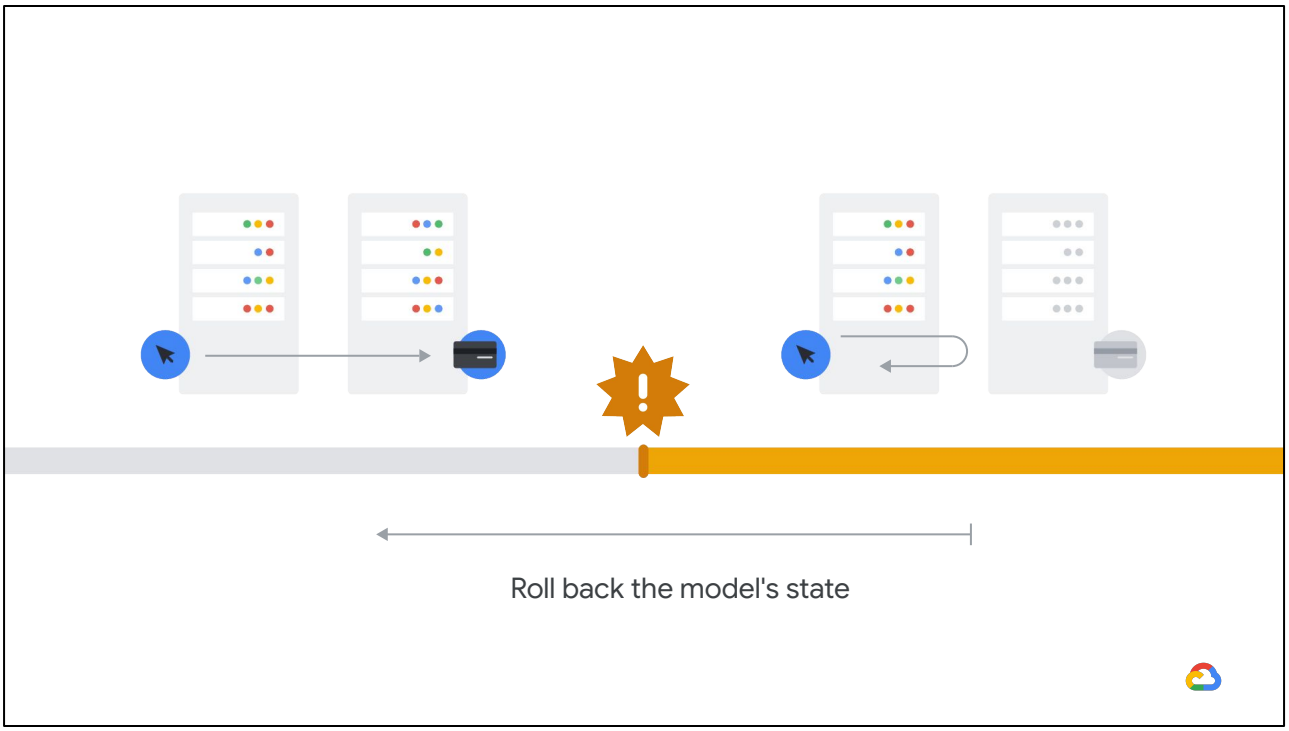
Here’s another slightly different scenario.

You've trained a product recommendation model based on users’ click and purchase behavior on your ecommerce site.

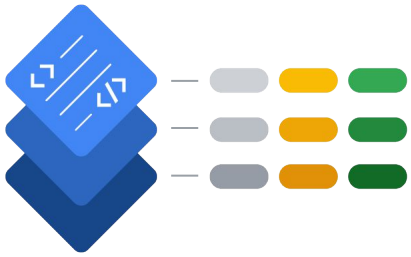
On Black Friday, your server responsible for transactions and payments goes down whilst the web server remains up and running, so the model thinks that no one who clicks is buying anything.



It's impossible to have models unlearn things that have already been learned but one thing you can do

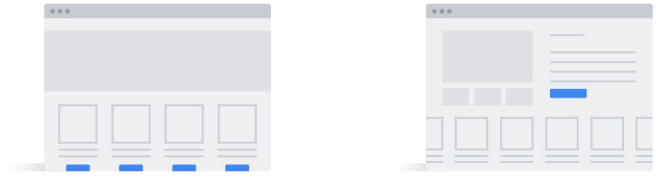


is roll back the model's state to a time prior to the data pollution.



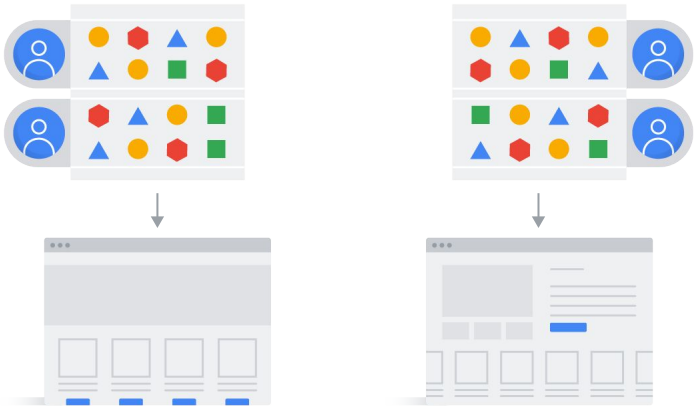
Automatically creates and saves models as well as their meta information.

Of course, in order to do this, you will need infrastructure that automatically creates and saves models as well as their meta information.

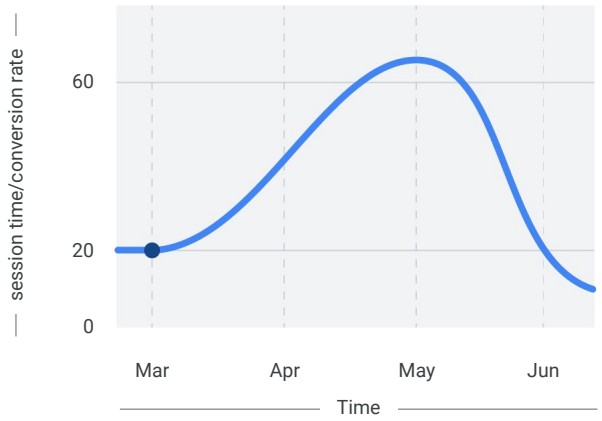


Here’s another scenario.

You’ve trained a static product recommendation model which alone will determine which products users see when they are on the home page and when they are viewing individual products.

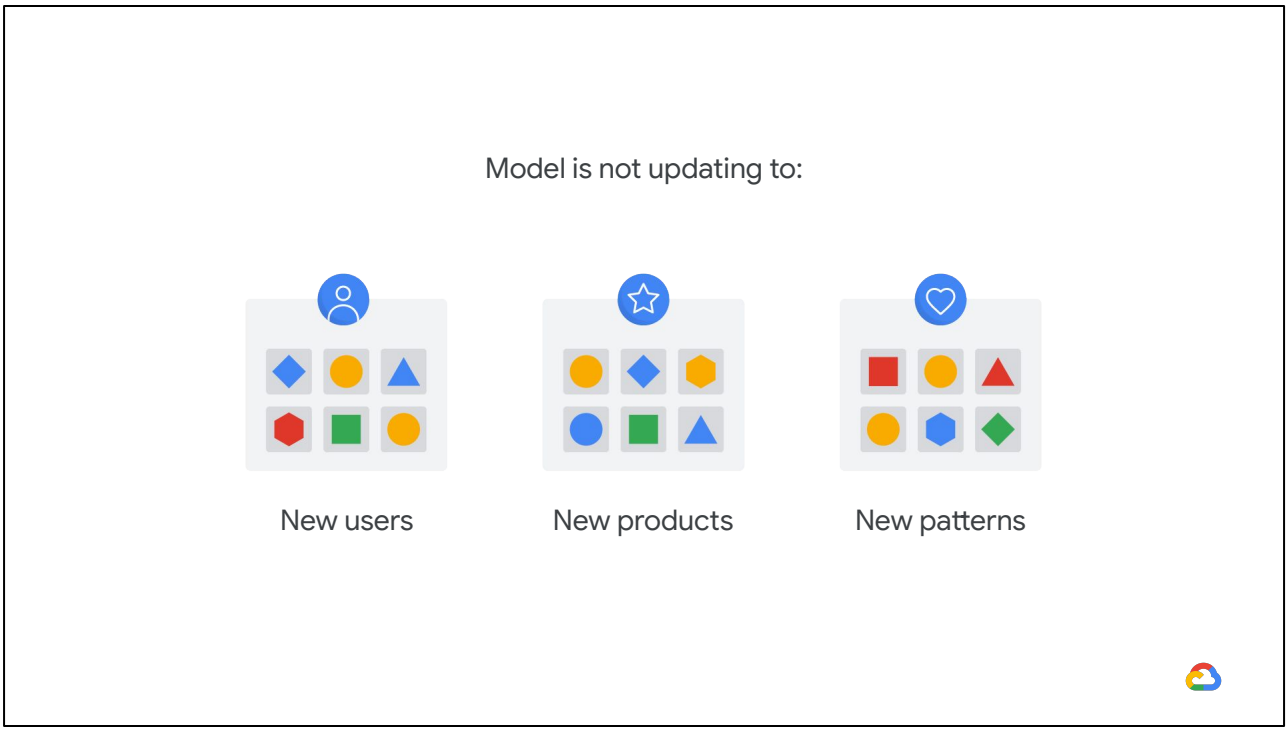


The model works by using purchasing behavior of other users.

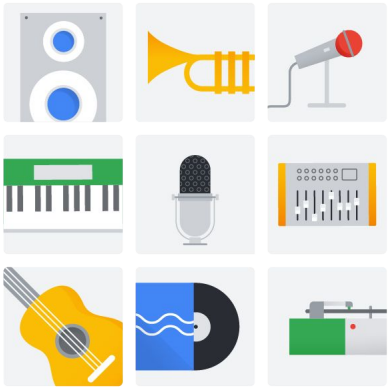


After deploying it, user session time and conversion rate initially increase. But, in the months that follow the release of the model, conversion rate and user session time steadily decline to slightly below the levels they were at before the launch of the model.

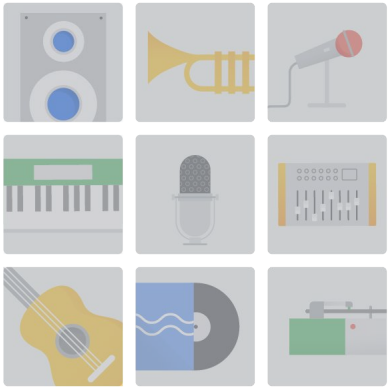
What went wrong?



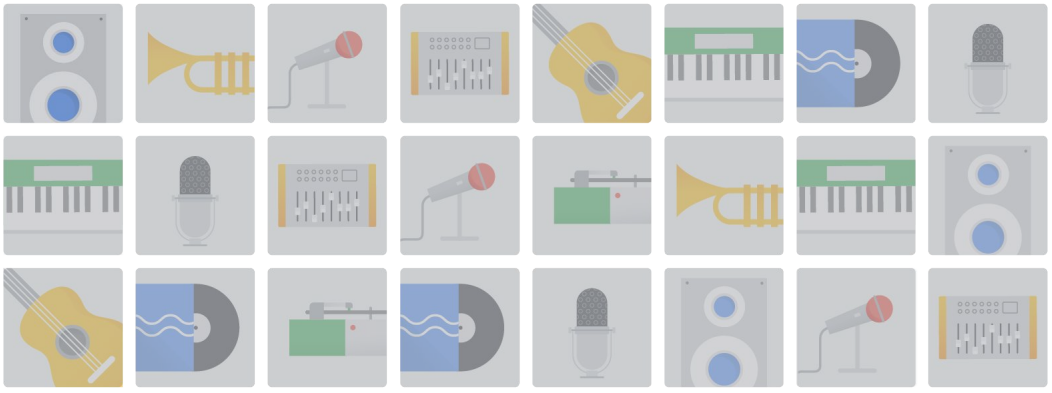
Well, your model is not updating to new users, new products, and new patterns in user preference.



Because the model only knows about

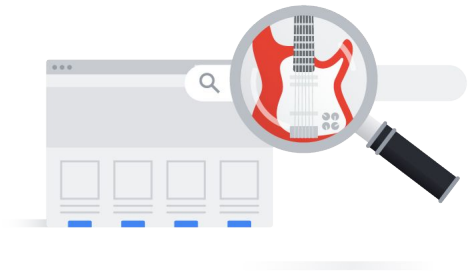


your older products,



it continues to recommend them long after they’ve fallen out of favor.

Ultimately, users simply ignored the recommendations altogether,

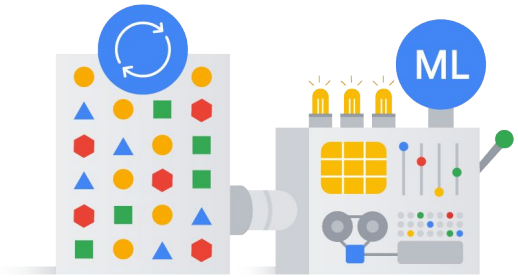


and made do with the site’s search functionality.

This “cold start” problem is common for this sort of recommendation model. We’ll talk more about recommendation systems later in the course.

Solution to a “cold start” problem

- Dynamically retrain the model
- Understand the limits of the model



The solution here is to dynamically retrain your model on newer data, and also to understand the limits of your model.

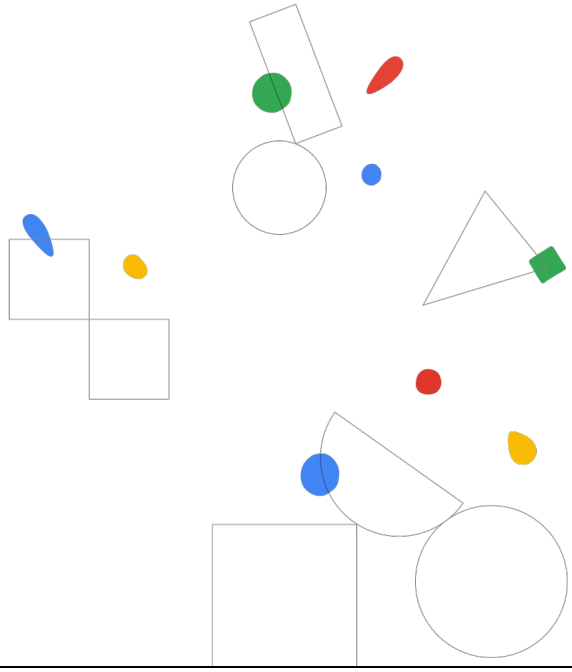
Here's one other scenario.

You've deployed a statically-trained fraud detection model and its performance starts off good but quickly degrades.

What’s gone wrong here?

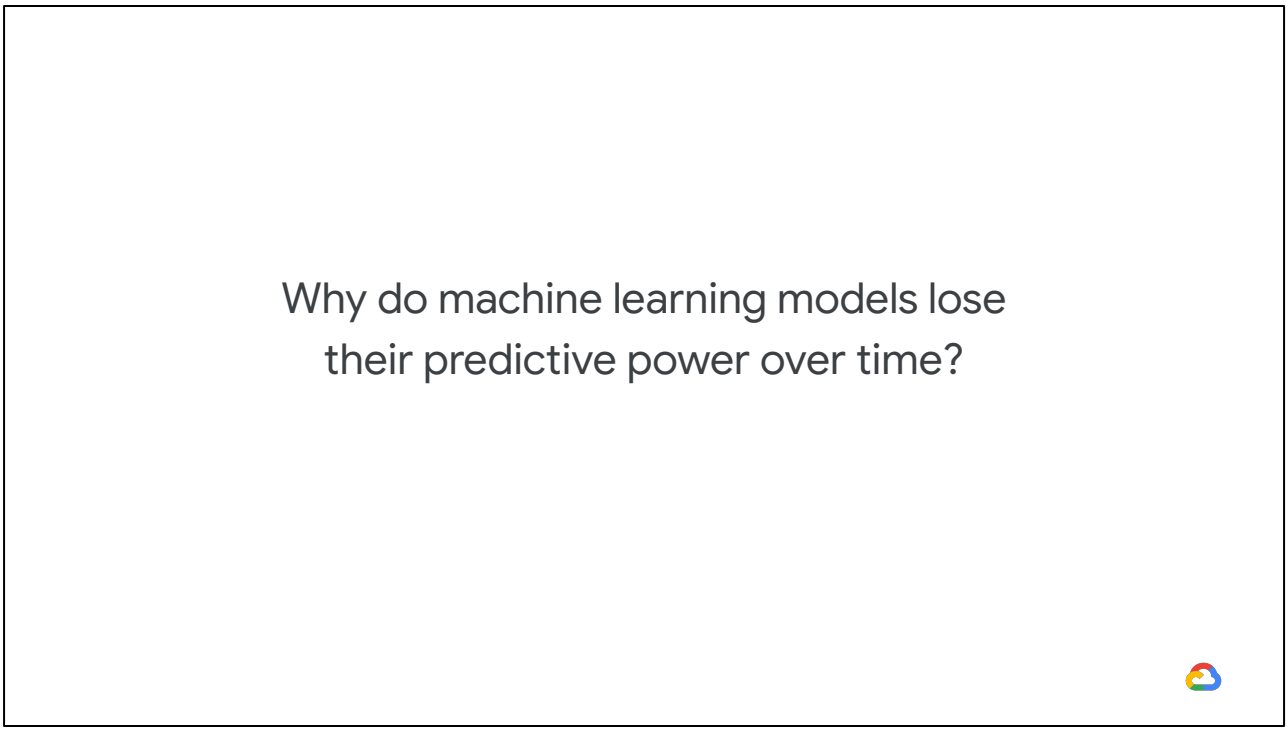
In adversarial environments, where one party is trying to beat another, it's particularly important to dynamically retrain the model, to keep up with the most recent strategies.

Concept drift

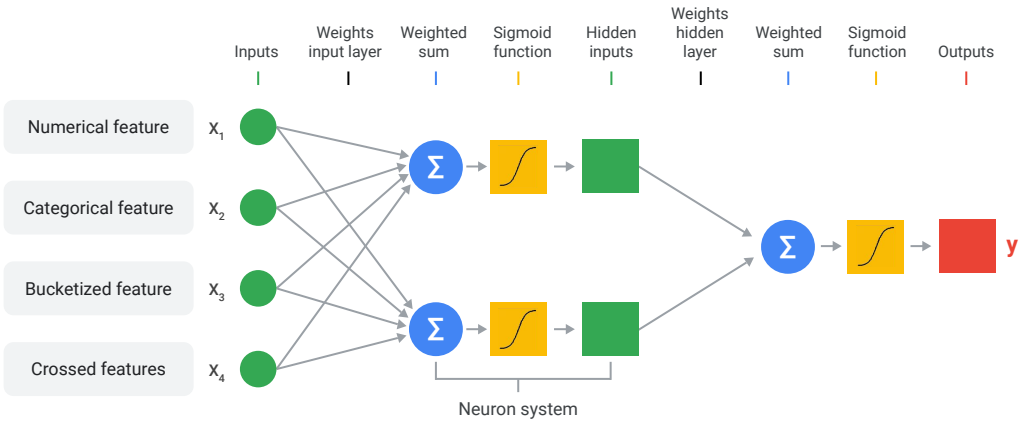


Module 02

Designing adaptable ML systems

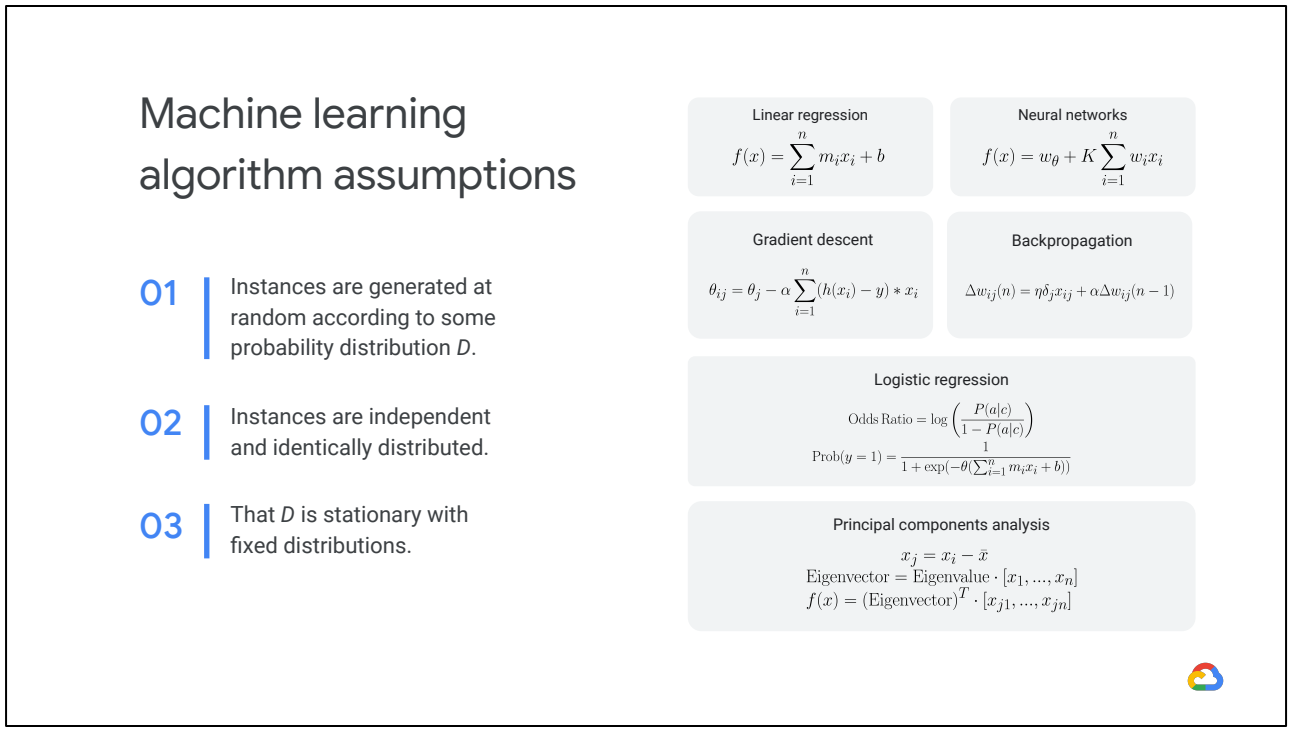


Why do machine learning models lose their predictive power over time?



You'll recall that machine learning models, such as neural networks, accept a feature vector and provide a prediction for our target variable. These models learn in a supervised fashion where a set of feature vectors with expected output is provided.

The traditional supervised learning assumes that the training and the application data come from the same distribution.



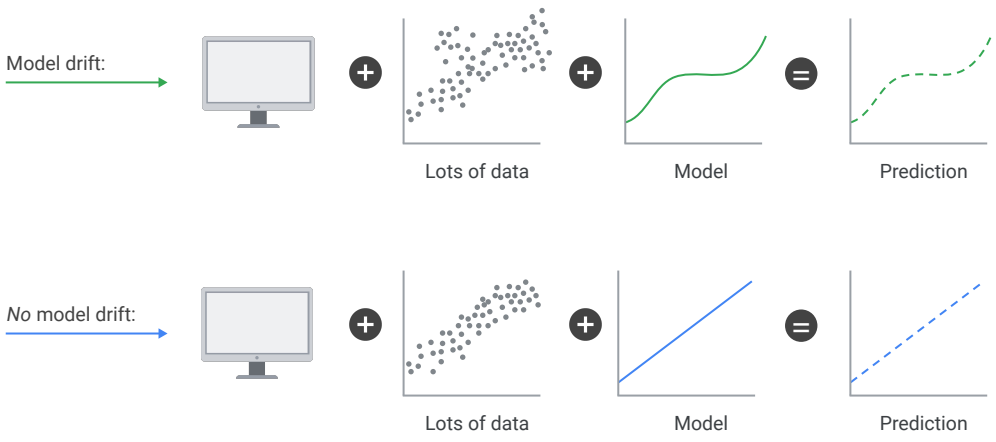
You'll also remember that traditional machine learning algorithms were developed with certain assumptions.

The first is that instances are generated at random according to some probability distribution D.

The second is that instances are independent and identically distributed.

And the third assumption is that D is stationary with fixed distributions.

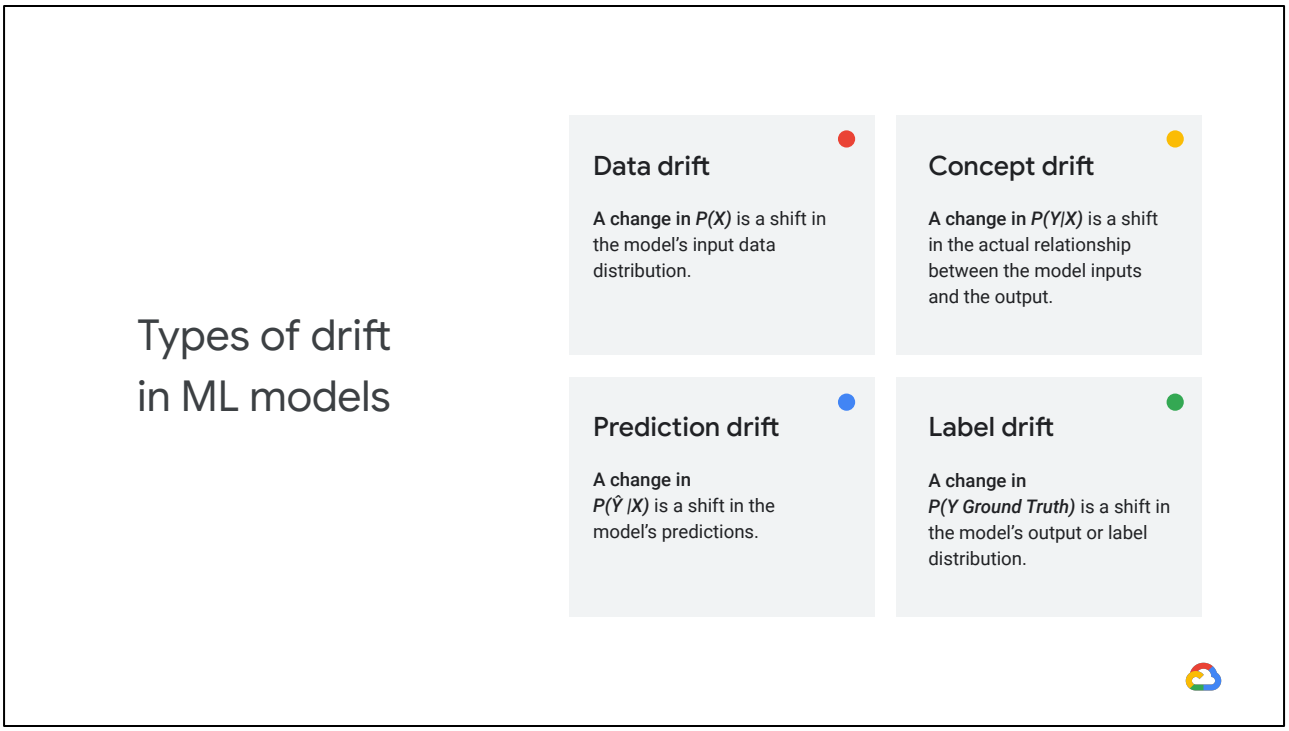
Drift is the change in an entity with respect to a baseline.



In the case of production ML models, this is the change between the real-time production data and a baseline data set, likely the training set, that is representative of the task the model is intended to perform.

If your model were running in a static environment, using static or stationary data - for example data whose statistical properties do not change, then model drift wouldn't occur and your model would not lose any of its predictive power because the data you're predicting comes from the same distribution as the data used for training.

But production data can diverge or drift from the baseline data over time due to changes in the real world.



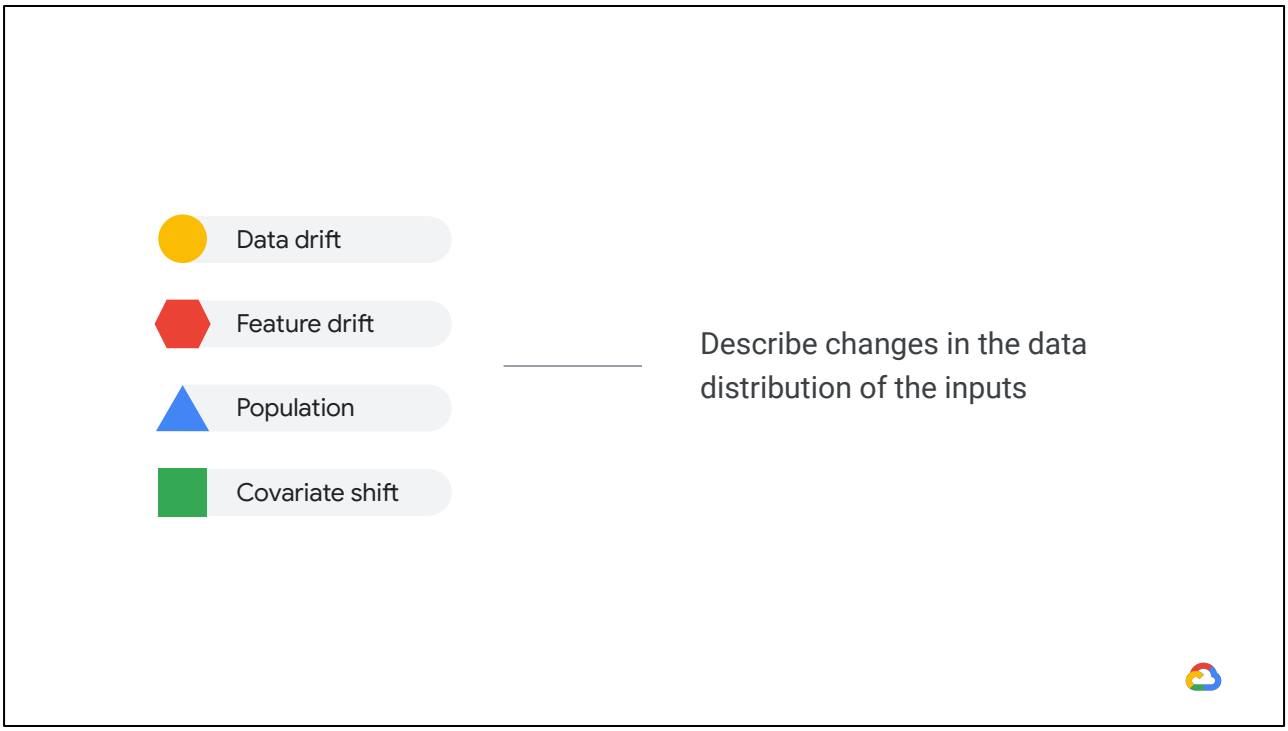
There are several types of drift in ML models...

Data Drift or change in probability of X P(X) is a shift in the model's input data distribution. For example, incomes of all applicants increase by 5%, but the economic fundamentals are the same.

Concept drift or change in probability of Y given X P(Y|X) is a shift in the actual relationship between the model inputs and the output. An example of concept drift is when macroeconomic factors make lending riskier, and there is a higher standard to be eligible for a loan. In this case, an income level that was earlier considered creditworthy is no longer creditworthy.

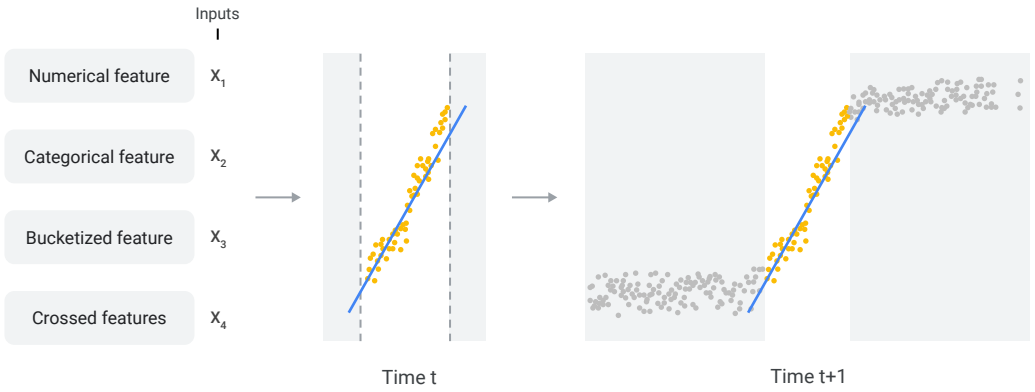
Prediction drift or change in the predicted value of Y given X P(Y hat Prediction|X) is a shift in the model's predictions. For example, a larger proportion of credit-worthy applications when your product was launched in a more affluent area. Your model still holds, but your business may be unprepared for this scenario.

Label drift or change in the predicted value of Y as your target variable P(Y Ground Truth) is a shift in the model's output or label distribution.



Data drift, feature drift, population, or covariate shift are all names to describe changes in the data distribution of the inputs. When data shift occurs, or when you observe that the model performs worse on unknown data regions, that means that the input data has changed.

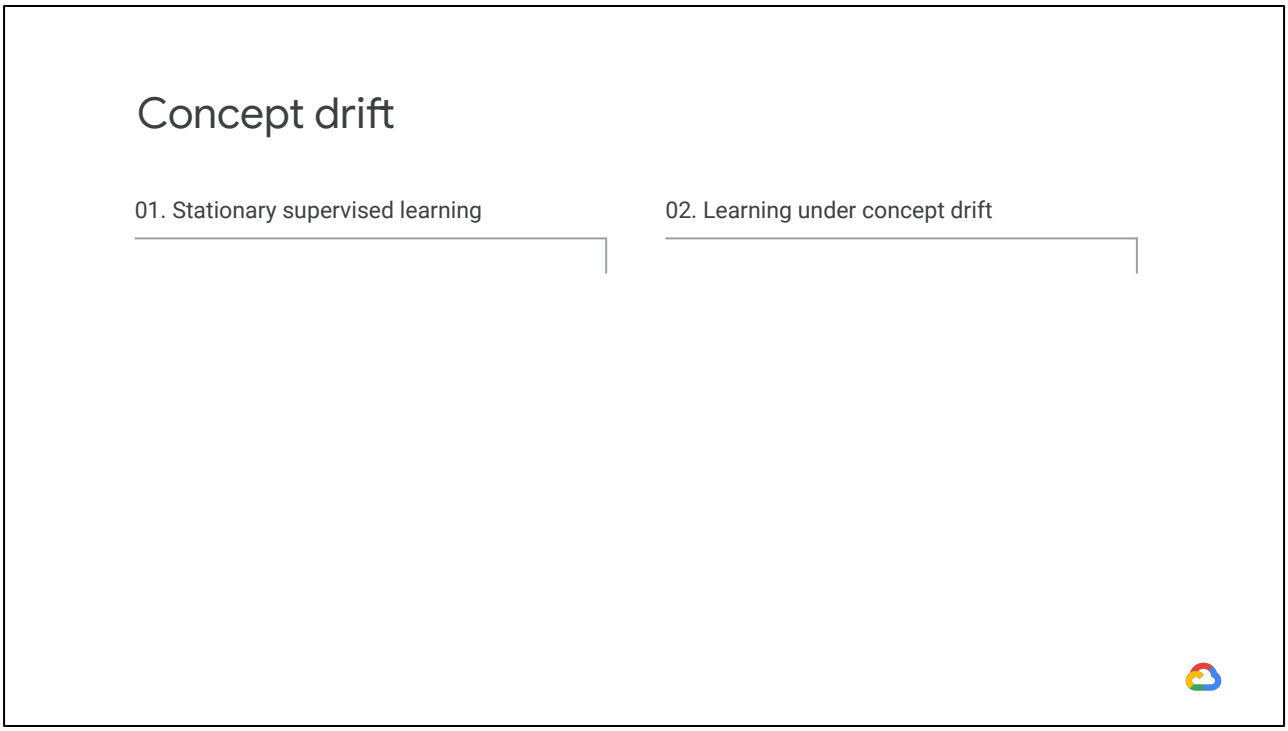
Data drift



The distribution of the variables is meaningfully different. As a result, the trained model is not relevant for this new data. It would still perform well on the data that is similar to the “old” one!

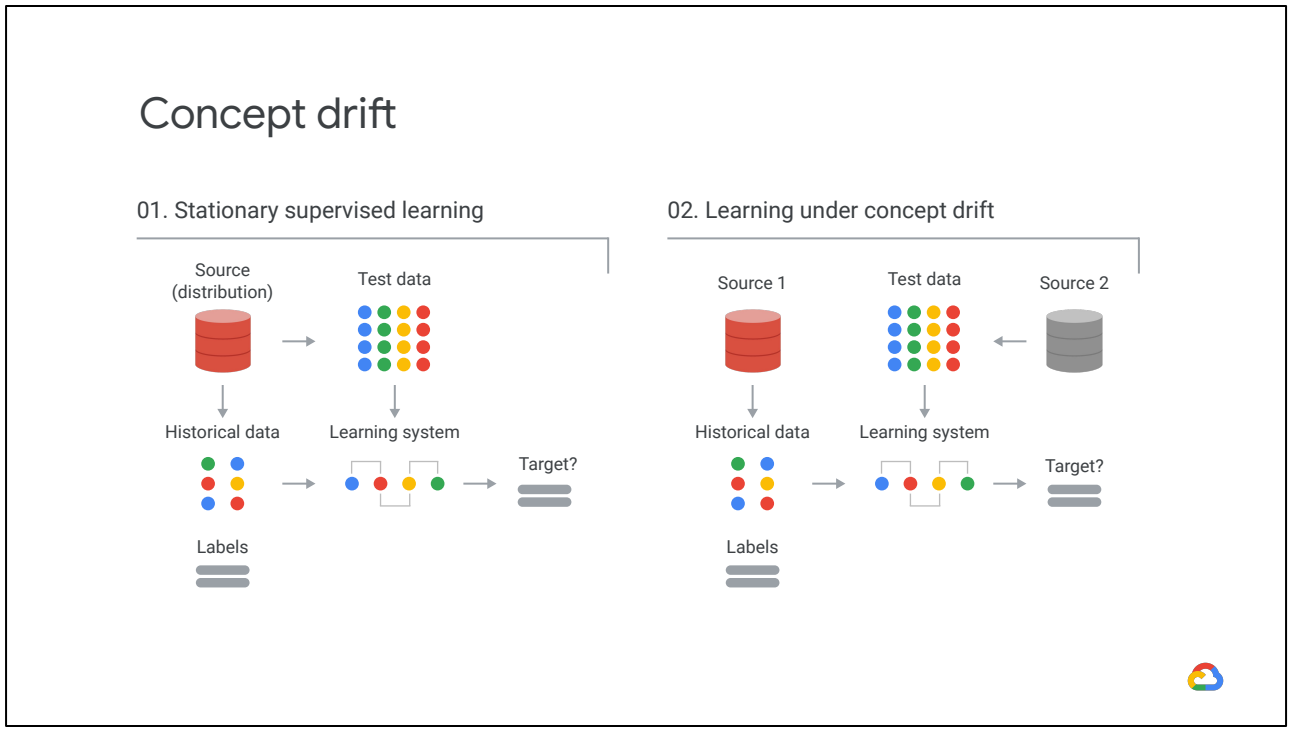
The model is fine on the “old data”, but in practical terms, it became dramatically less useful since we are now dealing with a new feature space.

Indeed, the relationships between the model inputs and outputs have changed.



In contrast, concept drift occurs when there is a change in the relationship between the input feature and the label (or target).

Let’s explore two examples of concept drift which highlight the change in the relationship between the input feature and the label.

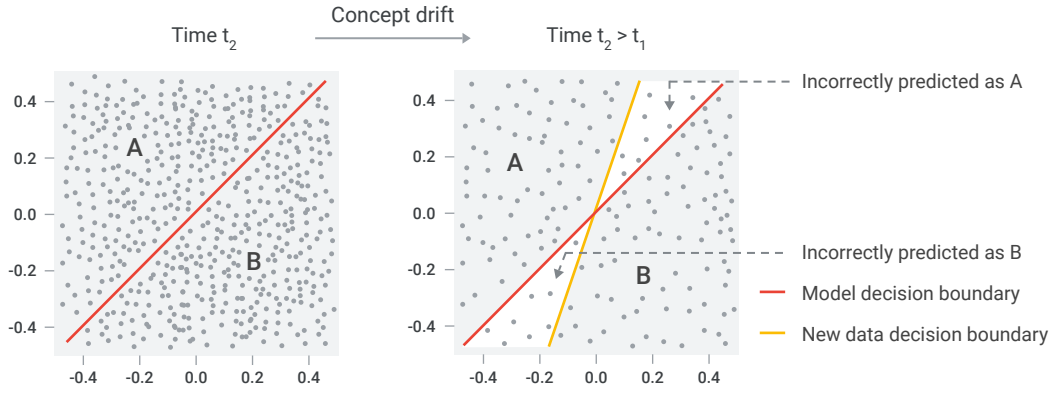


In this first example, stationary supervised learning, historical data is used to make predictions. You might recall that in supervised learning, a model is trained from historical data and that data is used to make predictions.

This second example is supervised learning under concept drift, where a new, secondary data source is ingested to provide both historical data and new data to make predictions.

This new data could be in batch or real time. Whatever the form, it's important to know that the statistical properties of the target variable may change over time. As a result, an interpretation of the data changes with time, while the general distribution of the feature input may not.

This illustrates concept drift, where the statistical properties of the class variable (the target we want to predict) changes over time.



In this supervised learning classification example, when the distribution of the label changes, it could mean that the relationship between features and labels is changing as well.

At the very least, it's likely that our model's predictions, which will typically match the distribution of the labels on the data on which it was trained, will be significantly less accurate.

Let’s take a look at one other example. This one deals with streaming data.

Data flows continuously over time in dynamic environments - particularly for streaming data, such as e-commerce, user modeling, spam emails, fraud detection, and intrusion,

Changes in underlying data occur due to changing personal interests, changes in population, or adversary activities, or they can be attributed to a complex nature of the environment.

In this example, a sensor’s measurement can drift due to a fault with the sensor or aging, changes in operation conditions or control command, and machine degradation as a result of wearing.

In these cases, the distribution of the feature inputs and the labels or targets may change - which will impact model performance and lead to model drift. There is no guarantee that future data will follow similar distributions of past data in a stream setting.

Concept drift

A probabilistic definition

Concept = $P_t(X, y)$

Concept drift = $P_t(X, y) \neq P_{t+1}(X, y)$

Concept drift occurs between times *t* and *t+1* when the distributions change.

J. Lu, A. Liu, F. Dong, F. Gu, J. Gama and G. Zhang, "Learning under Concept Drift: A Review," in *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 12, pp. 2346-2363, 1 Dec. 2019, doi: 10.1109/TKDE.2018.2876857

Accordingly, concept drift at time t can be defined as the change of joint probability of X and y at time t. The joint probability Pt(X, y) can be decomposed into two parts as the probability of X times the probability of y given X Pt(X, y) = Pt(X) × Pt(y|X).

Concept drift

A probabilistic definition

Concept = $P_t(X, y)$ — An observation, which is a feature vector with its corresponding label.

Concept drift = $P_t(X, y) \neq P_{t+1}(X, y)$

Concept drift occurs between times *t* and *t+1* when the distributions change.

J. Lu, A. Liu, F. Dong, F. Gu, J. Gama and G. Zhang, "Learning under Concept Drift: A Review," in *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 12, pp. 2346-2363, 1 Dec. 2019, doi: 10.1109/TKDE.2018.2876857

Let's be a little more careful here with the definition of concept drift. Let's use X to denote a feature vector and y to denote its corresponding label.

Of course, when doing supervised learning, our goal is to understand the relationship between X and y.

Concept drift

A probabilistic definition

Concept = $P_t(X, y)$ A concept, which is the (joint probability) distribution of an observation.

Concept drift = $P_t(X, y) \neq P_{t+1}(X, y)$

Concept drift occurs between times *t* and *t+1* when the distributions change.

J. Lu, A. Liu, F. Dong, F. Gu, J. Gama and G. Zhang, "Learning under Concept Drift: A Review," in *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 12, pp. 2346-2363, 1 Dec. 2019, doi: 10.1109/TKDE.2018.2876857

We will define a concept as a description of the distribution of our observations. More precisely, you can think of this as a joint probability distribution of our observations.

However, this concept could depend on time! Otherwise concept drift would be a non-issue, right?

Concept drift

A probabilistic definition

$$\text{Concept} = P_t(X, y)$$

$$\text{Concept drift} = P_t(X, y) \neq P_{t+1}(X, y)$$

P(X,y) written for a certain time is $P_t(X,y)$.

Concept drift occurs between times *t* and *t+1* when the distributions change.

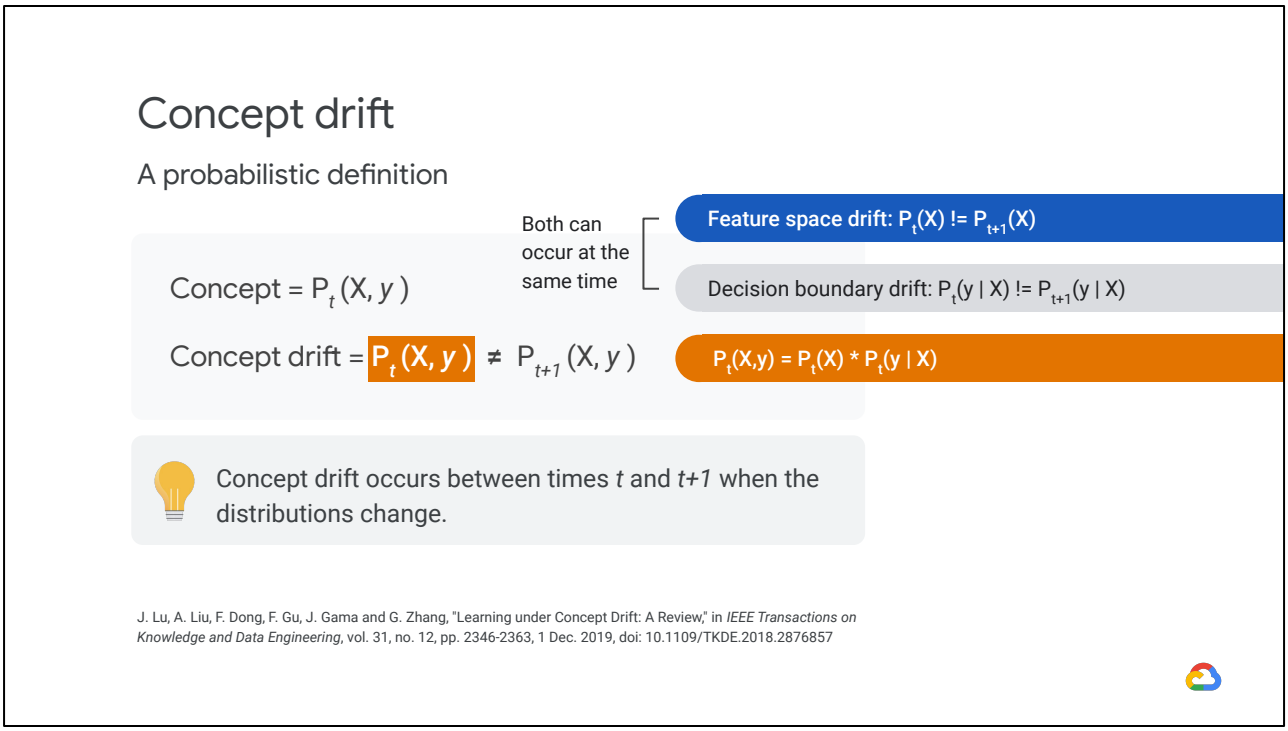
J. Lu, A. Liu, F. Dong, F. Gu, J. Gama and G. Zhang, "Learning under Concept Drift: A Review," in *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 12, pp. 2346-2363, 1 Dec. 2019, doi: 10.1109/TKDE.2018.2876857

We'll use the notation probability of X and y at time t P_t(X,y) when we want to consider the probability of X and y P(X,y) at a specific time.

Concept drift

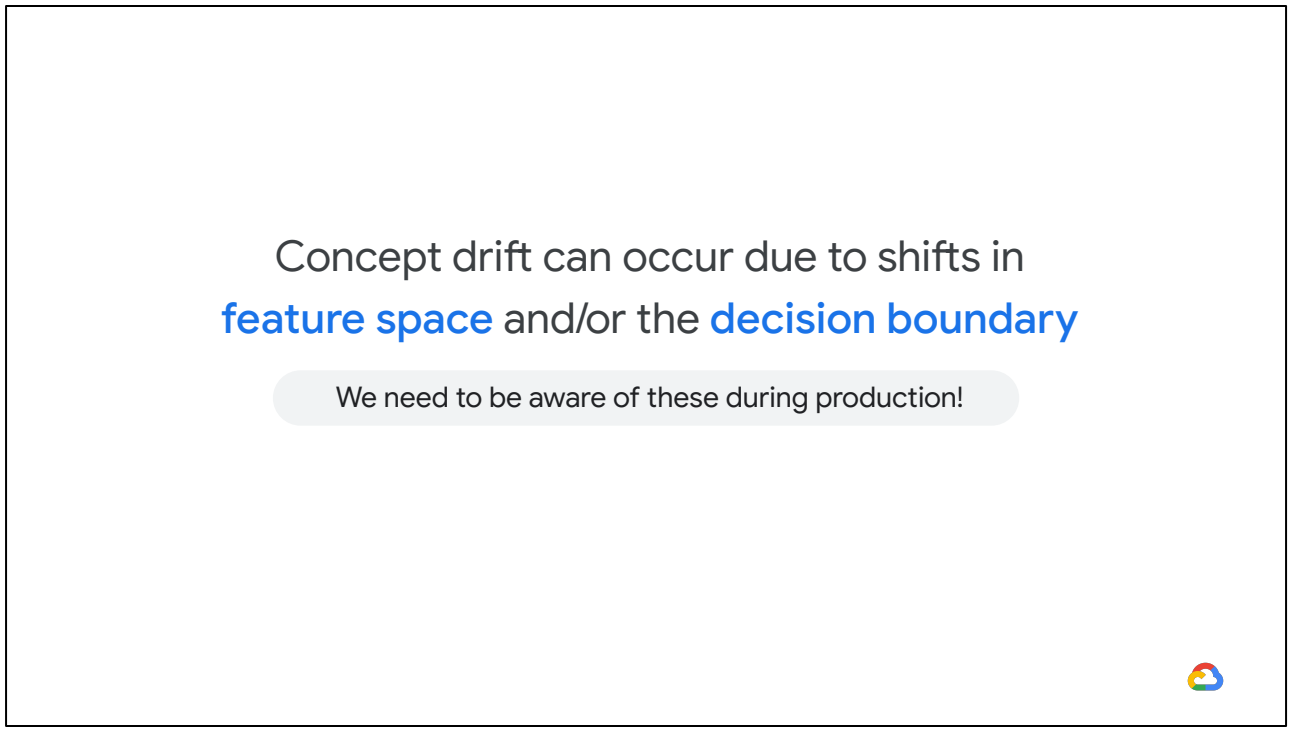
occurs when the distribution of our observations shifts over time, or that the joint probability distribution changes

Now it’s easy to give a more rigorous description of concept drift. Simply put, concept drift occurs when the distribution of our observations shifts over time, or that the joint probability distribution we mentioned before changes.



We can break down this distribution into two parts using properties of joint distributions.

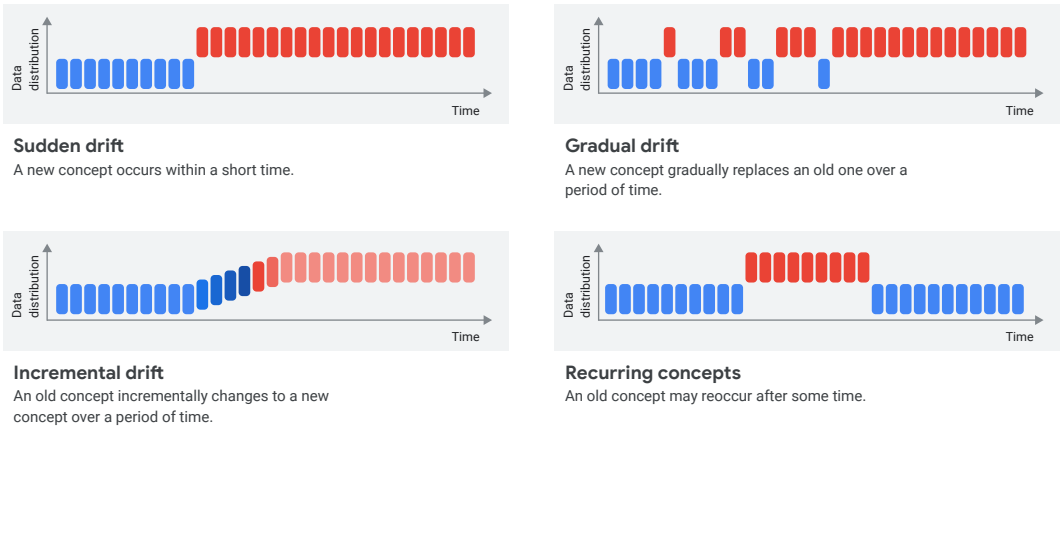
- First we have the distribution of the feature space, probability of X P(X), and then what we can think of as a description of our decision boundary, the probability of Y given X P(y | X) (y given X).
- If the drift is occurring for the decision boundary, then we call this decision boundary drift.
- Likewise, if the drift is occurring for the feature space, we call this feature space drift.
- Of course, both can be happening at the same time, and this can make it complicated to understand where the changes are happening!



OK, we've veered off in a little bit more of a technical direction here, so what's the point we're trying to make?

Concept drift can occur due to shifts in the feature space and / or the decision boundary, so we need to be aware of these during production.

If the data is changing, or if the relationship between the features and the label is changing, this is going to cause issues with our model.



There are different types of concept drift. Let's wrap up this video by taking a quick look at four of them:

In sudden drift a new concept occurs within a short time.

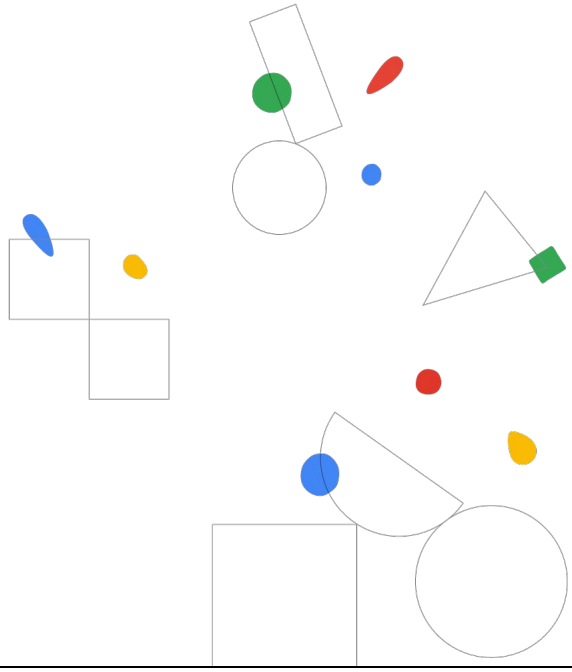
In gradual drift a new concept gradually replaces an old one over a period of time.

In incremental drift an old concept incrementally changes to a new concept over a period of time.

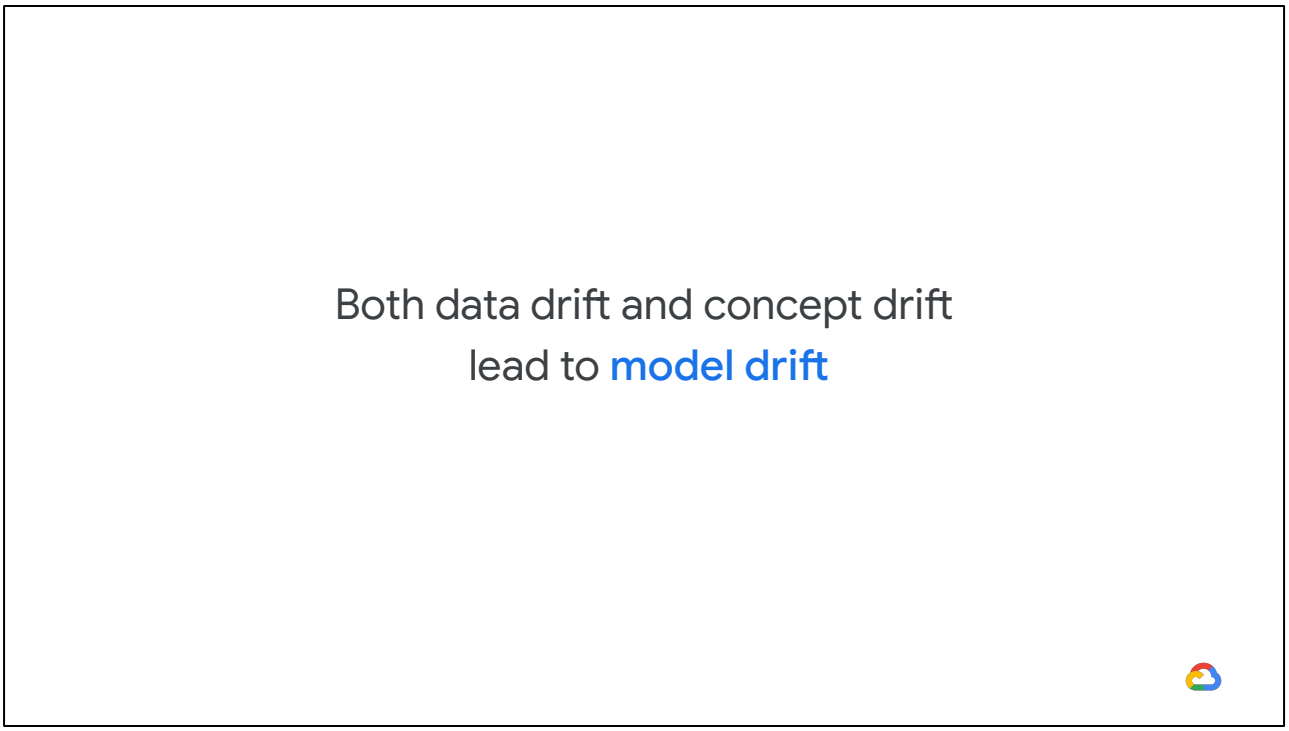
And in recurring concepts an old concept may reoccur after some time.

Google Cloud

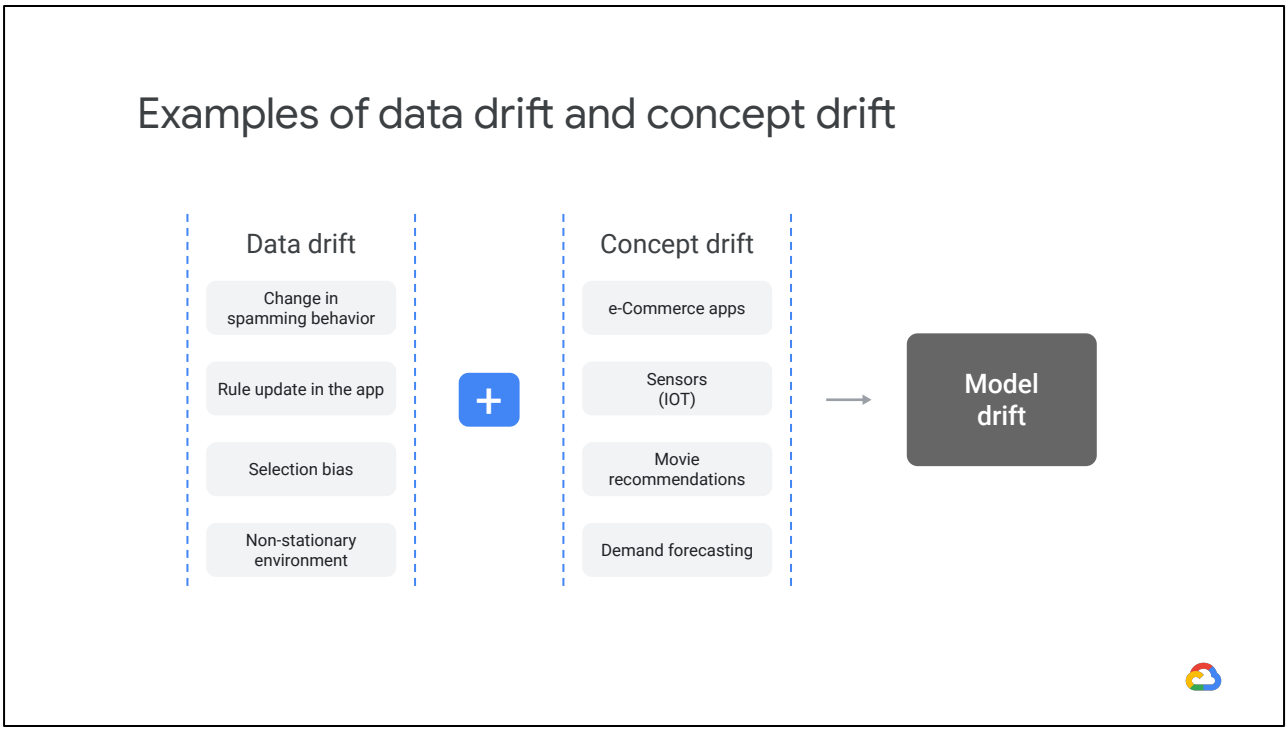
Actions to mitigate concept drift



Module 02
Designing adaptable ML systems



As previously mentioned, both data drift and concept drift lead to Model Drift.



Data drift - Examples of **data drift** can include concepts of:

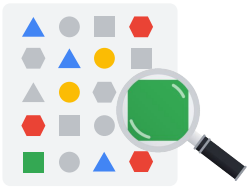
- The change in the spamming behavior to try to fool the model.
- A rule update in the app–in other words, a change in the limit of user messages per minute.
- Selection bias.
- Non-stationary environment--training data for a given season that has no power to generalize to another season.

Concept drift

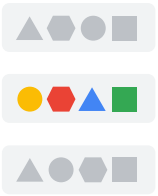
- eCommerce apps are good examples of potential **concept drift** due to their reliance on personalization, for example, the fact that people's preferences ultimately do change over time.
- Sensors may also be subject to concept drift due to the nature of the data they collect and how it may change over time.
- Movie recommendations - again - similar to eCommerce apps - rely on user preferences - and they may change.
- Demand forecasting heavily relies on time, and as we have seen, time is a major contributor to potential concept drift.

So, what if you diagnose data drift? If you diagnose data drift, enough of the data needs to be labeled to introduce new classes and the model retrained.

What if you diagnose concept drift? If you diagnose concept drift, the old data needs to be relabeled and the model retrained.



Design your systems to detect changes



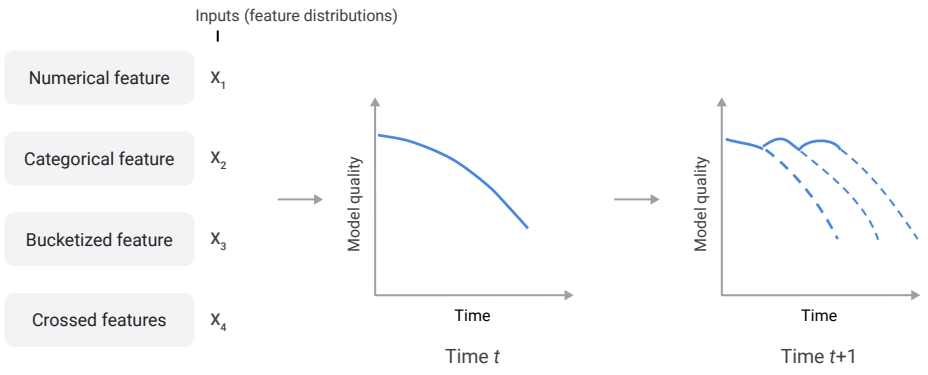
Use an ensemble approach to train

Also, for concept drift, you can design your systems to detect changes. Periodically updating your static model with more recent historical data, for example, is a common way to mitigate concept drift. You can either discard the static model completely or you can use the existing state as the starting point for a better model to update your model by using a sample of the most recent historical data.

You can also use an ensemble approach to train your new model in order to correct the predictions from prior models. The prior knowledge learnt from the old concept is used to improve the learning of the new concept. Ensembles which learnt the old concept with high diversity are trained by using low diversity on the new concept.

Concept drift

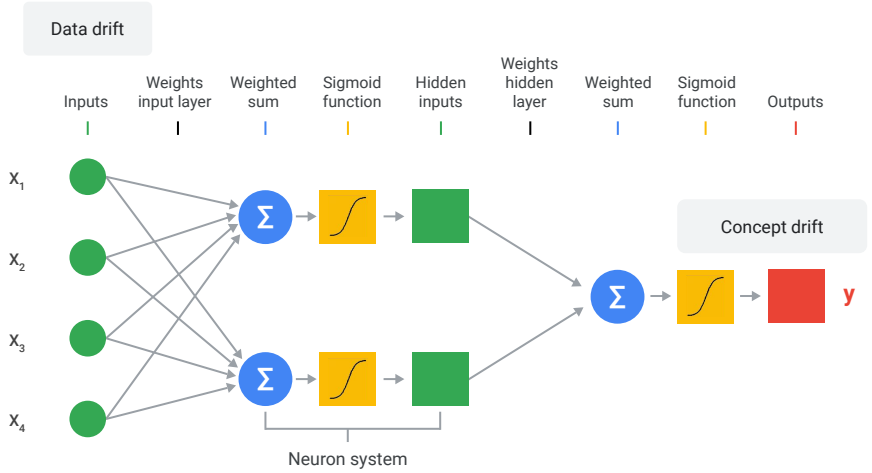
Change in relationships between the model inputs and the model output



Remember, concept drift is the change in relationships between the model inputs and the model output.

After your diagnosis and mitigation efforts, retraining or refreshing the model over time will help to maintain model quality.

Model drift = {Data drift, Concept drift}

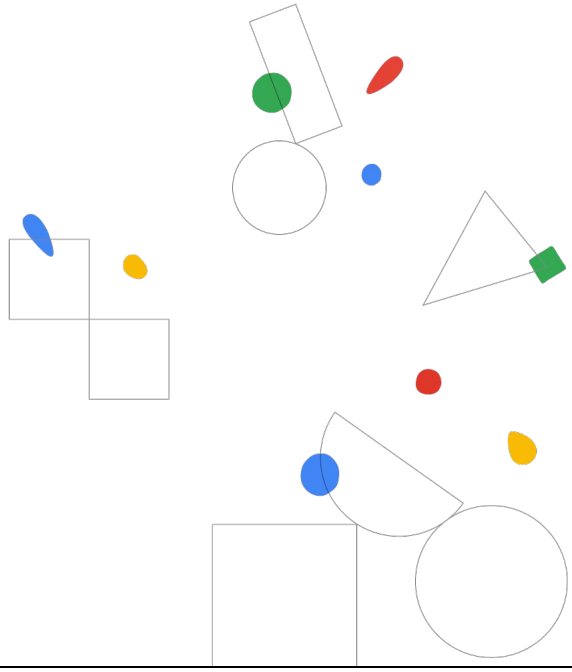


As the world changes, your data may change. The change can be gradual, sudden, and seasonal. These changes will impact model performance.

Thus, machine learning models can be expected to degrade or decay. Sometimes, the performance drop is due to low data quality, broken data pipelines, or technical bugs.

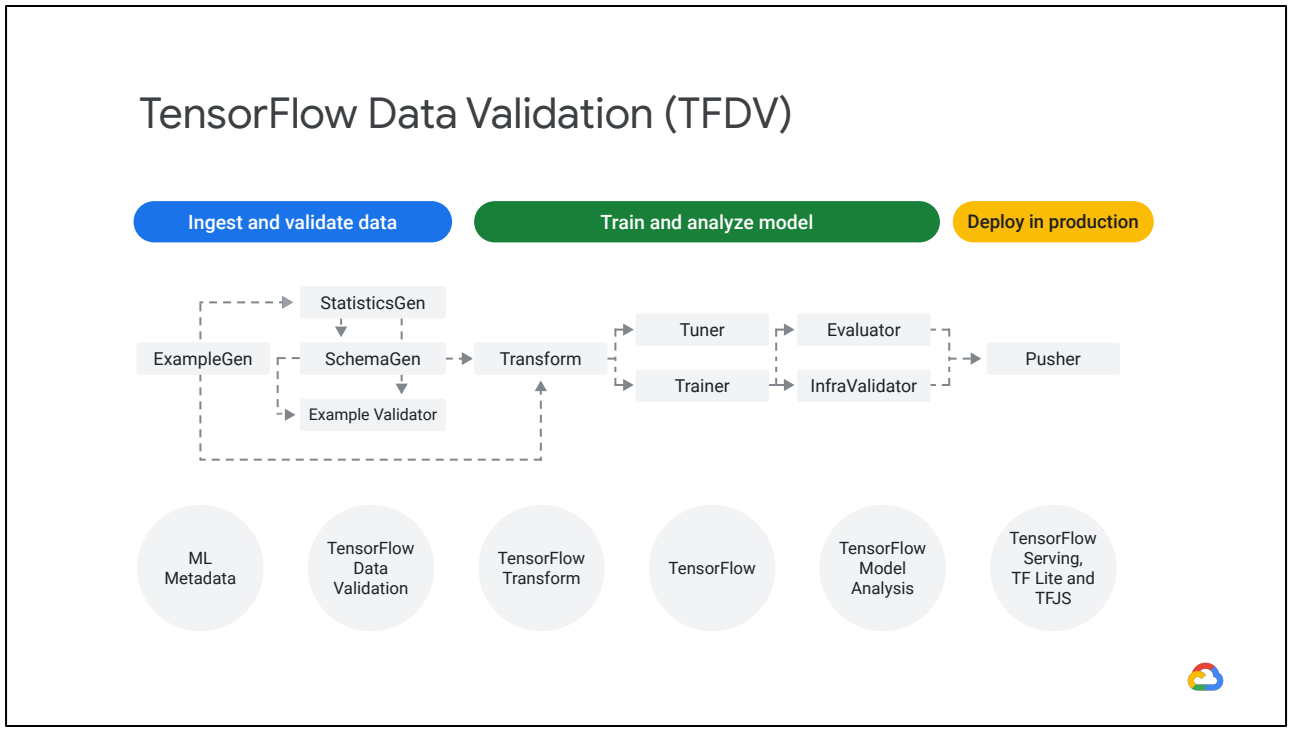
Google Cloud

TensorFlow Data Validation



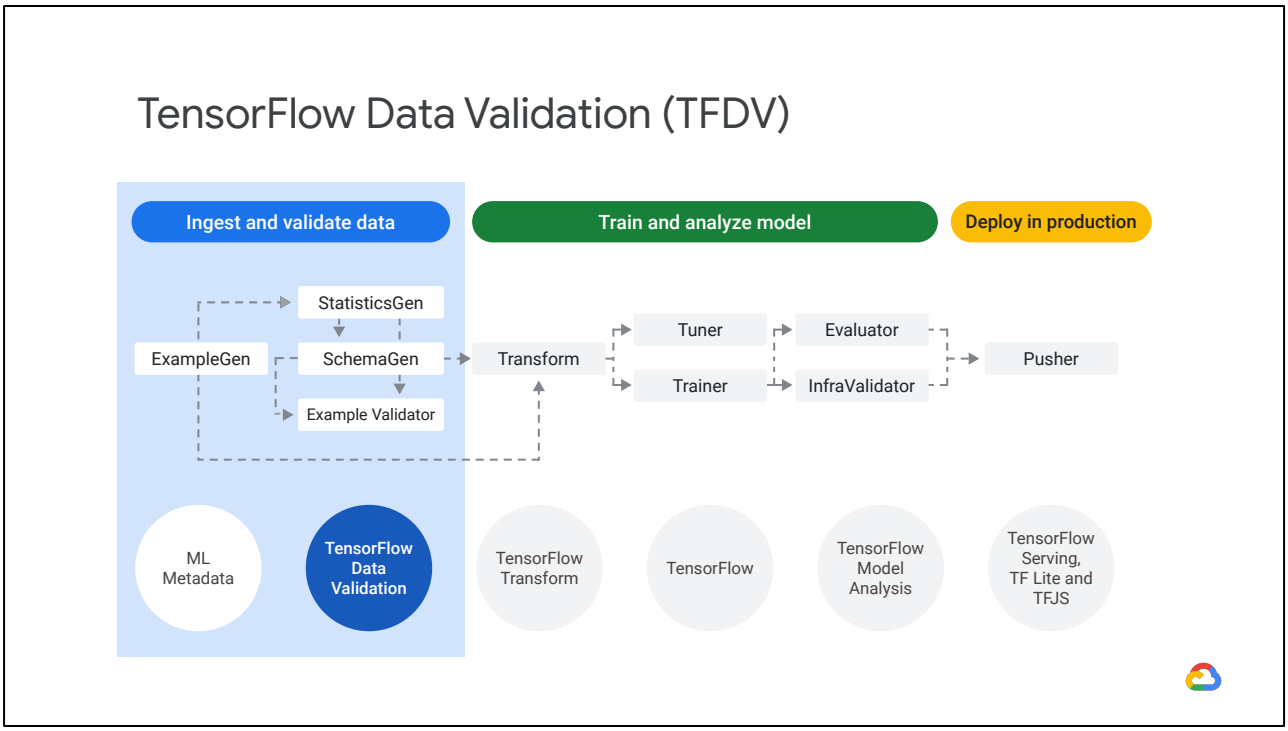
Module 02

Designing adaptable ML systems



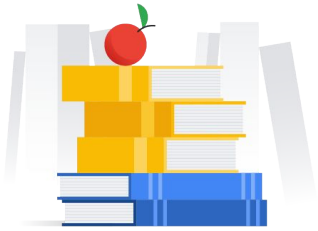
There are three phases in a pipeline:

- Data is ingested and validated
- A model is trained and analyzed
- The model is then deployed in production



In this video, we'll provide an overview of TensorFlow Data Validation, which is part of the ingest and validate data phase.

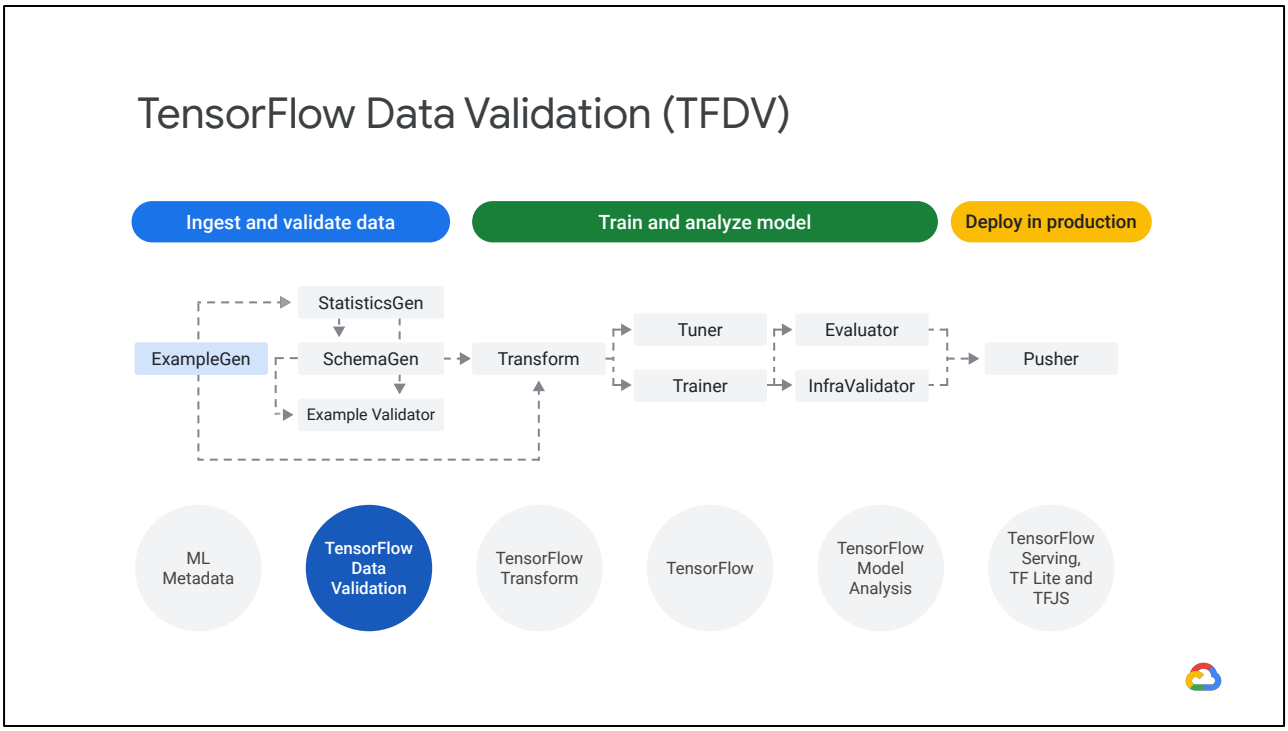
To learn more about the train and analyze the model phase, or how to deploy a model in production, please check out our ML Ops Course.



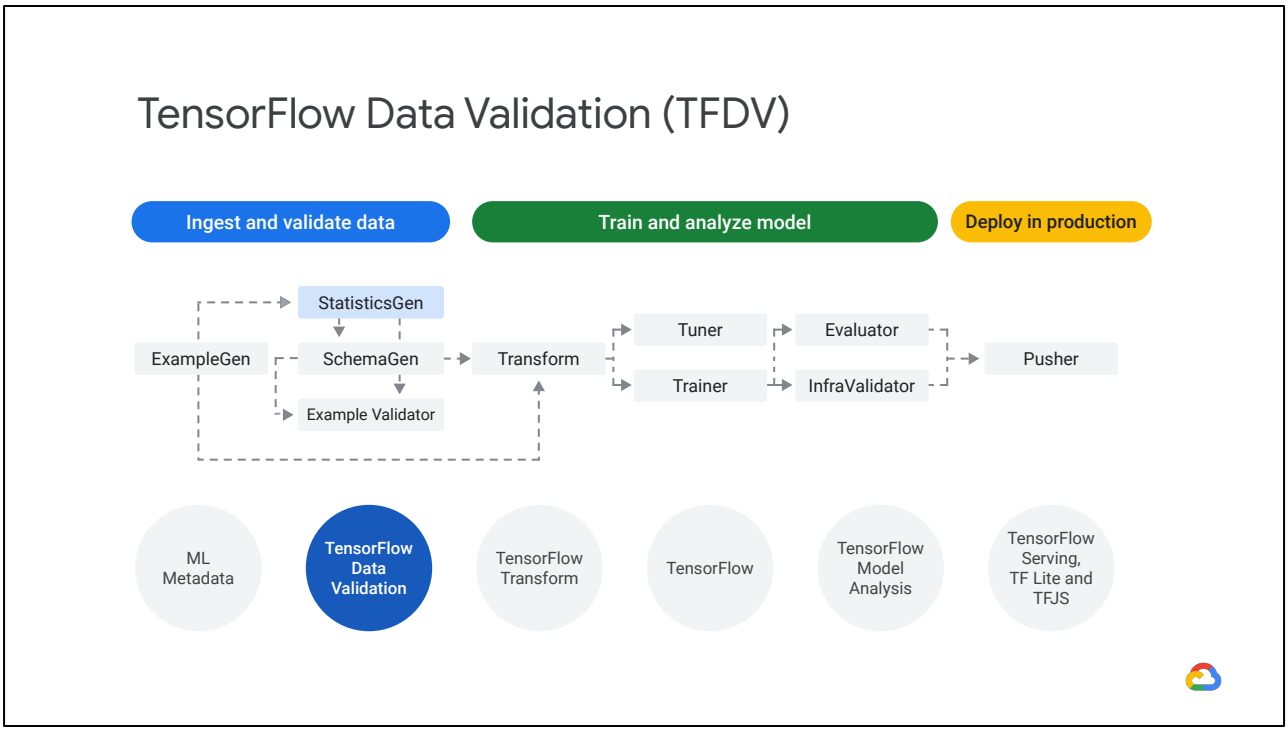
- Validation of continuously arriving data
- Training/serving skew detection

TensorFlow Data Validation is a library for analyzing and validating machine learning data.

Two common use-cases of TensorFlow Data Validation within a TensorFlow Extended pipelines are validation of continuously arriving data and training/serving skew detection.

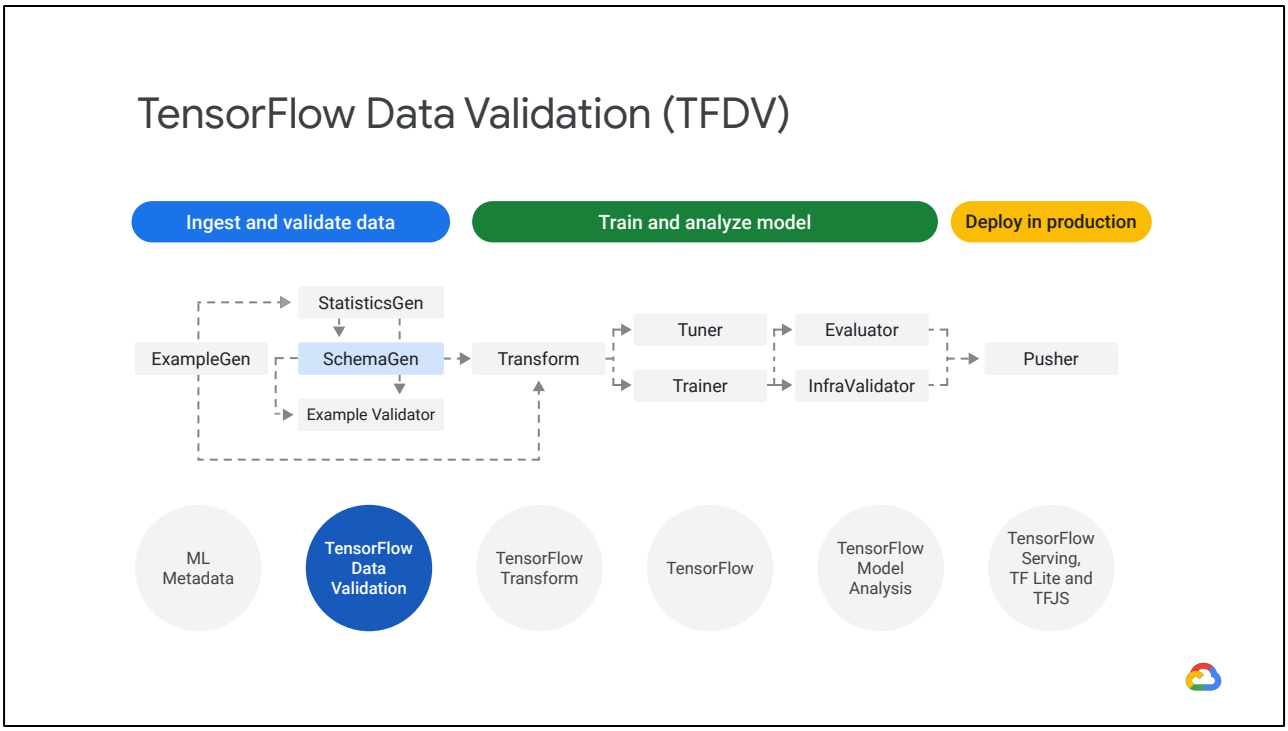


The pipeline begins with the ExampleGen component. This component takes raw data as input and generates TensorFlow examples, it can take many input formats (e.g. CSV, TF Record). It also does split the examples for you into Train/Eval. It then passes the result...

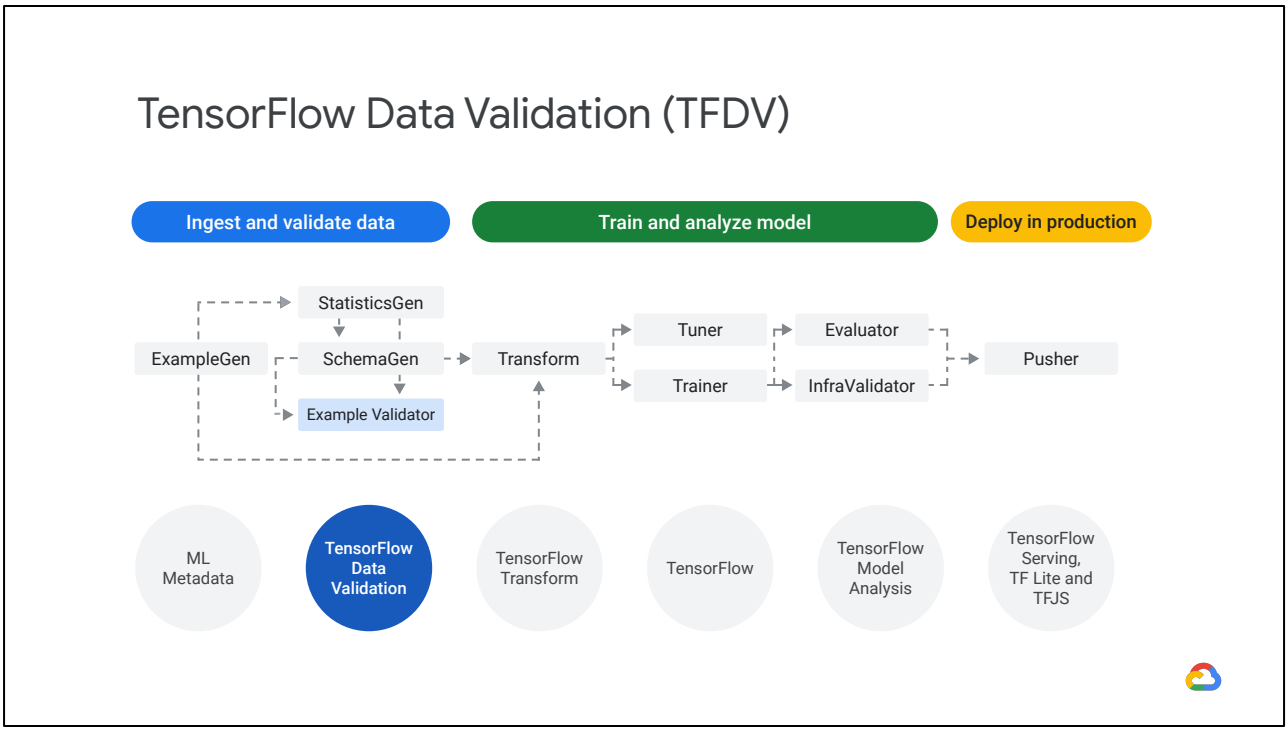


...to the StatisticsGen component. This brings us to the three main components of TensorFlow Data Validation.

The Statistics Generation component - which generates statistics for feature analysis,



the Schema Generation component - which gives you a description of your data,



and the Example Validator component - which allows you to check for anomalies.

We'll explore those three components in more depth in the next video, but first let's look at our use cases for TensorFlow Data Validation so that we can understand how these components work.

Reasons to analyze and transform your data

To find problems in your data.

Common problems include:

- Missing data
- Labels treated as features
- Features with values outside an expected range
- Data anomalies

To engineer more effective feature sets.

Identify:

- Informative features
- Redundant features
- Features that vary so widely in scale that they may slow learning
- Features with little or no unique predictive information

- There are many reasons and use cases where you may need to analyze and transform your data.

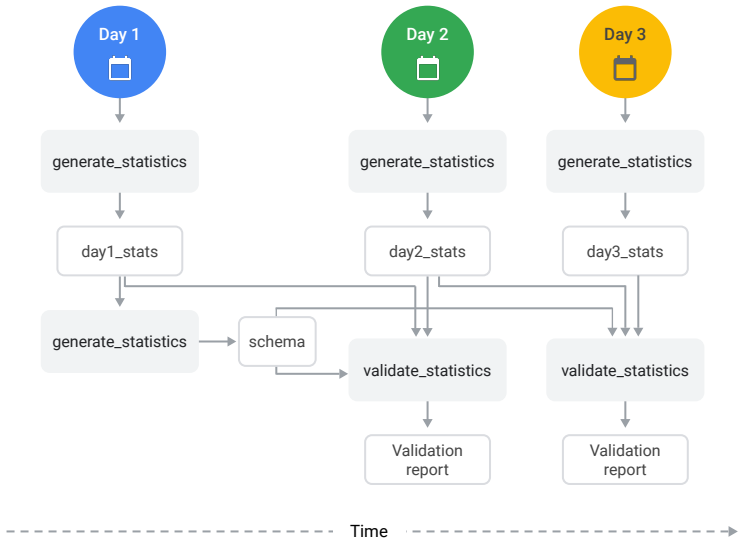
  For example, when you're missing data, such as features with empty values, or when you have labels treated as features, so that your model gets to peek at the right answer during training. You may also have features with values outside the range you expect or other data anomalies.

- To engineer more effective feature sets, you should identify:

  Especially informative features, redundant features, features that vary so widely in scale that they may slow learning, and features with little or no unique predictive information.

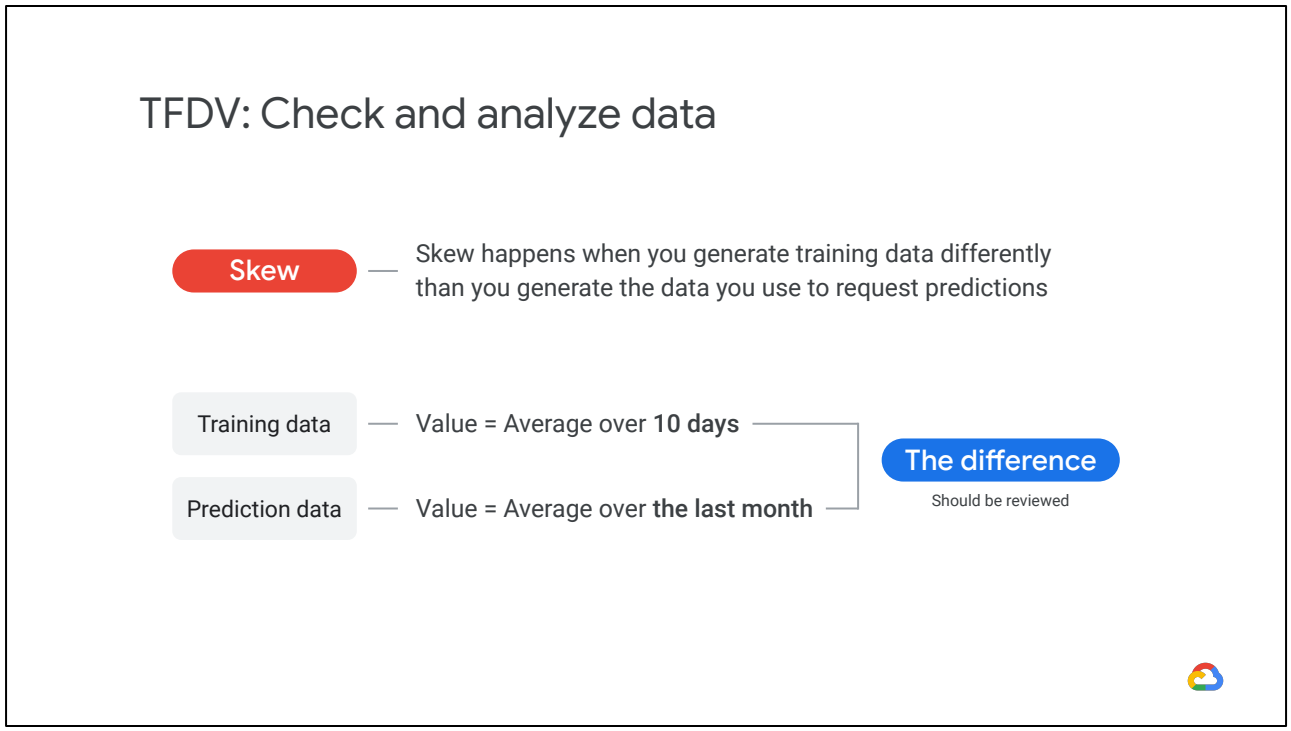
TFDV

Check and analyze data



One use case for TensorFlow Data Validation is to validate continuously arriving data.

- Let's say on day one you generate statistics based on data from day one.
- Then, you generate statistics based on day two data. From there, you can validate Day 2 statistics against day one statistics and generate a validation report.
- You can do the same for day three, validating day three statistics against statistics from both day two and day one.



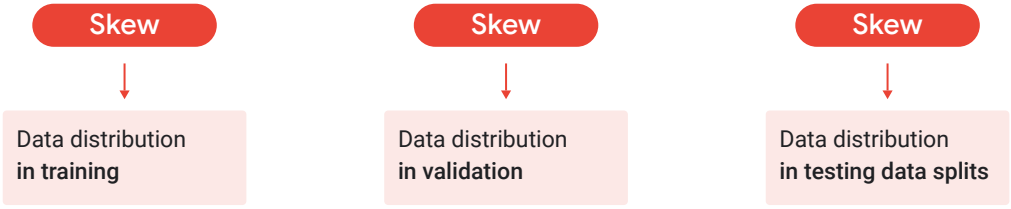
TensorFlow Data Validation can also be used to detect distribution skew between training and serving data. Training-serving skew occurs when training data is generated differently from how the data used to request predictions is generated.

But what causes distribution skew? Possible causes might come from a change in how data is handled in training vs in production, or even a faulty sampling mechanism that only chooses a subsample of the serving data to train on.

For example, if you use an average value, and for training purposes you average over 10 days, but when you request prediction, you average over the last month.

In general, any difference between how you generate your training data and your serving data (the data you use to generate predictions) should be reviewed to prevent training-serving skew.

TFDV: Check and analyze data

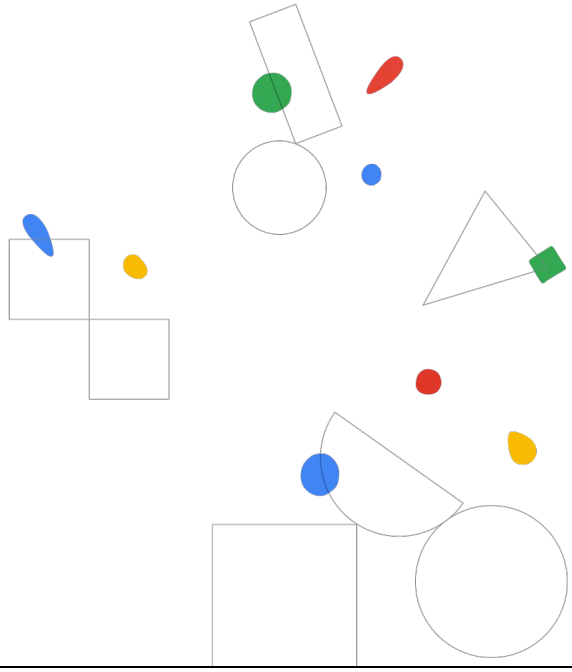


Training-serving skew can also occur based on your data distribution in your training, validation, and testing data splits.

To summarize, distribution skew occurs when the distribution of feature values for training data is significantly different from serving data and one of the key causes for distribution skew is how data is handled or changed in training vs in production.

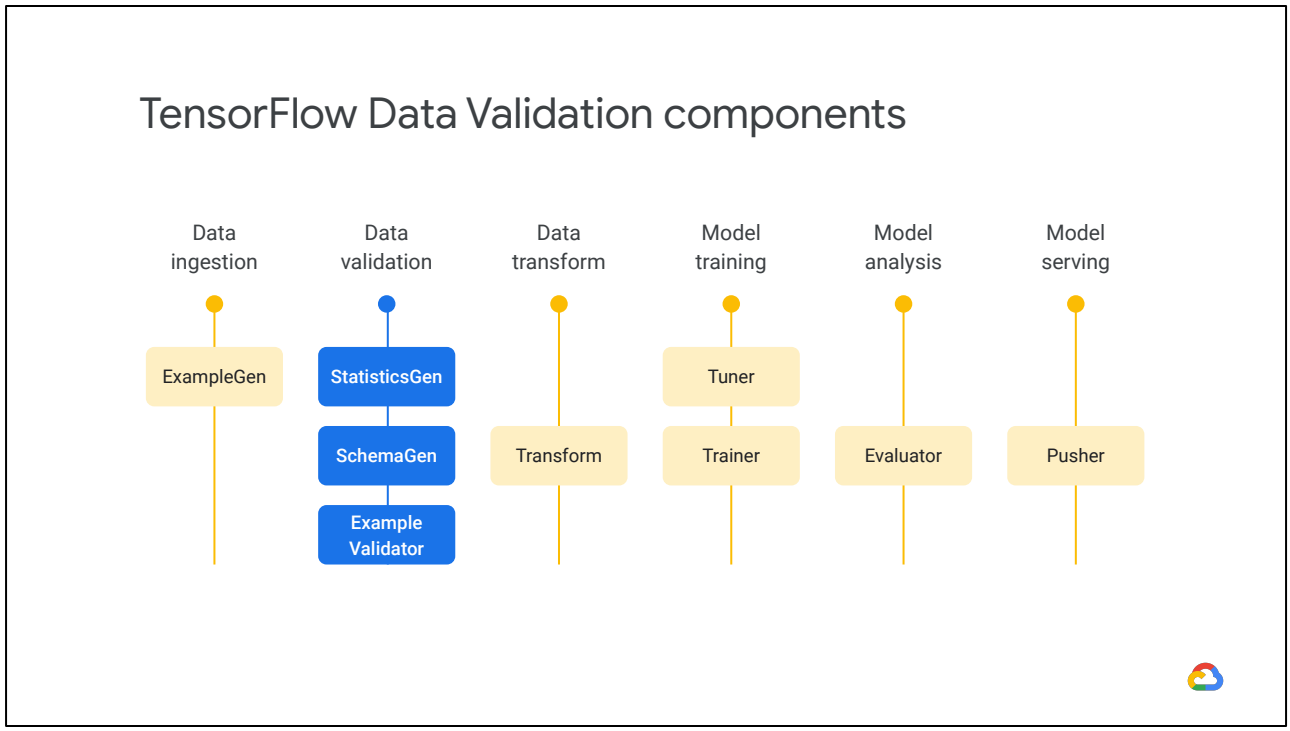
Google Cloud

Components of TensorFlow Data Validation



Module 02

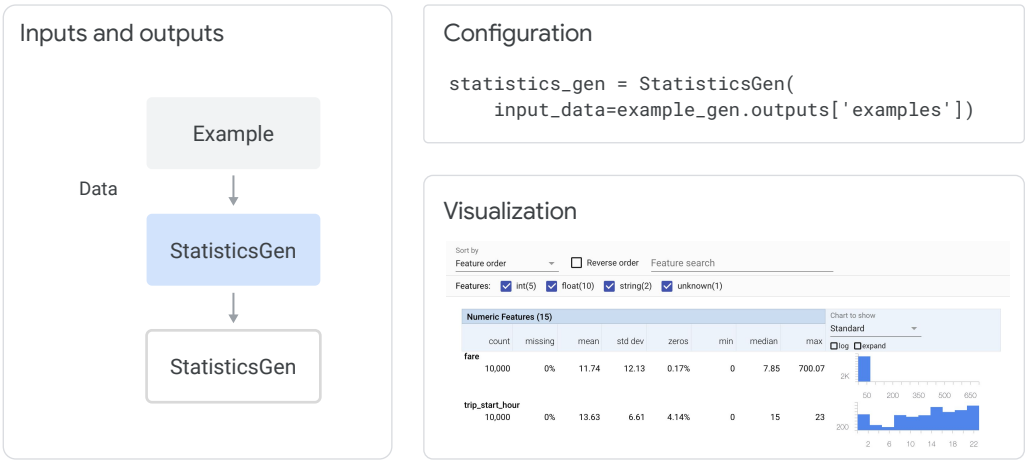
Designing adaptable ML systems



TensorFlow Data Validation is a library for analyzing and validating machine learning data, for which there are three components:

- The Statistics Generation component
- The Schema Generation component
- The Example Validator component

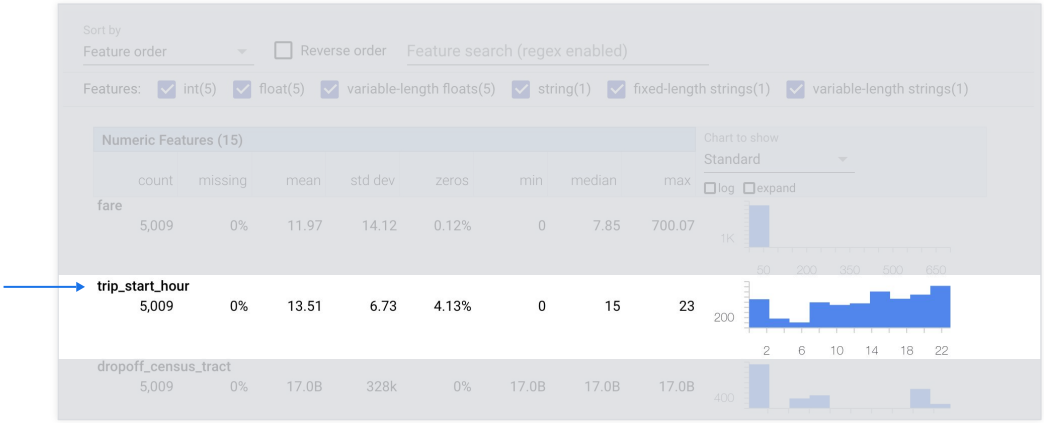
StatisticsGen



The StatisticsGen component generates features statistics and random samples over training data, which can be used for visualization and validation. It requires minimal configuration.

For example, StatisticsGen takes as input a dataset ingested using ExampleGen. After StatisticsGen finishes running, you can visualize the outputted statistics.

Analyzing data with TensorFlow Data Validation



In an example using taxi data, the StatisticsGen component has generated data statistics for the numeric features. For the trip_start_hour feature, it appears there is not that much data in the early morning hours.

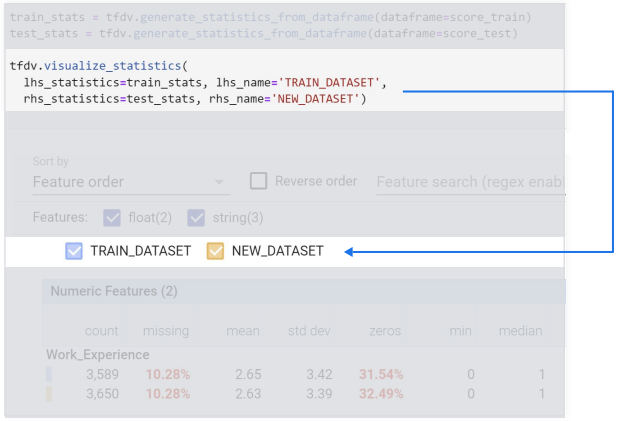
It appears that the trip_start_hour column, where the time window is between 2:00am to 6:00am, has data missing.

This helps determine the area we need to focus on to fix any data-related problems. We'll need to get more data, otherwise, the prediction for 4:00am data will be overgeneralized.

Generate and visualize data

How big is the difference between TRAIN_DATASET and NEW_DATASET?

Does this difference matter?



Let's look at another example, this time using a consumer spending score dataset.

Here we are generating statistics for both a training dataset, that may have arrived on day one, and a new dataset, that may have arrived on day two. These statistics are being generated from a Pandas DataFrame. You can also generate statistics from a CSV file or a TF.Record formatted file.

By comparing both datasets, you can analyze how big of a difference there is between the two, and then determine if that difference matters.

Numeric and categorical features

Numeric Features (2)

	count	missing	mean	std dev
Work_Experience				
	3,589	10.28%	2.65	3.42
	3,650	10.28%	2.63	3.39
Family_Size				
	3,831	4.23%	2.84	1.51
	3,902	4.08%	2.86	1.55

Categorical Features (3)

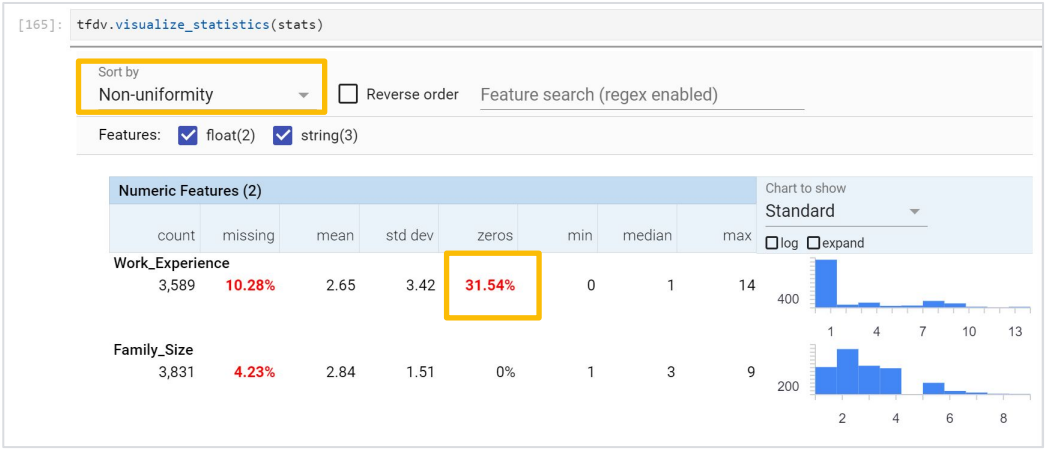
	count	missing	unique	top	freq top	avg str len
Graduated						
	3,964	0.9%	2	Yes	2,433	2.61
	4,026	1.03%	2	Yes	2,535	2.63
Profession						
	3,944	1.4%	9	Artist	1,218	8.14
	4,000	1.67%	9	Artist	1,298	8.09
Spending_Score						
	4,000	0%	3	Low	2,448	4.11
	4,068	0%	3	Low	2,430	4.14

StatisticsGen generates both numeric and categorical features.

In this example, our dataset has two numerical features, Work_Experience and Family_Size.

Our dataset also has three categorical features: Graduated, Profession, and Spending_Score. Notice that for our categorical features, in addition to seeing the number of missing values, we also see the number of unique values.

Identify unbalanced data distributions



TensorFlow Data Validation can also help you identify unbalanced data distributions. For example, if you have a dataset you are using for a classification problem and you see that one feature has a lower percentage of values than the other, you can use TensorFlow Data Validation to detect this “unbalance”.

The most unbalanced features will be listed at the top of each feature-type list. For example, the following screenshot shows Work_Experience with zero has a data value - which means that 31.54% of the values in this feature are zeroes.

Data validation checks

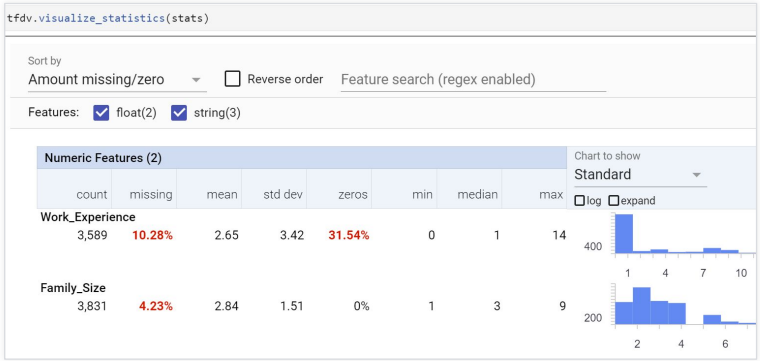
Feature min, max, mean, mode, median

Feature correlations

Class imbalance

Check to see missing values

Histograms of features (for both numerical and categorical)

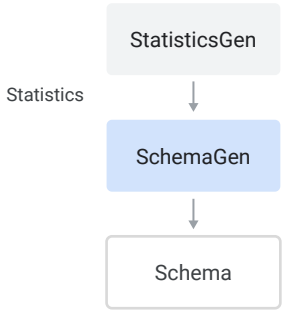


There are a number of StatisticsGen data validation checks that you should be aware of. These include:

- Feature min, max, mean, mode, and median
- Feature correlations
- Class imbalance
- Check to see missing values
- Histograms of features (for both numerical and categorical)

SchemaGen

Inputs and outputs



Configuration

```
infer_schema =
    SchemaGen(statistics=statistics_gen.outputs['statistics'],
              infer_feature_shape=False)
```

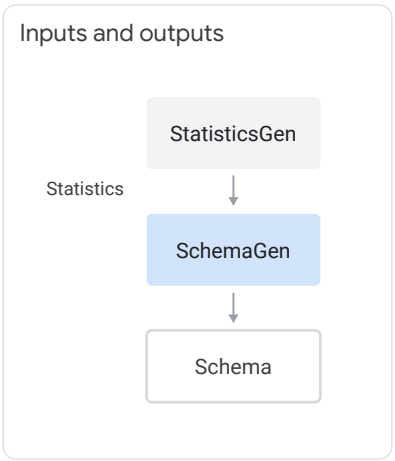
Visualization

Feature name	Type	Presence	Valency	Domain
'fare'	FLOAT	required	single	-
'trip_start_hour'	INT	required	single	-
'pickup_census_tract'	BYTES	optional		-
'dropoff_census_tract'	FLOAT	optional	single	-
'company'	STRING	optional	single	'company'

The SchemaGen TFX pipeline component can specify data types for feature values, whether a feature has to be present in all examples, allowed value ranges, and other properties. A SchemaGen pipeline component will automatically generate a schema by inferring types, categories, and ranges from the training data.

In essence, SchemaGen is looking at the data type of the input, is it an int, float, categorical, etc. If it is categorical then what are the valid values? It also comes with a visualization tool to review the inferred schema and fix any issues.

SchemaGen



Configuration

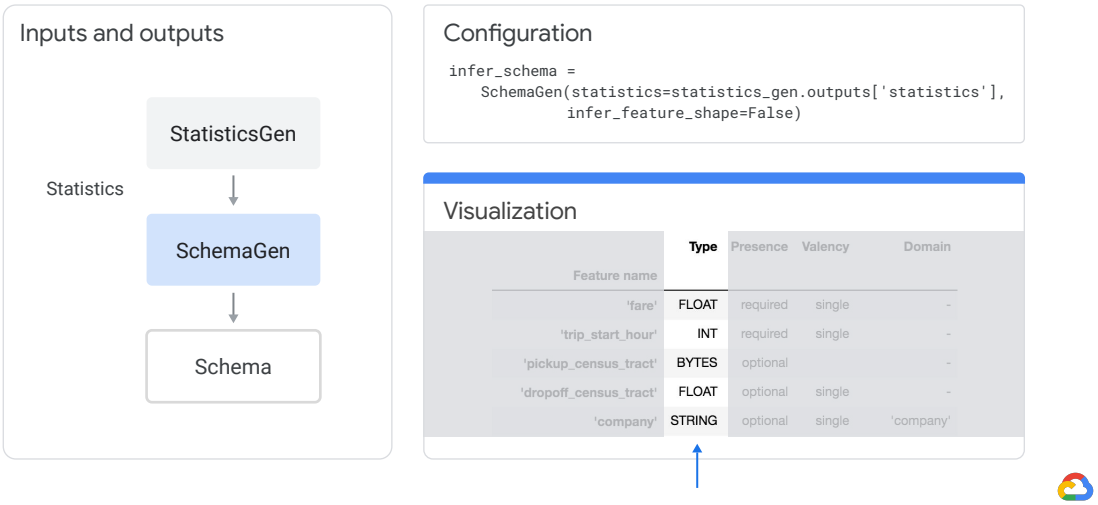
```
infer_schema =
    SchemaGen(statistics=statistics_gen.outputs['statistics'],
              infer_feature_shape=False)
```

Visualization

Feature name	Type	Presence	Valency	Domain
'fare'	FLOAT	required	single	-
'trip_start_hour'	INT	required	single	-
'pickup_census_tract'	BYTES	optional		-
'dropoff_census_tract'	FLOAT	optional	single	-
'company'	STRING	optional	single	'company'

In this example visualization:

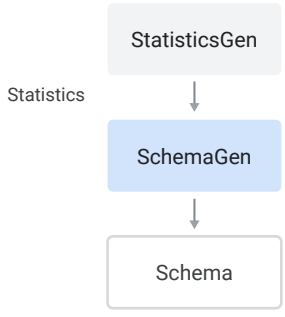
SchemaGen



Type indicates the feature datatype.

SchemaGen

Inputs and outputs



Configuration

```
infer_schema =
    SchemaGen(statistics=statistics_gen.outputs['statistics'],
              infer_feature_shape=False)
```

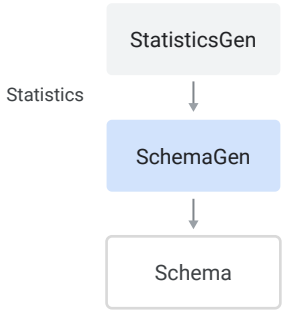
Visualization

Feature name	Type	Presence	Valency	Domain
'fare'	FLOAT	required	single	-
'trip_start_hour'	INT	required	single	-
'pickup_census_tract'	BYTES	optional		-
'dropoff_census_tract'	FLOAT	optional	single	-
'company'	STRING	optional	single	'company'

Presence indicates whether the feature must be present in 100% of examples (required) or not (optional).

SchemaGen

Inputs and outputs



Configuration

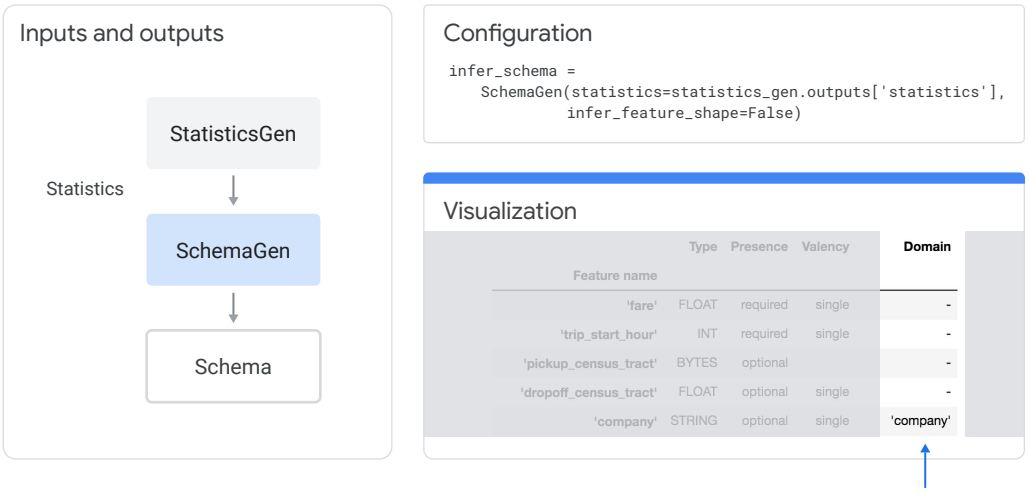
```
infer_schema =
    SchemaGen(statistics=statistics_gen.outputs['statistics'],
              infer_feature_shape=False)
```

Visualization

Feature name	Type	Presence	Valency	Domain
'fare'	FLOAT	required	single	-
'trip_start_hour'	INT	required	single	-
'pickup_census_tract'	BYTES	optional		-
'dropoff_census_tract'	FLOAT	optional	single	-
'company'	STRING	optional	single	'company'

Valency indicates the number of values required per training example.

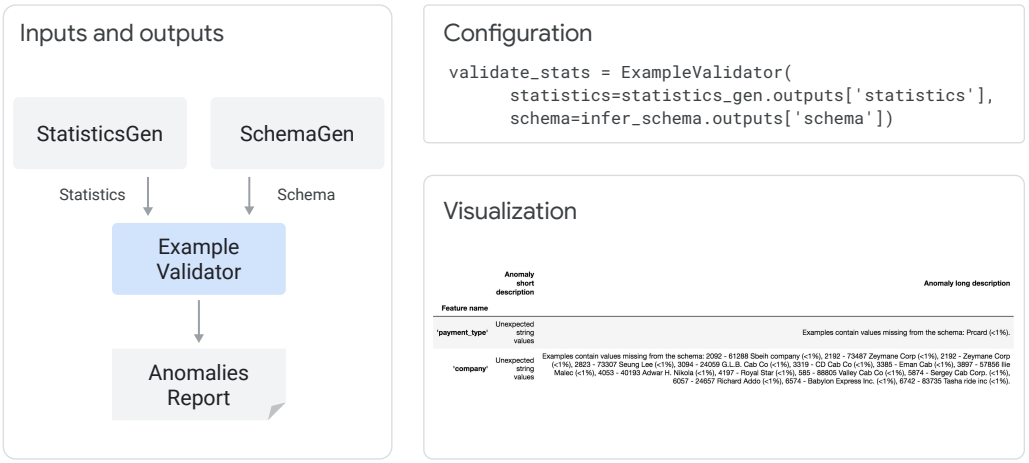
SchemaGen



Domain and Values indicates the feature domain and its values.

In the case of categorical features, single indicates that each training example must have exactly one category for the feature.

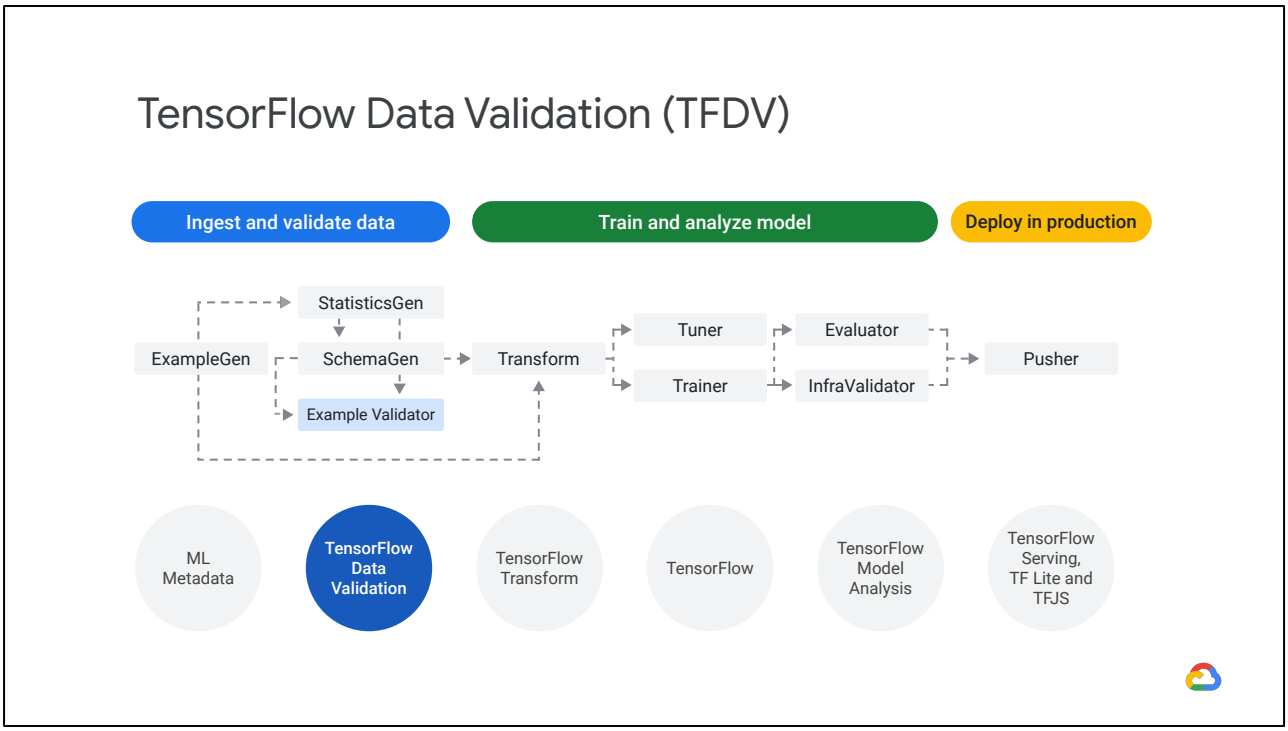
ExampleValidator



The ExampleValidator pipeline component identifies anomalies in training and serving data. It can detect different classes of anomalies in the data and emit validation results.

The ExampleValidator pipeline component identifies any anomalies in the example data by comparing data statistics computed by the StatisticsGen pipeline component against a schema.

It takes the inputs and looks for problems in the data, like missing values, and reports any anomalies.



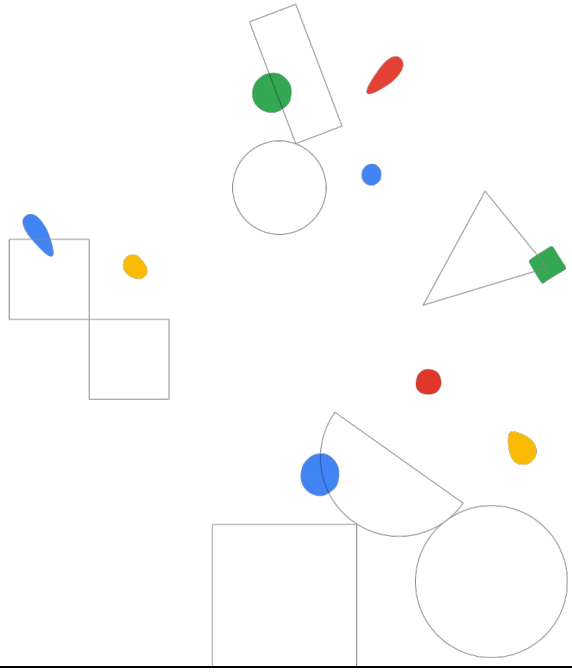
As we've explored, TensorFlow Data Validation is a component of TensorFlow Extended and it helps you to analyze and validate your data.

Data validation checks include identifying feature correlations, checking for missing values, and identifying class imbalances.

Google Cloud

Lab

Introduction to TensorFlow Data Validation

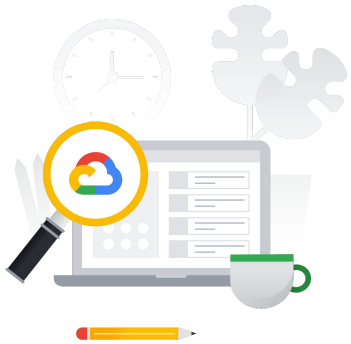


Module 02

Designing adaptable ML systems

This lab provides a hands-on introduction to TensorFlow Data Validation.

Lab objectives



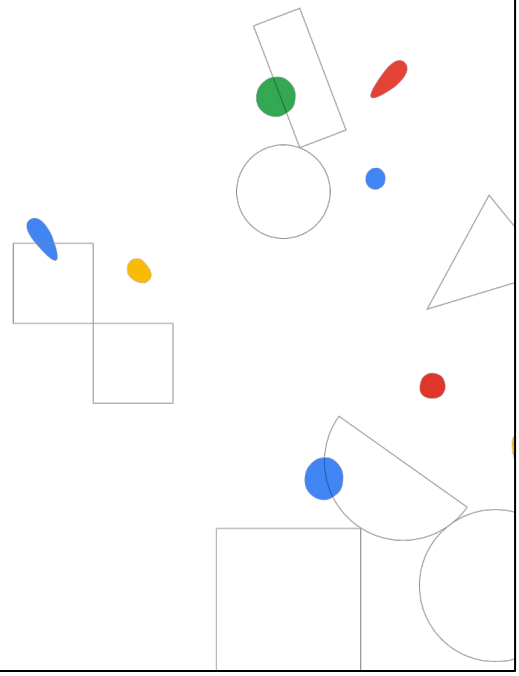
- Review TFDV methods.
- Generate statistics.
- Visualize statistics.
- Infer a schema.
- Update a schema.

- You'll begin by reviewing the different TensorFlow Data Validation methods,
- then continue on to generate statistics,
- visualize statistics,
- infer a schema, and, finally,
- update a schema.

Google Cloud

Lab

Advanced Visualizations with TensorFlow Data Validation

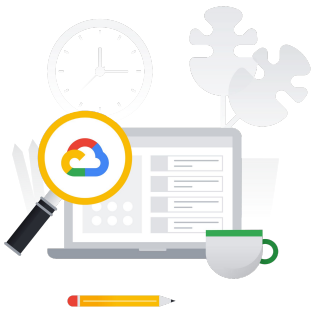


Module 02

Designing adaptable ML systems

This lab demonstrates how TensorFlow Data Validation can be used to investigate and visualize a dataset.

Lab objectives



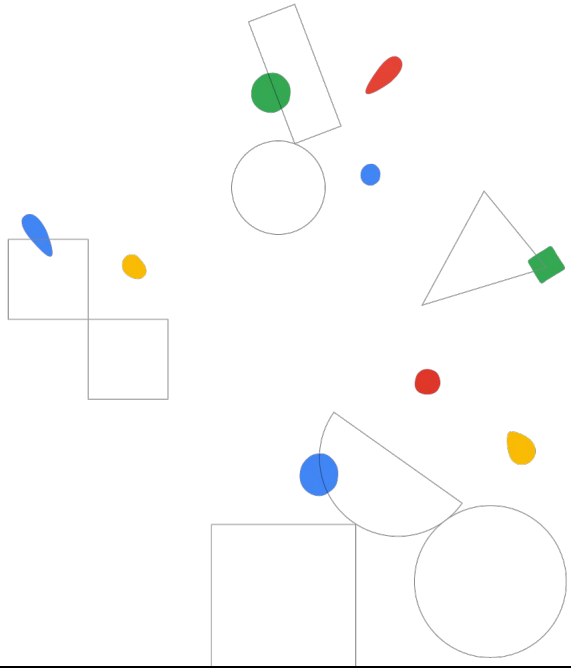
- Install TensorFlow Data Validation.
- Compute and visualize statistics.
- Infer a schema.
- Check evaluation data for errors.
- Check for and fix evaluation anomalies.
- Check for drift and skew.
- Freeze a schema.

You'll begin with:

- steps to install TensorFlow Data Validation,
- then continue on to compute and visualize statistics,
- infer a schema,
- check evaluation data for errors,
- check for and fix evaluation anomalies,
- check for drift and skew,
- and finally, freeze a schema.

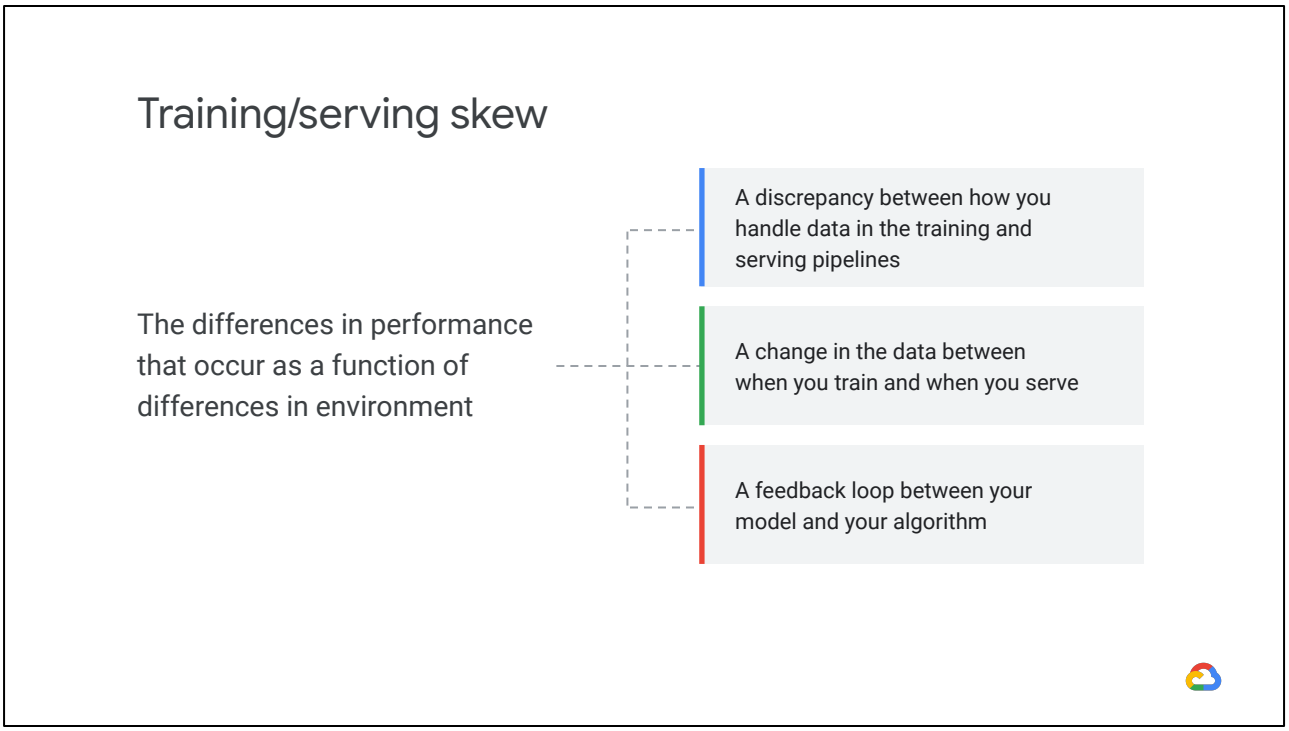
Google Cloud

Mitigating Training-Serving Skew Through Design



Module 02

Designing adaptable ML systems



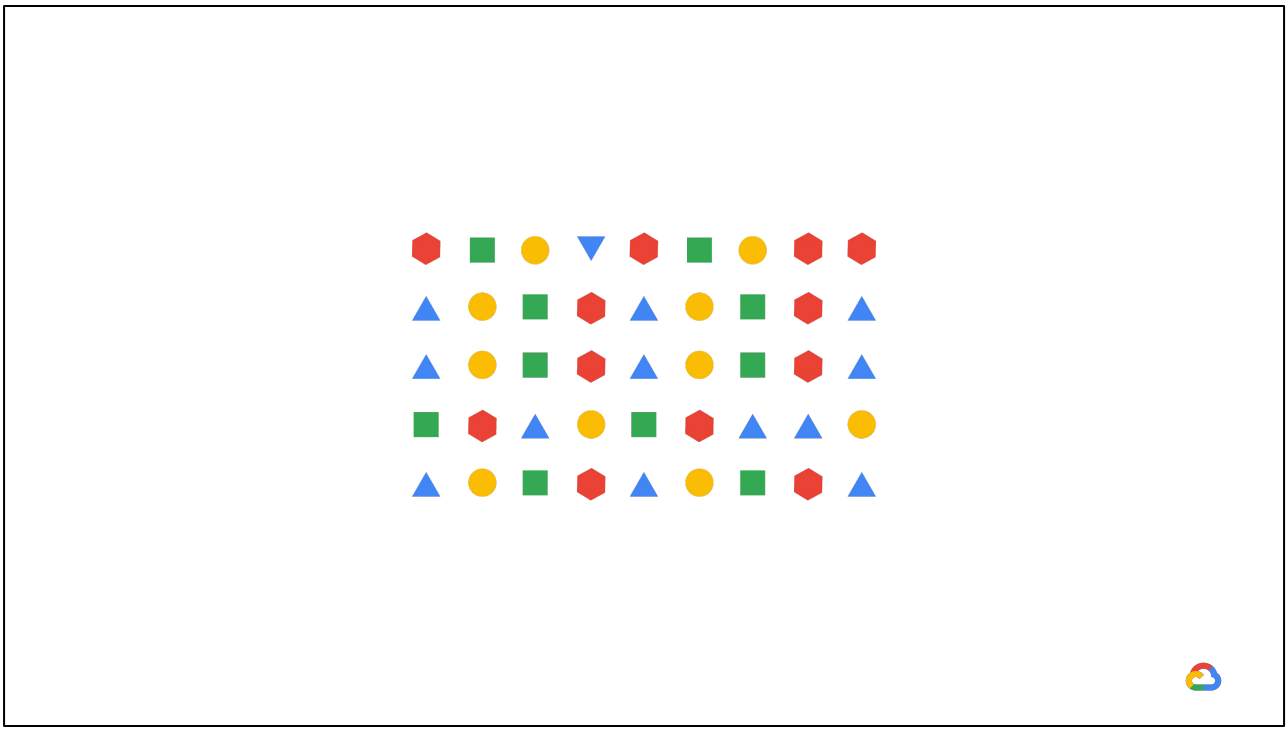
We've talked about training/serving skew a number of times in previous videos, but always at a high level.

Let's look at it now in a little more detail.

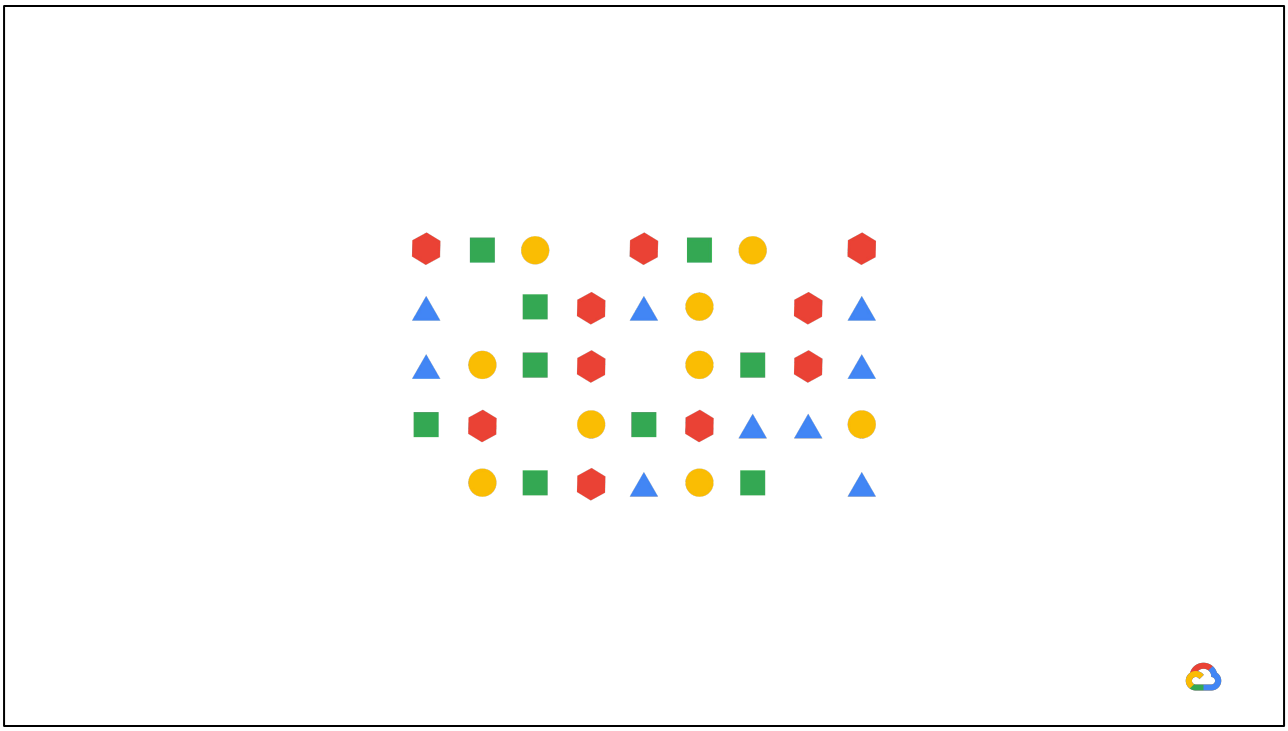
Training/Serving skew refers to differences in performance that occur as a function of differences in environment.

Specifically, training/serving skew refers to differences caused by one of three things:

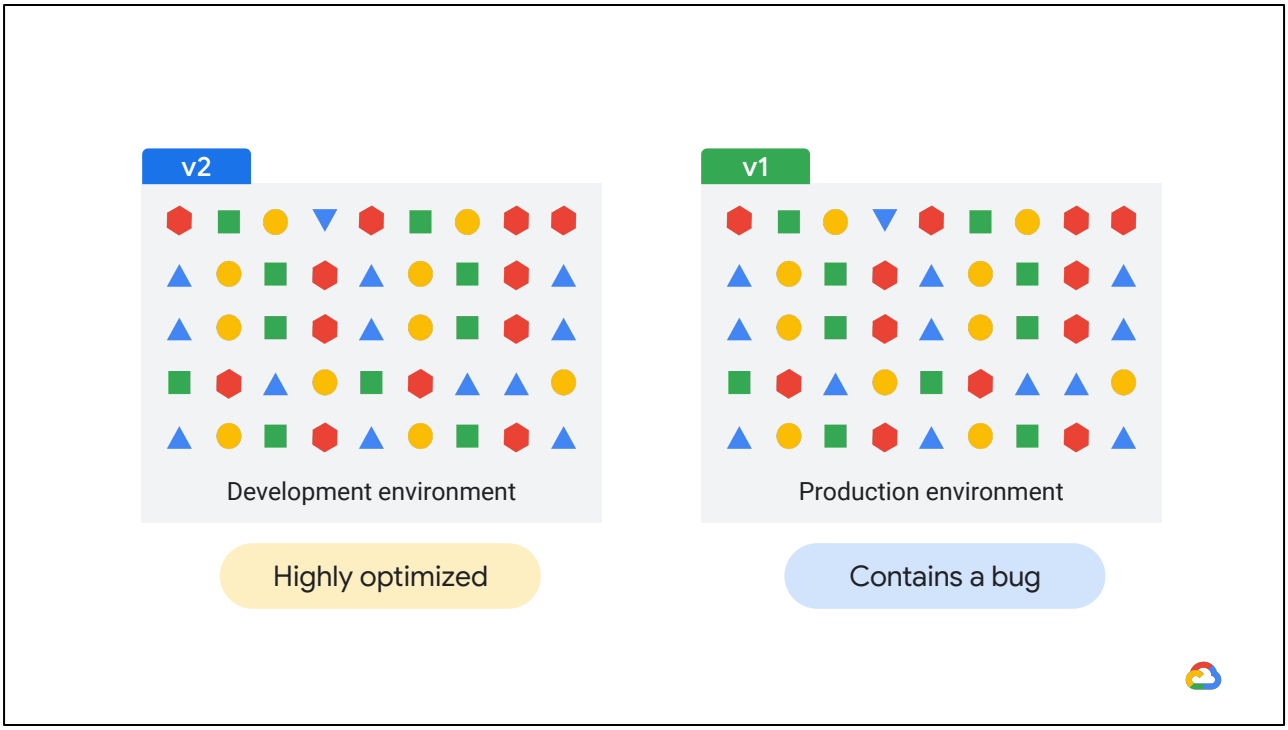
- A discrepancy between how you handle data in the training and serving pipelines.
- A change in the data between when you train and when you serve, or
- A feedback loop between your model and your algorithm.



Up until now, we've focused on the data aspect of training-serving skew,



but it’s also possible to have inconsistencies that arise after the data have been introduced.

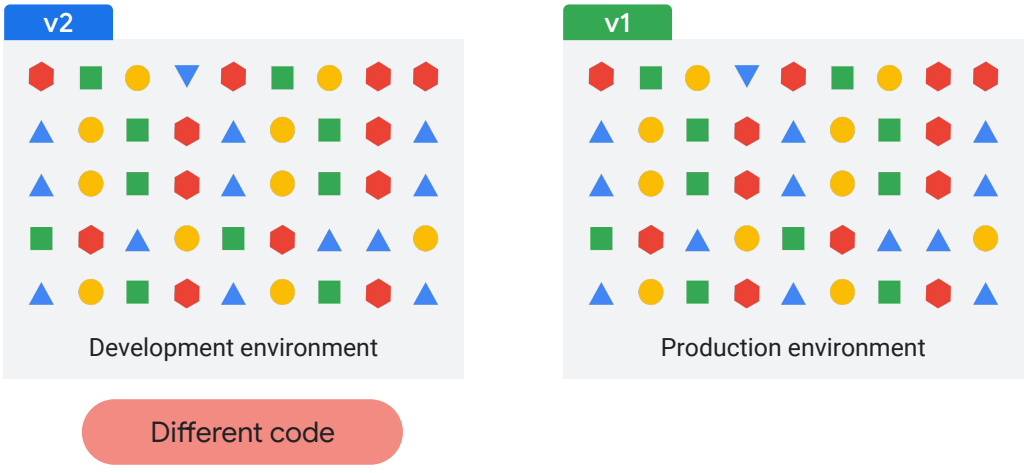


Say, for example, that in your development environment, you have version 2 of a library, but in production you have version 1.

The libraries may be functionally equivalent but version 2 is highly optimized and version 1 isn't.

Consequently, predictions might be significantly slower or consume more memory in production than they did in development.

Alternately, it's possible that version 1 and version 2 are functionally different, perhaps because of a bug.

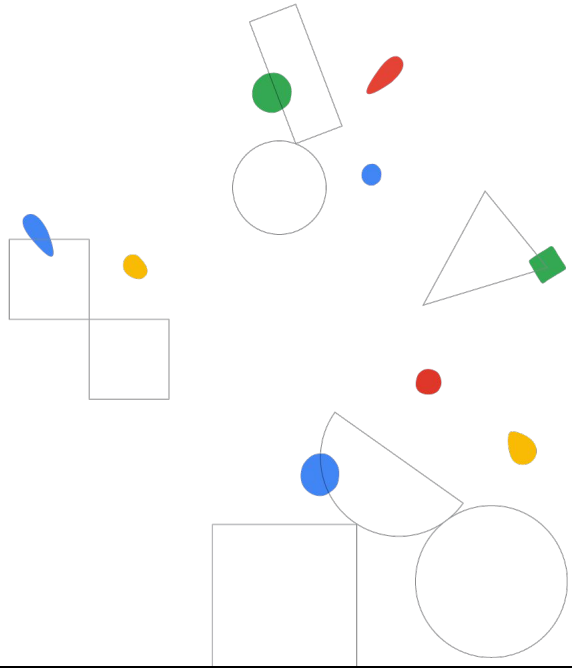


Finally, it's also possible that different code is used in production vs. development, perhaps because of recognition of one of the other issues, but though the intent was to create equivalent code, the results were imperfect.

Google Cloud

Lab

Serve ML Predictions in Batch and Real Time



Module 02

Designing adaptable ML systems

This lab provides hands-on practice serving machine learning predictions in batch and real time.

Lab objectives

- Create a prediction service to call a trained model that has been deployed in Google Cloud.
- Run a Dataflow job where the prediction service reads in batches from a CSV file.
- Run a streaming Dataflow pipeline to read requests in real time from Pub/Sub.
- Write predictions to a BigQuery table.

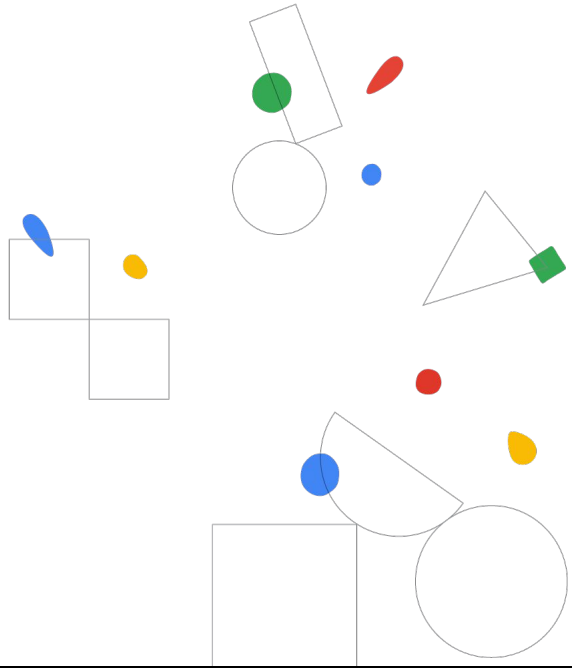
You'll begin by creating a prediction service to call a trained model that has been deployed in Google Cloud.

From there, you'll practice running a Dataflow job where the prediction service reads in batches from a CSV file.

And, finally, you'll learn how to run a streaming Dataflow pipeline to read requests in real time from Cloud Pub/Sub, and also write predictions to a BigQuery table.

Google Cloud

Diagnosing a production model

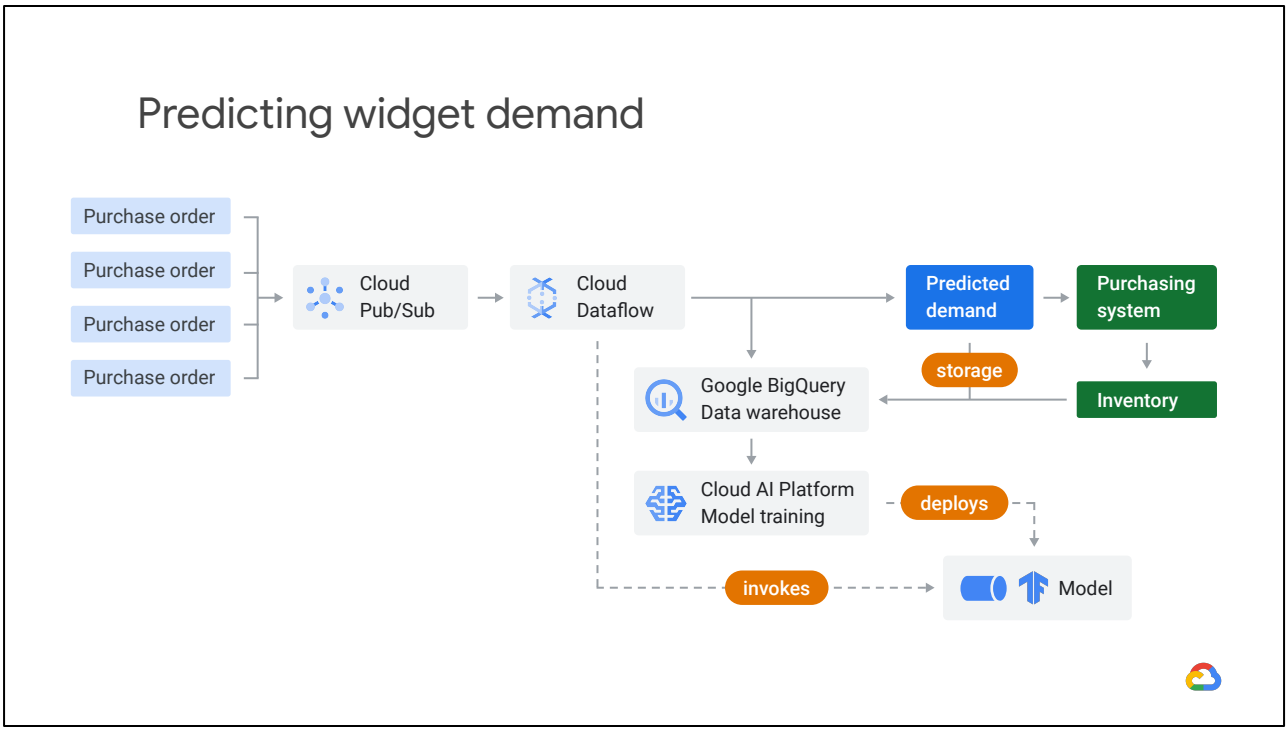


Module 02

Designing adaptable ML systems

Diagnosing a production model

In this section, we'll put our learnings into practice by diagnosing a production model.



Let's say you've architected an ML system to predict the demand for widgets.

Your streaming purchase orders arrive in Pub/Sub and are fulfilled asynchronously, but let's ignore fulfillment and focus on demand prediction.

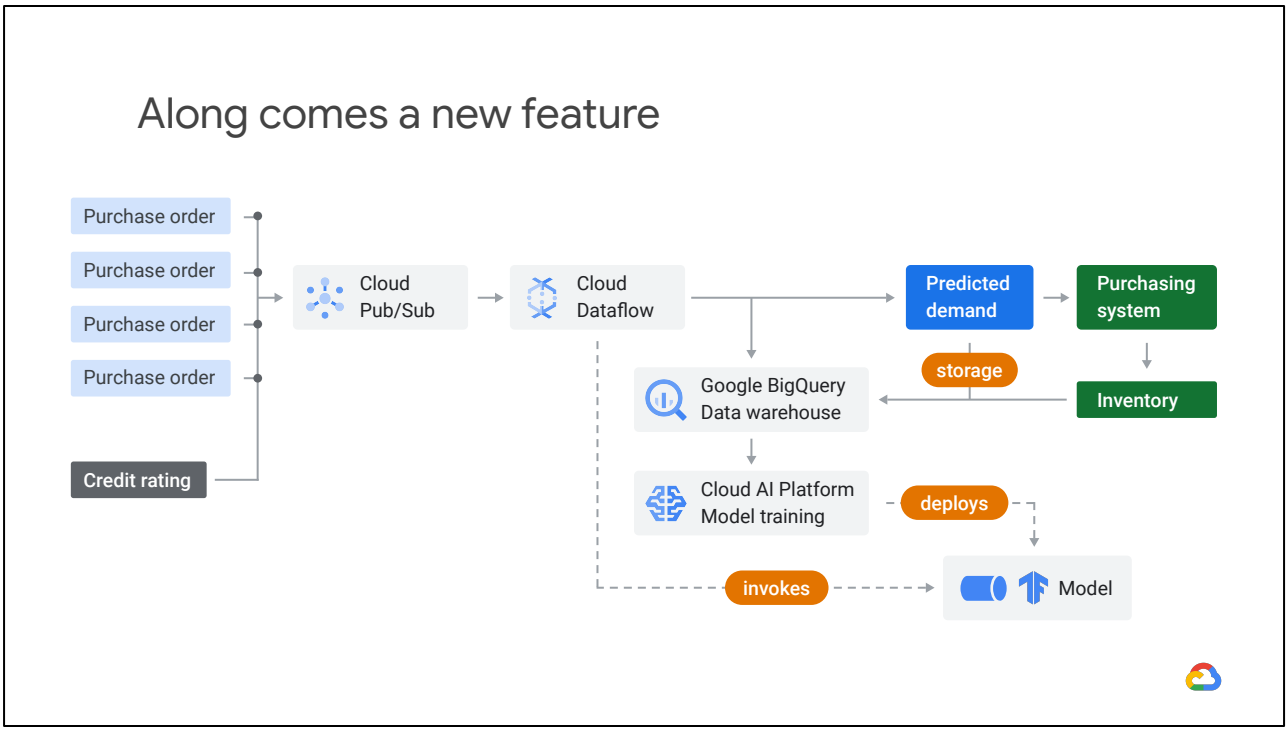
Dataflow processes this stream and using windowing functions, aggregates purchase orders over time. It then uses the output of these windowing functions and passes that to an ML model that lives in Cloud AI Platform, where the data are joined against historical purchase data that lives in a data warehouse like BigQuery.

The model returns a predicted demand for a particular time. The predictions returned by the model are both written to storage and sent to the purchasing system.

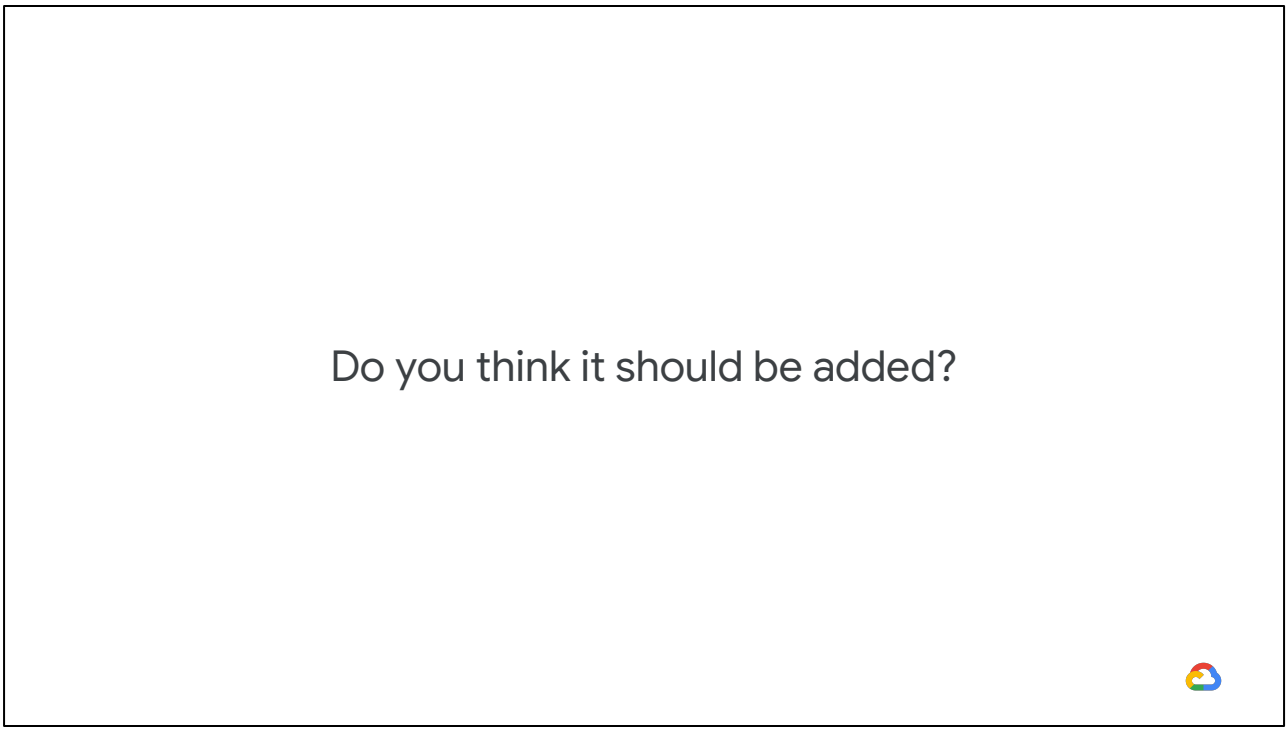
The purchasing system determines what's in the inventory, and the inventory is also logged in the data warehouse. The model is retrained daily, based on actual inventory, sales, and other signals.

The model is deployed to Cloud AI Platform for training.

The model can then be invoked on any new or updated data within the **Dataflow**.

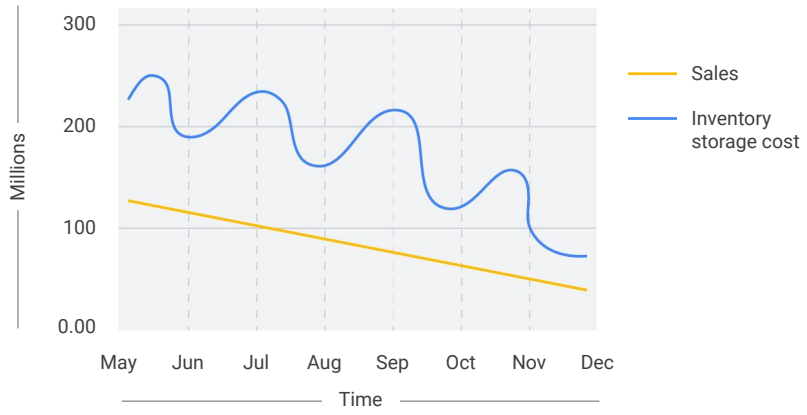


One day, your product manager has the idea to add a credit rating to each purchase order.



You’re the head of the machine learning engineering team. Do you think it should be added?

Scenario 1

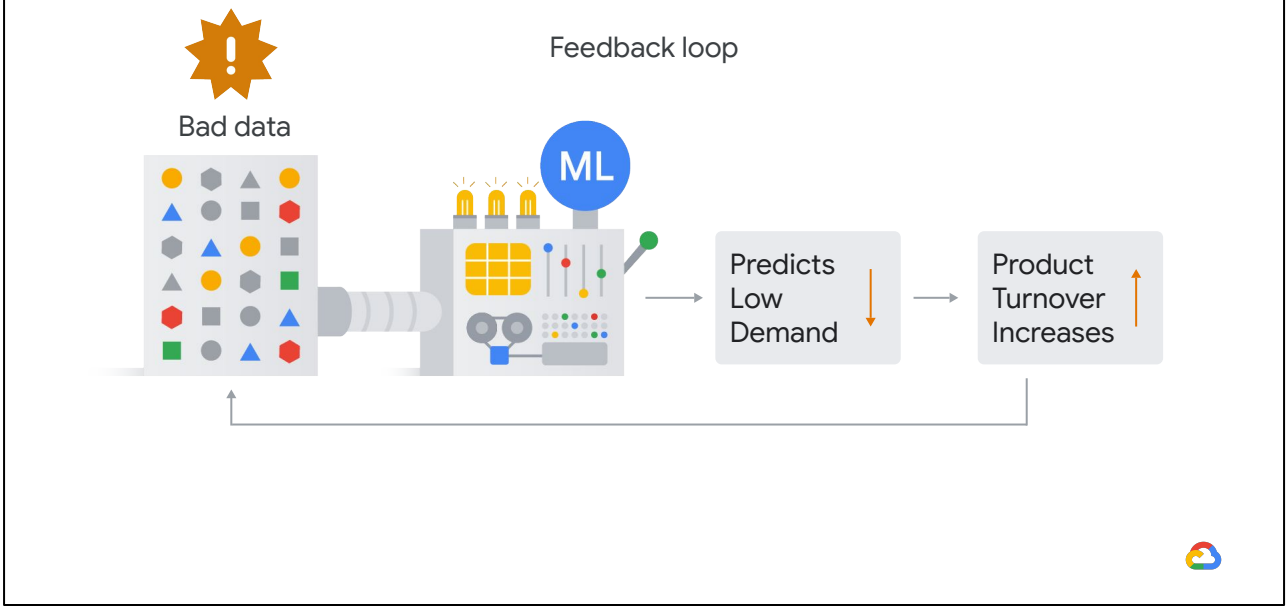


You get an email from the head of the business unit saying that he's just noticed that sales are down significantly. The warehouse manager, who is copied on the email, says that inventory storage costs are also down significantly.

The room is suddenly getting quite warm. What could have happened here?

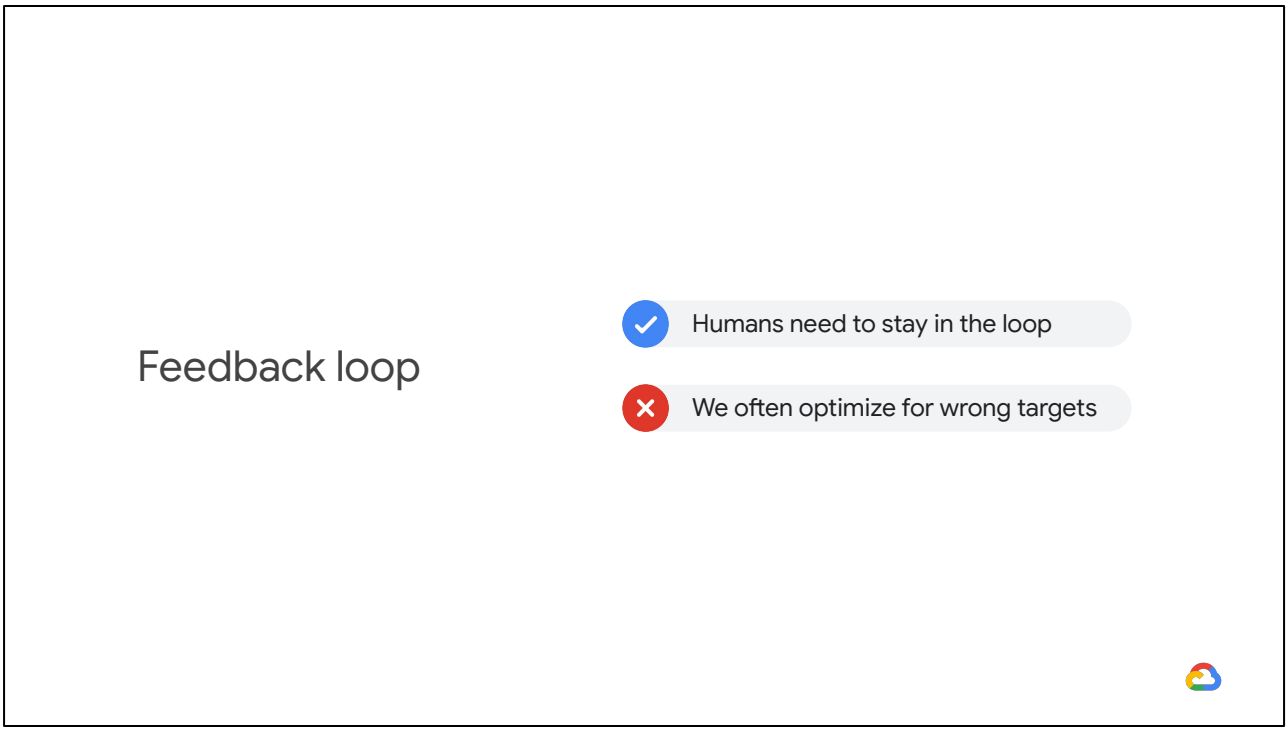
Feedback loop

This is a great example of a feedback loop!



What might have happened was that the model started under predicting demand, perhaps because of some corrupted historical data or an error in the pipeline.

Once demand started to go down, product turnover started to creep up. If this problem went unnoticed for a while, the model might have learned to keep zero inventory all the time!



In addition to being a great reminder that humans need to stay in the loop, it's also a reminder that we are often optimizing for something other than what we ultimately care about. In this case, we were optimizing for matching predicted demand when what we cared about was minimizing carrying costs in order to maximize profits.

Scenario 2

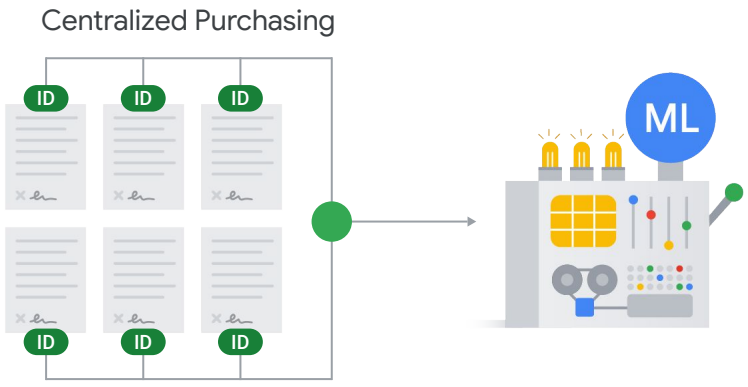
Here’s another scenario.

One of your salespeople just shared some amazing news. By leveraging their contacts at one of Megacorp’s many regional divisions, they signed Megacorp to a five-year deal and it’s the biggest contract yet!

Scenario 2

Great, you think, not realizing that this could have implications for your model's performance.

How can these be related?

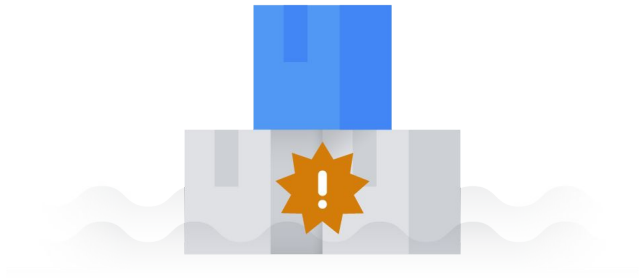


It all depends on how the sales orders come in, and how independent the divisions actually are.

If the divisions are entirely dependent (because there's actually just one purchasing decision, split up by division) and these orders come in separately, your model may still treat these orders as independent, in which case it would look much more compelling as evidence of an uptick in demand.

The solution here would be to add some aggregation by company ID in your pipeline before computing other statistics.

Scenario 3



Okay, let's look at one last scenario.

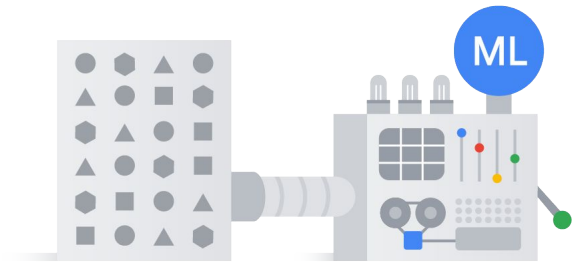
Your warehouse manager emails you and tells you that the warehouse flooded and they've had to scrap a large portion of the inventory.

Scenario 3

They've ordered replacements from purchasing, but it will be four days before those arrive and unfulfilled orders in the meantime will have to wait.

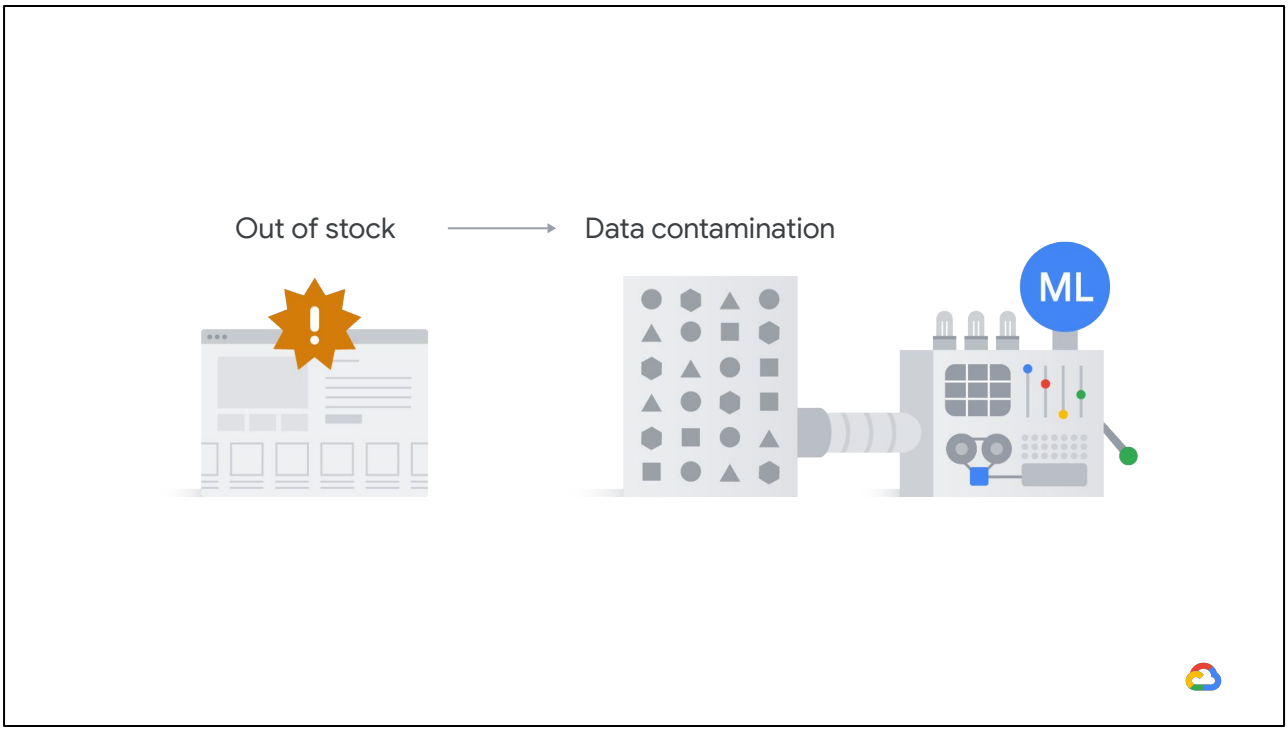
You realize that you have the skills to address this problem.

What do you do?



You stop your automatic model deployment process

You stop your automatic model deployment process.



The reason you do so is because data collected during this period will be contaminated. Since the products will show as out of stock on the website, customer orders will be low.