



Model Deployment

- “Deploy” is a loose term that generally means making model running and accessible
- To be deployed, model will have to leave the development environment
 - During model development, model usually runs in a development environment
 - can be deployed to a staging environment for testing
 - can be deployed to a production environment to be used by end users
- Production is a broad spectrum
 - For some teams, production means generating nice plots in notebooks to show to the business team
 - For other teams, production means keeping your models up and running for millions of users a day
- If your work is in the first scenario, production environment is similar to the development environment
- If your work is closer to the second scenario, welcome to world of model deployment!

Deployment is easy!

If you ignore all hard parts

- If want to deploy a model for friends to play with, all needs to be done is to
 - wrap predict function in a POST request endpoint using Flask or FastAPI,
 - put the dependencies this predict function needs to run in a container
 - push model and its associated container to a cloud service like AWS or GCP to expose the endpoint
- Can use this exposed endpoint for downstream applications:
 - e.g., when an application receives a prediction request from a user,
 - this request is sent to the exposed endpoint, which returns a prediction
- If familiar with the necessary tools, can have a functional deployment in an hour!

Machine Learning Deployment Myths

- Deploying an ML model can be very different from deploying a traditional software program
- This difference might cause people who have never deployed a model before
 - to either dread the process
 - or underestimate how much time and effort it will take
- Common myths about the deployment process
 - Myth 1: You Only Deploy One or Two ML Models at a Time
 - Myth 2: If We Don't Do Anything, Model Performance Remains the Same
 - Myth 3: You Won't Need to Update Your Models as Much
 - Myth 4: Most ML Engineers Don't Need to Worry About Scale

Deployment : Reality

- The hard parts include
 - making model available to millions of users
 - with a latency of milliseconds and 99% uptime,
 - setting up the infrastructure
 - so that the right person can be immediately notified when something goes wrong,
 - figuring out what went wrong
 - seamlessly deploying the updates to fix what's wrong
- In many companies,
 - the responsibility of deploying models falls into the hands of the same people who developed those models
- In many other companies, once
 - a model is ready to be deployed, it will be exported and handed off to another team to deploy it
- However, this separation of responsibilities can cause high overhead communications across teams and make it slow to update model.
 - It also can make it hard to debug should something go wrong.

Productionalization and Deployment

- A key component of MLOps
 - presents an entirely different set of technical challenges than developing the model
 - domain of the software engineer and the DevOps team
- Organizational challenges in managing the information exchange between the data scientists and these teams must not be underestimated!
 - without effective collaboration between the teams, delays or failures to deploy are inevitable

Model Deployment Types and Contents

what exactly is going into production, and what does a model consist of?

- Commonly two types of model deployment:
- Model-as-a-service, or live-scoring model
 - Typically the model is deployed into a simple framework to provide a REST API endpoint
 - (the means from which the API can access the resources it needs to perform the task)
 - that responds to requests in real time
- Embedded model
 - Here the model is packaged into an application, which is then published
 - A common example is an application that provides batch-scoring of requests
- What to-be-deployed models consist of depends, of course, on the technology chosen,
 - but typically they comprise a **set of code** (commonly Python, R, or Java) and **data artifacts**
 - can have **version dependencies on runtimes and packages** that need to match in the production environment
 - because the use of different versions may cause model predictions to differ

Model Deployment : Dependency Management

Model Export

- Can export the model to a portable format such as PMML, PFA, ONNX, or POJO
 - improves model portability between systems and simplify deployment
 - helps in reducing dependencies on the production environment
- Come at a cost
 - each format supports a limited range of algorithms,
 - sometimes the portable models behave in subtly different ways than the original
- Whether or not to use a portable format is a choice to be made based on a thorough understanding of the technological and business context.

Model Deployment: Dependency Management(2)

Containerization

- Containerization is an increasingly popular solution
 - to the headaches of dependencies when deploying ML models
- Container technologies such as Docker are lightweight alternatives to virtual machines,
 - allowing applications to be deployed in independent, self-contained environments,
 - matching the exact requirements of each model
- Also enable
 - new models to be seamlessly deployed using the blue-green deployment technique
 - scaling of compute resources for models
- Orchestrating many containers is the role of technologies such as Kubernetes
 - can be used both in the cloud and on-premise.

Model Deployment Requirements

What needs to be addressed – when moving from development to the production

- In customer-facing, mission-critical use cases, a more robust CI/CD pipeline is required
- Typically involves:
 - 1) Ensuring all coding, documentation and sign-off standards have been met
 - 2) Re-creating the model in something approaching the production environment
 - 3) Revalidating the model accuracy
 - 4) Performing explainability checks
 - 5) Ensuring all governance requirements have been met
 - 6) Checking the quality of any data artifacts
 - 7) Testing resource usage under load
 - 8) Embedding into a more complex application, including integration tests

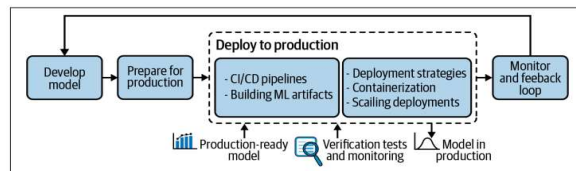
Model Deployment Requirements(2)

- One thing is for sure is needed:
 - **rapid, automated deployment is always preferred to labor-intensive processes**
- In heavily regulated industries (e.g., finance and pharmaceuticals),
 - governance and regulatory checks will be extensive
 - are likely to involve manual intervention
- The desire in MLOps, just as in DevOps, is to **automate the CI/CD pipeline as far as possible**
 - speeds up the deployment process
 - enables more extensive regression testing
 - reduces the likelihood of errors in the deployment



Deploying to Production

- Business leaders view rapid deployment of systems into production as key to maximizing business value
 - only true if deployment can be done smoothly and at low risk
- Let's dive into the concepts and considerations when deploying machine learning models to production
 - that impact and drive—the way MLOps deployment processes are built



Deployment to production highlighted in the larger context of the ML project life cycle

CI/CD Pipelines

A common acronym for continuous integration and continuous delivery (or put more simply, deployment)

- Forms a modern philosophy of agile software development and a set of practices and tools
 - to release applications more often and faster, while also better controlling quality and risk
- Ideas are decades old and already used to various extents by software engineers,
 - different people and organizations use certain terms in very different ways
- Essential to keep in mind that
 - these concepts should be tools to serve the purpose of delivering quality fast
 - first step is always to identify the specific risks present at the organization
- CI/CD methodology should be adapted based on the needs of the team and the nature of the business.

CI/CD for ML

- CI/CD concept apply just as well to machine learning systems
 - are a critical part of MLOps strategy
- An example of such pipeline could be:
 - Build the model
 - Build the model artifacts
 - Send the artifacts to long-term storage
 - Run basic checks (smoke tests/sanity checks)
 - Generate fairness and explainability reports
 - Deploy to a test environment
 - Run tests to validate ML performance, computational performance
 - Validate manually
 - Deploy to production environment
 - Deploy the model as canary
 - Fully deploy the model

CI/CD for ML(2)

- Many scenarios are possible, depend on the application,
 - the risks from which the system should be protected
 - and the way the organization chooses to operate
- Generally, an incremental approach to building a CI/CD pipeline is preferred:
 - a simple or even naive workflow on which a team can iterate
 - often much better than starting with complex infrastructure from scratch
- A starting project does not have the infrastructure requirements of a tech giant
 - can be hard to know up front which challenges deployments will present
- There are common tools and best practices,
 - but there is no one-size-fits-all CI/CD methodology
 - means the best path forward is starting from a simple (but fully functional) CI/CD workflow
 - then introducing additional or more sophisticated steps along the way as quality or scaling challenges appear

Building ML Artifacts

- The goal of a continuous integration pipeline is
 - to avoid unnecessary effort in merging the work from several contributors
 - also to detect bugs or development conflicts as soon as possible
- The very first step is using centralized version control systems
 - unfortunately, working for weeks on code stored only on a laptop is still quite common)
- The most common version control system is Git, an open source software
 - majority of software engineers across the world already use Git,
 - increasingly being adopted in scientific computing and data science
- Git allows for
 - maintaining a clear history of changes,
 - safe rollback to a previous version of the code,
 - multiple contributors to work on their own branches of the project before merging to the main branch, etc
- Git is appropriate for code, but not designed
 - to store other types of assets common in data science workflows, such as large binary files (for example, trained model weights),
 - or to version the data itself

ML Artifact

- Once code and data is in a centralized repository,
 - a testable and deployable bundle of the project must be built
 - are usually called **artifacts** in the context of CI/CD
- Each of the following elements needs to be bundled into an artifact
 - that goes through a testing pipeline and is made available for deployment to production:
 - Code for the model and its preprocessing
 - Hyperparameters and configuration
 - Training and validation data
 - Trained model in its runnable form
 - An environment including libraries with specific versions, environment variables, etc.
 - Documentation
 - Code and data for testing scenarios

The Testing Pipeline

- Testing pipeline can validate a wide variety of properties of the model contained in the artifact
 - good tests should make it as easy as possible to diagnose the source issue when they fail
- Automating tests as much as possible is essential and is a key component of efficient MLOps
- A lack of automation or speed wastes time,
 - also discourages the development team from testing and deploying often,
 - which can delay the discovery of bugs or design choices that make it impossible to deploy to production

Deployment concepts

- Integration
 - Process of merging a contribution to a central repository and performing more or less complex tests
 - typically merging a Git feature branch to the main branch
- Delivery
 - Same as used in the continuous delivery (CD) part of CI/CD,
 - Process of building a fully packaged and validated version of the model ready to be deployed to production
- Deployment
 - Process of running a new model version on a target infrastructure
 - Fully automated deployment is not always practical or desirable
- Release
 - In principle, release is yet another step, directing production workload to model
 - deploying a model version (even to the production infrastructure) does not necessarily mean that the production workload is directed to the new version
 - multiple versions of a model can run at the same time on the production infrastructure

Categories of Model Inferences

Two ways to approach model deployment

- Batch scoring,
 - where whole datasets are processed using a model, such as in daily scheduled jobs
- Real-time scoring,
 - where one or a small number of records are scored,
 - such as when an ad is displayed on a website and a user session is scored by models to decide what to display
- In some systems, scoring on one record is technically identical to requesting a batch of one!
- In both cases, multiple instances of the model can be deployed
 - to increase throughput and potentially lower latency

Considerations When Sending Models to Production

- When sending a new model version to production, first consideration is often **to avoid downtime**,
 - in particular for real-time scoring
- Blue-green or red-black— deployment
 - basic idea is that rather than shutting down the system, upgrading it, and then putting it back online,
 - a new system can be set up next to the stable one
 - and when it's functional, the workload can be directed to the newly deployed version
 - and if it remains healthy, the old one is shut down)
- Canary deployment
 - idea is that the stable version of the model is kept in production,
 - but a certain percentage of the workload is redirected to the new model, and results are monitored
 - usually implemented for real-time scoring, but a version of it could also be considered for batch

Maintenance in Production

Once a model is released, it must be maintained

- At a high level, there are three maintenance measures:
 - Resource monitoring
 - Health check
 - ML metrics monitoring
- Resource monitoring
 - Just as for any application running on a server, collecting IT metrics such as CPU, memory, disk, or network usage
 - can be useful to detect and troubleshoot issues
- Health check
 - Need to check if the model is indeed online and to analyze its latency
 - simply queries the model at a fixed interval (on the order of one minute) and logs the results
- ML metrics monitoring
 - about analyzing the accuracy of the model and comparing it to another version or detecting when it is going stale
 - may require heavy computation, this is typically lower frequency
- Finally, when a malfunction is detected, a **rollback** to a previous version may be necessary
 - critical to have the rollback procedure ready and as automated as possible;
 - testing it regularly can make sure it is indeed functional

Deployment Patterns

Pravin Y Pawar

Adapted from "Machine Learning Engineering"
By Andriy Burkov

Model Deployment Patterns

- Once the model has been built and thoroughly tested, it can be deployed
 - means to make it available for accepting queries generated by the users of the production system
- Once the production system accepts the query, the latter is transformed into a feature vector
 - The feature vector is then sent to the model as input for scoring
 - The result of the scoring then is returned to the user
- A trained model can be deployed in various ways
 - can be deployed on a server, or on a user's device
 - can be deployed for all users at once, or to a small fraction of users
- A model can be deployed following several patterns:
 - statically, as a part of an installable software package,
 - dynamically on the user's device,
 - dynamically on a server, or
 - via model streaming

Static Deployment

- Very similar to traditional software deployment
 - prepare an installable binary of the entire software
 - model is packaged as a resource available at the runtime
- Depending on the operating system and the runtime environment
 - Objects of both the model and the feature extractor can be packaged as a
 - part of a dynamic-link library (DLL on Windows),
 - Shared Objects (*.so files on Linux),
 - or be serialized and saved in the standard resource location for virtual machine-based systems,
 - such as Java and .Net.

Static Deployment(2)

Pros and Cons

- Many advantages:
 - software has direct access to the model, so the execution time is fast for the user
 - user data doesn't have to be uploaded to the server at the time of prediction
 - saves time and preserves privacy
 - model can be called when the user is offline
 - software vendor doesn't have to care about keeping the model operational
 - becomes the user's responsibility
- Several drawbacks:
 - The separation of concerns between the machine learning code and the application code isn't always obvious
 - makes it harder to upgrade the model without also having to upgrade the entire application
 - If the model has certain computational requirements for scoring (such as access to an accelerator or a GPU)
 - may add complexity and confusion as to where the static deployment can or cannot be used

Dynamic Deployment on User's Device

- Similar to a static deployment, in the sense the user runs a part of the system as a software application on their device
- Difference is that in dynamic deployment, the model is not part of the binary code of the application
- **Achieves better separation of concerns!**
 - Pushing model updates is done without updating the whole application running on the user's device
- Dynamic deployment can be achieved in several ways:
 - by deploying model parameters,
 - by deploying a serialized object, and
 - by deploying to the browser

Deployment of Model Parameters

- In this deployment scenario, the model file only contains the learned parameters
 - user's device has installed a runtime environment for the model
- Some machine learning packages, like TensorFlow,
 - have a lightweight version that can run on mobile devices
- Alternatively, frameworks such as Apple's Core ML allow running models on Apple devices
 - created using popular packages, including scikit-learn, Keras, and XGBoost

Deployment of a Serialized Object

- Model file is a serialized object that the application would deserialize
- The advantage is that don't need to have a runtime environment for model on the user's device
 - all needed dependencies will be deserialized with the object of the model
- An evident drawback is that an update might be quite "heavy,"
 - which is a problem if your software system has millions of users

Deploying to Browser

- Most modern devices have access to a browser, either desktop or mobile
- Some machine learning frameworks, such as TensorFlow.js,
 - have versions that allow to train and run a model in a browser, by using JavaScript as a runtime
- Even possible to train a TensorFlow model in Python,
 - then deploy it to, and run it in the browser's JavaScript runtime environment
 - if a GPU (graphics processing unit) is available on the client's device, Tensorflow.js can leverage it

Dynamic Deployment on User's Device(2)

Advantages and Drawbacks

- Advantages
 - Calls to the model will be fast for the user
 - Reduces the impact on the organization's servers, as most computations are performed on the user's device
- If the model is deployed to the browser,
 - advantage is organization's infrastructure only needs to serve a web page that includes the model's parameters
 - A downside is bandwidth cost and application startup time might increase.
 - users must download the model's parameters each time they start the web application
 - as opposed to doing it only once when they install an application

Dynamic Deployment on User's Device(3)

Advantages and Drawbacks

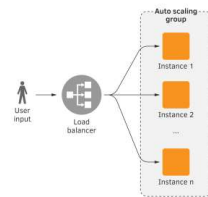
- Monitoring
 - Deploying to a user's device makes it difficult to monitor the model performance
- Model updates
 - A serialized object can be quite voluminous
 - Some users may be offline during the update, or even turn off all future updates
 - may end up with different users using very different model versions
 - becomes difficult to upgrade the server-side part of the application
- Third-party analyses
 - Deploying models on the user's device means that the model easily becomes available for third-party analyses
 - may try to reverse-engineer the model to reproduce its behavior
 - may search for weaknesses by providing various inputs and observing the output
 - may adapt their data so the model predicts what they want

Dynamic Deployment on a Server

- Because of the complications with other approaches, and problems with performance monitoring,
- Most frequent deployment pattern is to place the model on a server (or servers),
 - make it available as
 - a Representational State Transfer application programming interface (REST API) in the form of a web service,
 - Google's Remote Procedure Call (gRPC) service
- Four common practices
 - Deployment on a Virtual Machine
 - Deployment in a Container
 - Serverless Deployment
 - Model Streaming

Deployment on a Virtual Machine(VM)

- In a typical web service architecture deployed in a cloud environment
 - predictions are served in response to canonically-formatted HTTP requests
- A web service running on a virtual machine
 - receives a user request containing the input data,
 - calls the machine learning system on that input data
 - then transforms the output of the machine learning system into the output JSON or XML string
- To cope with high load, several identical VMs are running in parallel
 - A load balancer dispatches the incoming requests to a specific virtual machine
 - VMs can be added and closed manually, or be a part of an autoscaling group that launches
 - VMs can be terminated virtual machines based on their usage



: Deploying a machine learning model as a web service on a virtual machine.

Deployment on a Virtual Machine(VM)2

- In Python, a REST API web service is usually implemented
 - using a web application framework such as Flask or FastAPI
- TensorFlow, a popular framework used to train deep models,
 - comes with TensorFlow Serving, a built-in gRPC service
- Advantage: Architecture of the software system is conceptually simple: a typical web or gRPC service
- Downsides
 - Need to maintain servers (physical or virtual)
 - If virtualization is used, there is an additional computational overhead due to virtualization and running multiple OS
 - Network latency, which can be a serious issue, depending on how fast you need to process scoring results
 - has a relatively higher cost, compared to deployment in a container, or a serverless deployment

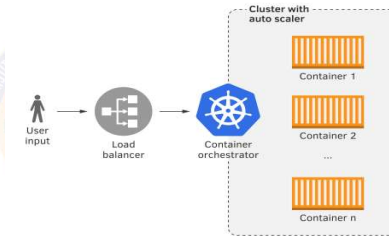
Deployment in a Container

- A more modern alternative to a virtual-machine-based deployment
 - considered more resource-efficient and flexible than with virtual machines
- A container is similar to a virtual machine
 - in the sense that it is also an isolated runtime environment with its own filesystem, CPU, memory, and process space
- The main difference, however, is that all containers are running on the same virtual or physical machine
 - share the operating system, while each virtual machine runs its own instance of the operating system
- Deployment Process
 - The machine learning system and the web service are installed inside a container
 - Usually, a container is a Docker container, but there are alternatives
 - Then a container-orchestration system is used to run the containers on a cluster of physical or virtual servers
 - A typical choice of a container-orchestration system for running on-premises or in a cloud platform, is Kubernetes
 - Some cloud platforms provide both their own container-orchestration engine, such as AWS Fargate and Google Kubernetes Engine

Deployment in a Container(2)

Organization

- Virtual or physical machines are organized into a cluster,
 - whose resources are managed by the container orchestrator
- New virtual or physical machines can be manually added to the cluster, or closed
- If your software is deployed in a cloud environment,
 - a cluster autoscaler can launch (and add to the cluster) or terminate virtual machines
 - based on the usage of the cluster.



Deploying a model as a web service in a container running on a cluster.

Deployment in a Container(3)

- Advantage
 - More resource-efficient as compared to the deployment on a virtual machine
 - Allows the possibility to automatically scale with scoring requests
 - Allows us to scale-to-zero- reduced down to zero replicas when idle and brought back up if there is a request to serve
 - the resource consumption is low compared to always running services
 - leads to less power consumption and saves cost of cloud resources
- Drawback
 - Containerized deployment is generally seen as more complicated, and requires expertise

Serverless Deployment

- Several cloud services providers, including Amazon, Google, and Microsoft, offers serverless computing
 - Lambda-functions on Amazon Web Services, and Functions on Microsoft Azure and Google Cloud Platform
- The serverless deployment consists of preparing a zip archive
 - with all the code needed to run the machine learning system (model, feature extractor, and scoring code)
 - must contain a file with a specific name that contains a specific function
 - is uploaded to the cloud platform and registered under a unique name
- The cloud platform provides an API to submit inputs to the serverless function
 - specifies its name, provides the payload, and yields the outputs
- The cloud platform takes care of
 - deploying the code and the model on an adequate computational resource,
 - executing the code,
 - routing the output back to the client

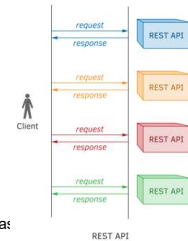
Serverless Deployment(2)

Advantages and Limitations

- Advantages to relying on serverless deployment
 - don't have to provision resources such as servers or virtual machines
 - don't have to install dependencies, maintain, or upgrade the system
 - highly scalable and can easily and effortlessly support thousands of requests per second
 - support both synchronous and asynchronous modes of operation
 - cost-efficient: only pay for compute-time
 - simplifies canary deployment, or canarying
 - Rollbacks are also very simple in the serverless deployment because it is easy to switch back to the previous version of the function
- Limitations
 - Restrictions by the cloud service provider
 - the function's execution time, zip file size, and amount of RAM available on the runtime
 - Zip file size limit can be a challenge - A typical model requires multiple heavyweight dependencies
 - Unavailability of GPU access can be a significant limitation for deploying deep models

Model Serving – REST API Revisited

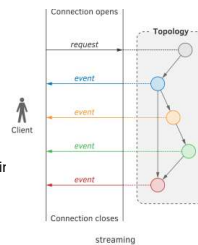
- In complex systems, there can be many models applied to the same input
 - a model can input a prediction from another model
- For example, the input may be a news article
 - One model can predict the topic of the article
 - Another model can extract named entities
 - Third model can generate a summarization of the article, and so on
- According to the REST API deployment pattern, need one REST API per model
 - client would call one API by sending a news article as a part of the request - get the topic as response
 - client calls another API by sending a news article, and gets the named entities as response; etc.




Model Streaming

Can be seen as an inverse to the REST API

- Streaming works differently
 - All models, as well as the code needed to run them, are registered within a stream-processing engine (SPE)
 - Apache Storm, Apache Spark, and Apache Flink
 - Or, they are packaged as an application based on a stream-processing library (SPL), such as Apache Samza, Apache Kafka Streams, and Akka Streams
- Based on notion of data processing topology
 - Input data flows in as an infinite stream of data elements sent by the client
 - Following a predefined topology, each data element in the stream undergoes a transformation in the nodes of the topology
 - Transformed, the flow continues to other nodes.
- In a stream-processing application, nodes transform their input in some way, then either,
 - send the output to other nodes, or
 - send the output to the client, or
 - persist the output to the database or a filesystem.



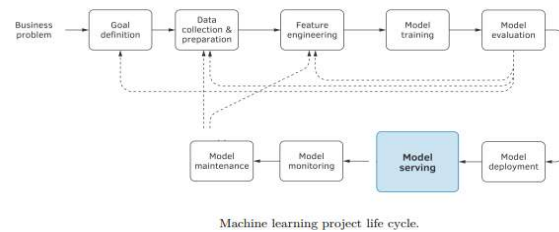

BITS Pilani
 Pilani | Dubai | Goa | Hyderabad

Essentials of Model Serving

Pravin Y Pawar

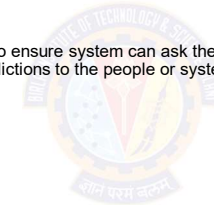
Adapted from "Machine Learning Engineering" and "Reliable Machine Learning"

Model Serving in ML Lifecycle



Model Serving

- Model built in training phase, needs to be taken into the world so that it can start predicting!
- Aka Model Serving
- Process of creating a structure to ensure system can ask the model to make predictions on new examples, and return those predictions to the people or systems that need them



Properties of the Model Serving Runtime

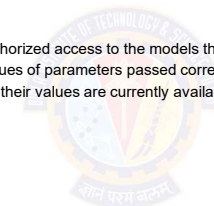
- The model serving runtime is the environment in which the model is applied to the input data
 - The runtime properties are dictated by the model deployment pattern
- However, an effective runtime will have several additional properties such as
 - Security and Correctness
 - Ease of Deployment
 - Guarantees of Model Validity
 - Ease of Recovery
 - Avoidance of Training/Serving Skew
 - Avoidance of Hidden Feedback Loops



Properties of the Model Serving Runtime(2)

Security and Correctness

- The runtime is responsible for authenticating the user's identity, and authorizing their requests.
- Things to check are:
 - whether a specific user has authorized access to the models they want to run,
 - whether the names and the values of parameters passed correspond to the model's specification,
 - whether those parameters and their values are currently available to the user.



Properties of the Model Serving Runtime(3)

Ease of Deployment and Recovery

- Ease of Deployment
 - The runtime must allow the model to be updated with minimal effort
 - ideally, without affecting the entire application
 - If the model was deployed as a web service on a physical server,
 - then a model update must be as simple as replacing one model file with another, and restarting the web service
 - If the model was deployed as a virtual machine instance or container,
 - then the instances or containers running the old version of the model should be replaceable by gradually stopping the running instances and starting new instances from a new image
 - The same principle applies to the orchestrated containers
- Ease of Recovery
 - An effective runtime allows easy recovery from errors by rolling back to previous versions
 - The recovery from an unsuccessful deployment should be produced in the same way, and with the same ease, as the deployment of an updated model
 - The only difference is that, instead of the new model, the previous working version will be deployed.

Properties of the Model Serving Runtime(4)

Avoidance of Training/Serving Skew

- When it concerns feature extraction, Strongly recommended to avoid using two different codebases,
 - one for training the model, and one for scoring in production
 - even a tiny difference between two versions of feature extractor code may lead to suboptimal or incorrect model performance
- The engineering team may reimplement the feature extractor code for production for many reasons
 - most common being data analyst's code is inefficient or incompatible with the production ecosystem
- Runtime should allow easy access to the feature extraction code for various needs,
 - including model retraining, ad-hoc model calls, and production.
- One way to implement it is by wrapping the feature extraction object into a separate web service
- If cannot avoid using two different codebases to generate features for training and production,
 - then the runtime should allow for the logging of feature values generated in the production environment
 - Those values should then be used as training values

Key questions for model serving

- Lot of ways to create structures around the model that support serving,
- Each with very different sets of trade-offs
- Useful to think through specific questions about needs of system
 - What will be the load to model?
 - What are the prediction latency needs of model?
 - Where does the model need to live?
 - What are the hardware needs for model?
 - How will the serving model be stored, loaded, versioned and updated?
 - What will feature pipeline for serving look like?

What will be the load to model?

- QPS (Queries Per second) – need to know in serving environment what is the level of traffic that model will be asked to handle – when queries are done on demand
 - Model serving predictions to millions of daily users may need handle thousands of QPS
 - Model runs audio recognizer on mobile device may run at a few QPS
 - Model predicting real estate prices might not be served on demand at all!
- Few basic strategies to handle large traffic loads
 - Replicate model across many machines and run these parallel – may be on cloud
 - Use more powerful hardware – accelerators like GPU or specialized chips
 - Tune computation cost of model itself by using fewer features or layers or parameters
 - Model cascades can be effective at cost reduction

What are the prediction latency needs of model?

- Prediction latency is time between the moment request is made and moment response is received
- Acceptable prediction latency can vary dramatically among applications
 - Is major determiner of serving architecture choices
- Taken together, latency and traffic load define overall computational needs of ML system
- If latency is too high, can be mitigated by
 - Using more powerful hardware
 - Making model less expensive to compute
- But creating a larger number of model replicas is not a solution for latency issue!

Where does the model need to live?

Model needs to be stored on physical device in specific location – home of model

- This choice has significant implications on overall serving architecture
- On a local machine
 - Not a practical solution – may be suitable for small batch predictions
 - Not recommended beyond small-scale prototyping or bespoke uses
- On servers owned or managed by our organization
 - Important when specific privacy or security concerns are in place
 - Right option if latency is hypercritical concern or if speciality hardware is needed to run models
 - Can limit flexibility in terms of scaling up/down, require special attention to monitoring
- In the cloud
 - Can allow easy scaling overall computation footprint up or down
 - Can be done by
 - running model servers on own virtual servers and controlling how many of them to be used
 - managed inference service
- On-device
 - Everything from mobile phones to smart watches, digital assistants, automobiles, printers etc.
 - Has strict constraints on model size, because of limited memory and power

What are the hardware needs for model?

- Range of computational hardware and chip options have emerged
 - Enabled dramatic improvements in serving efficiency for various model types
- Multicore CPUs
 - Suitable for non-deep methods, non deep matrix multiplications
- Hardware accelerators – commonly GPU
 - Choice of serving deep learning models as they involve dense matrix multiplications
 - But has drawbacks
 - Specialized hardware – need to invest or use a cloud service – costly options
 - Not suited for operations not involving large amounts of dense matrix calculations

How will the serving model be stored, loaded, versioned and updated?

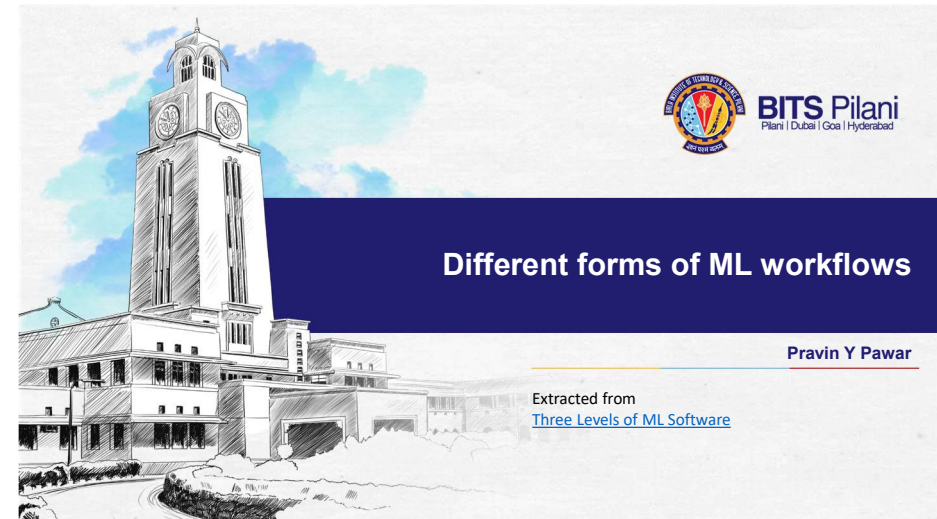
Model is physical object – has size and needs space

- Model serving in offline environment be stored on disk and loaded by specific binaries in batch jobs
 - Main requirement is disk space to store model and I/O capacity to load model from disk and RAM needed to load model into memory for use
- Model used in online serving needs to be stored in RAM in dedicated machine
 - For high-throughput services in latency critical settings, copies of these models likes to be stored and served in many replica machine in parallel
- Eventually these models needs to be updated by retraining
- means needs to swap version of model currently used in serving on a given machine with a new version
- Deciding exactly how many versions to be supported and at what capacity is important architectural choice
 - Requires balancing resourcing, system complexity and organizational requirements together

What will feature pipeline for serving look like?

Feature needs to be processed at serving time as well as at training time

- Any feature processing or other data manipulation that is done to data at training time will almost certainly need to be repeated for all examples sent to model at serving time
 - Computational requirements for this may be considerable
- Actual code used to turn incoming examples data to features of model may be different at serving time from the code used for similar tasks at training time
 - Main source of classic training-serving skew and bugs notoriously difficult to detect and debug
- Promise is in form of feature stores – handle both training and serving together in a single package
- Creating feature for model to use at serving time is key source of latency
 - Means serving feature pipeline is far from an afterthought
 - But is indeed often most production-critical part of the entire serving stack



Different forms of ML workflows

- Operating an ML model might assume **several architectural styles**
- Primarily **Four architectural patterns** which are classified along two dimensions:
 - ML Model Training
 - ML Model Prediction
- For sake of simplicity disregard the third dimension - ML Model Type
 - which denotes the type of machine learning algorithm such as
 - Supervised
 - Unsupervised
 - Semi-supervised
 - and Reinforcement Learning

Model Training Patterns

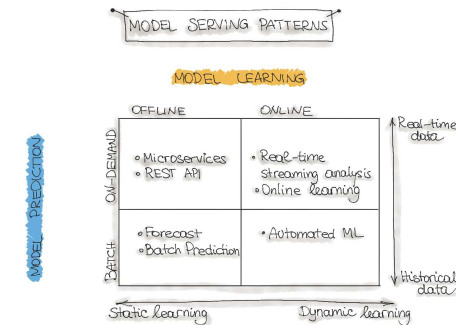
- Two ways to perform ML Model Training:
 - Offline learning
 - Online learning
- Offline learning (aka batch or static learning)
 - The model is trained on a set of already collected data
 - After deploying to the production environment, the ML model remains constant until it re-trained
 - Model will see a lot of real-live data and becomes stale - 'model decay' and should be carefully monitored
- Online learning (aka dynamic learning)
 - The model is regularly being re-trained as new data arrives, e.g. as data streams
 - Usually the case for ML systems that use time-series data, such as sensor, or stock trading data to accommodate the temporal effects in the ML model

Model Prediction Patterns

- Two modes to denote the mechanics of the ML model to makes predictions
 - Batch predictions
 - Real-time predictions
- Batch predictions
 - The deployed ML model makes a set of predictions based on historical input data
 - often sufficient for data that is not time-dependent, or when it is not critical to obtain real-time predictions as output
- Real-time predictions (aka on-demand predictions)
 - Predictions are generated in real-time using the input data that is available at the time of the request

ML architecture patterns

Forecast, Web-Service, Online Learning, and AutoML



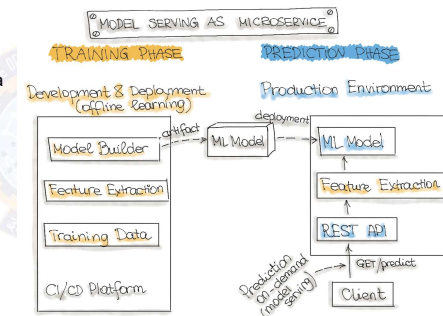
Forecast

- Widely spread in academic research or data science education (e.g., Kaggle or DataCamp)
 - used to experiment with ML algorithms and data as it is the easiest way to create a machine learning system
 - not very useful in an industry setting for production systems (e.g. mobile applications)
- Usually involves steps
 - take an available dataset
 - train the ML model
 - run this model on another (mostly historical) data
 - makes predictions

Web-Service (or Microservices)

Architecture for wrapping trained models as deployable services

- The most commonly described deployment architecture for ML models
- The web service takes input data and outputs a prediction for the input data points
 - Model is trained offline on historical data, but it uses real-live data to make predictions
 - Model remains constant until it is re-trained and re-deployed into the production system.
- The difference from a forecast (batch predictions) is that
 - the ML model runs near real-time
 - handles a single record at a time instead of processing all the data at once



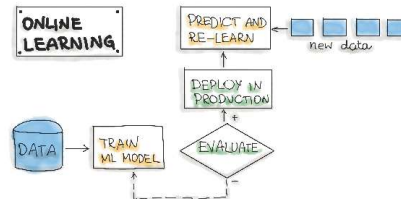
Online Learning (Real-time streaming analytics)

Most dynamic way to embed machine learning into a production system

- Confusing name because the core learning or ML model training is usually not performed on the live system
 - call it incremental learning- term online learning is already established within the ML community

- In this type of ML workflow

- ML learning algorithm is continuously receiving a data stream, either as single data points or in small groups called mini-batches
- System learns about new data on the fly as it arrives, so the ML model is incrementally being re-trained with new data
- Continually re-trained model is instantly available as a web service
- Technically works well with the lambda architecture in big data systems
- Model would typically run as a service on a Kubernetes cluster or similar





- A big difficulty with the online learning system in production is that
 - if bad data is entering the system, the ML model, as well as the whole system performance, will increasingly decline

AutoML

Sophisticated version of online learning

- AutoML is getting a lot of attention
 - considered the next advance for enterprise ML
 - promises training ML models with minimal effort and without machine learning expertise
 - User needs to provide data, and the AutoML system automatically selects an ML algorithm and configures the selected algorithm
 - very experimental way to implement ML workflows
 - usually provided by big cloud providers, such as Google or MS Azure
 - Instead of updating the model, need to execute an entire ML model training pipeline in production that results in new models on the fly





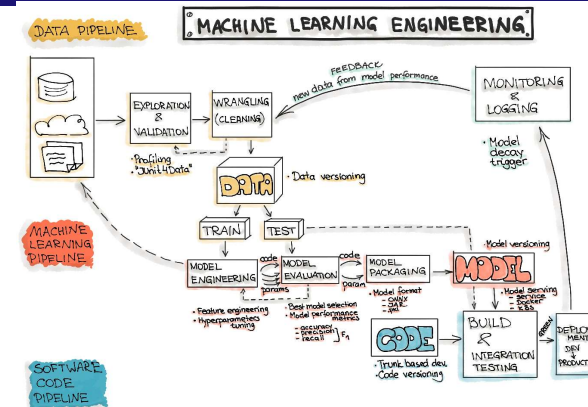
BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Model Serving Patterns

Pravin Y Pawar

Extracted from
[Three Levels of ML Software](#)

Machine Learning Workflow



Code: Deployment Pipelines

- The final stage of delivering an ML project includes the following three steps:
 - Model Serving
 - Model Performance Monitoring
 - Model Performance Logging
- Model Serving
 - The process of deploying the ML model in a production environment
- Model Performance Monitoring
 - The process of observing the ML model performance based on live and previously unseen data, such as prediction or recommendation
 - interested in ML-specific signals, such as prediction deviation from previous model performance
 - signals might be used as triggers for model re-training
- Model Performance Logging
 - Every inference request results in a log-record.

Model Serving Patterns

- Three components should be considered when ML model is served in a production environment
 - The inference is the process of getting data to be ingested by a model to compute predictions
 - requires a model, an interpreter for the execution, and input data
- Deploying an ML system to a production environment includes two aspects,
 - First deploying the pipeline for automated retraining and ML model deployment
 - Second, providing the API for prediction on unseen data
- Model serving is a way to integrate the ML model in a software system
- Five patterns to put the ML model in production:
 - Model-as-Service
 - Model-as-Dependency
 - Precompute
 - Model-on-Demand
 - and Hybrid-Serving

Model Serving Patterns(2)

Machine Learning Model Serving Taxonomy

Machine Learning Model Serving Taxonomy			
	ML Model		
Service & Versioning	Together with the consuming application	Independent from the consuming application	
Compile/ Runtime Availability	Build & runtime available	Available remotely through REST API/RPC	Available at the runtime scope
Serving Patterns	"Model-as-Dependency"	"Model-as-Service"	"Precompute" and "Model on Demand"
	Hybrid Model Serving (Federated Learning)		

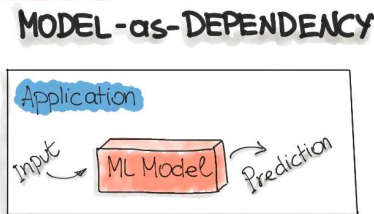
Model-as-Service

- A common pattern for wrapping an ML model as an independent service
 - can wrap the ML model and the interpreter within a dedicated web service
 - applications can request through a REST API or consume as a gRPC service
- Used for various ML workflows
 - Forecast
 - Web Service
 - Online Learning



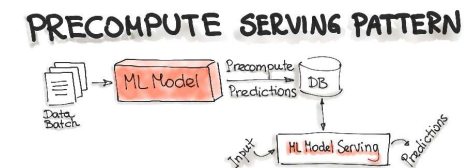
Model-as-Dependency

- Probably the most straightforward way to package an ML model
 - mostly used for implementing the Forecast pattern.
- A packaged ML model is considered as a dependency within the software application
 - For example, the application consumes the ML model like a conventional jar dependency by invoking the prediction method and passing the values
 - The return value of such method execution is some prediction that is performed by the previously trained ML model



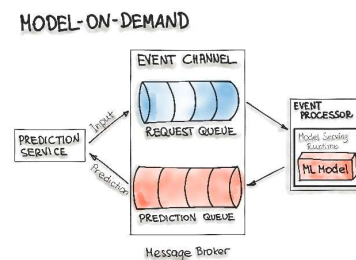
Precompute Serving Pattern

- Tightly related to the Forecast ML workflow
- Use an already trained ML model and precompute the predictions for the incoming batch of data
 - Resulting predictions are persisted in the database
 - For any input request, query the database to get the prediction result



Model-on-Demand

- Treats the ML model as a dependency that is available at runtime
 - contrary to the Model-as-Dependency pattern, has its own release cycle and is published independently
- Message-broker architecture is typically used for such on-demand model serving
 - contains two main types of architecture components:
 - a broker component
 - an event processor component
 - Broker component is the central part that contains the event channel that are utilised within the event flow
 - The event channels, which are enclosed in the broker component, a message queues
 - Message broker allows one process to write prediction-requests in a input queue
 - Event processor contains the model serving runtime and the ML model
 - connects to the broker, reads these requests in batch from the queue and sends them to the model to make the predictions
- The model serving process runs the prediction generation on the input data
 - writes the resulted predictions to the output queue
 - queued prediction results are pushed to the prediction service that initiated the prediction request

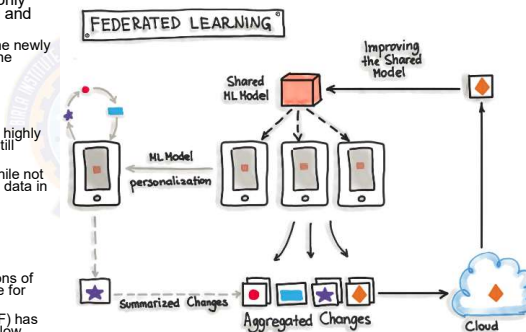


Hybrid-Serving (Federated Learning)

- Unique in the way it does, there is not only one model that predicts the outcome, but there are also lots of it
 - Exactly spoken there are as many models as users exist, in addition to the one that's held on a server
- Start with the unique model, the one on the server
 - Model on the server-side is trained only once with the real-world data
 - sets the initial model for each user
 - a relatively general trained model so it fits for the majority of users
- On the other side, there are the user-side models - really unique models
 - possible for the devices to train their own models due to hardware enhancements
 - devices will train their own highly specialized model for their own user
 - Once in a while, the devices send their already trained model data (not the personal data) to the server
- Then the server model will be adjusted
 - the actual trends of the whole user community will be covered by the model
 - this updated server model is set to be the new initial model that all devices are using

Hybrid-Serving (Federated Learning) 2

- Not having any downsides for the users, while the server model gets updated, this happens only when the device is idle, connected to WiFi and charging
 - testing is done on the devices, therefore the newly adopted model from the server is sent to the devices and tested for functionality
- Big benefit
 - data used for training and testing, which is highly personal, never leaves the devices while still capturing all data that is available
 - possible to train highly accurate models while not having to store tons of (probably personal) data in the cloud
- Constraints
 - mobile devices are less powerful
 - the training data is distributed across millions of devices and these are not always available for training
 - Exactly for this TensorFlow Federated (TFF) has been created - lightweight form of TensorFlow



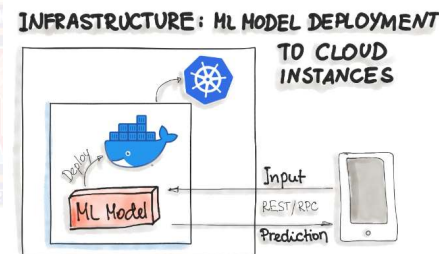
Deployment Strategies

- Common ways for wrapping trained models as deployable services, namely deploying ML models as
 - Docker Containers to Cloud Instances
 - Serverless Functions



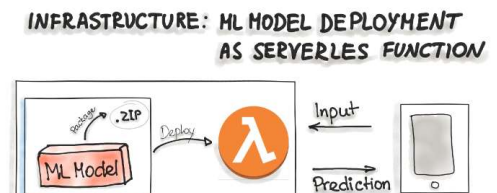
Deploying ML Models as Docker Containers

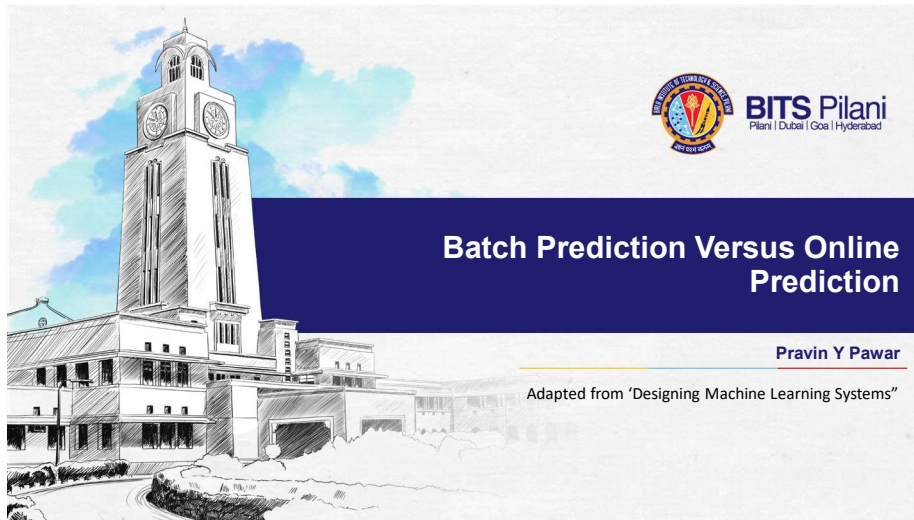
- No standard, open solution to ML model deployment!
- Containerization becomes the de-facto standard for delivery
 - as ML model inference being considered stateless, lightweight, and idempotent
 - means deploy a container that wraps an ML model inference code
- Docker is considered to be de-facto standard containerization technology
 - for on-premise, cloud, or hybrid deployments
- One ubiquitous way is to package the whole ML tech stack (dependencies) and the code for ML model prediction into a Docker container
 - Then Kubernetes or an alternative (e.g. AWS Fargate) does the orchestration
 - The ML model functionality, such as prediction, is then available through a REST API (e.g. implemented as Flask application)



Deploying ML Models as Serverless Functions

- Various cloud vendors already provide machine-learning platforms - can deploy model with their services
 - Amazon AWS Sagemaker, Google Cloud AI Platform, Azure Machine Learning Studio, and IBM Watson Machine Learning
 - Commercial cloud services also provide containerization of ML models such as AWS Lambda and Google App Engine serviet host
- In order to deploy an ML model as a serverless function,
 - the application code and dependencies are packaged into .zip files, with a single entry point function
 - could be managed by major cloud providers such as Azure Functions, AWS Lambda, or Google Cloud Functions
 - However, attention should be paid to possible constraints of the deployed artifacts such as the size of the artifacts



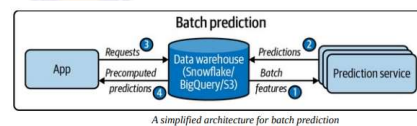


Three main modes of prediction

- Batch prediction, which uses only batch features
- Online prediction that uses only batch features (e.g., precomputed embeddings)
 - Aka on-demand prediction
- Online prediction that uses both batch features and streaming features
 - aka streaming prediction

Batch prediction

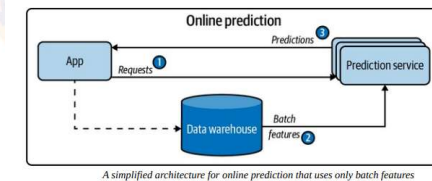
- When predictions are generated periodically or whenever triggered
 - Predictions are stored somewhere, such as in SQL tables or an in-memory database, and retrieved as needed
- For example,
 - Netflix might generate movie recommendations for all of its users every four hours,
 - the precomputed recommendations are fetched and shown to users when they log on to Netflix
- AKA asynchronous prediction: predictions are generated asynchronously with requests



Online prediction

Using only batch features

- When predictions are generated and returned as soon as requests for these predictions arrive
 - For example, enter an English sentence into Google Translate and get back its French translation immediately
 - aka on-demand prediction
- Traditionally, when doing online prediction, requests are sent to the prediction service via RESTful APIs
 - aka synchronous prediction: predictions are generated in synchronization with requests



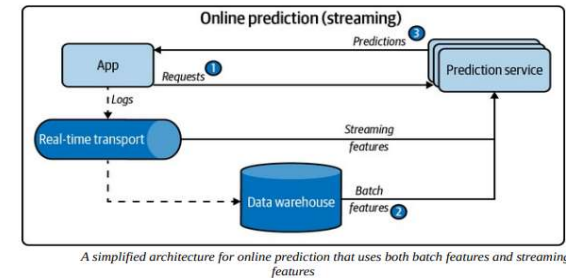
Online prediction

Using both batch and streaming features - "streaming prediction"

- Batch features
 - features computed from historical data, such as data in databases and data warehouses
 - for e.g. a restaurant's rating
- Streaming features
 - Features computed from streaming data — data in real-time transports
 - for e.g. estimation for delivery time
- In batch prediction, only batch features are used
- In online prediction, however, it's possible to use both batch features and streaming features.
 - For example, after a user puts in an order on DoorDash, they might need the following features to estimate the delivery time:
 - The mean preparation time of this restaurant in the past
 - In the last 10 minutes, how many other orders they have, and how many delivery people are available

Streaming prediction

A simplified architecture for online prediction that uses both streaming features and batch features

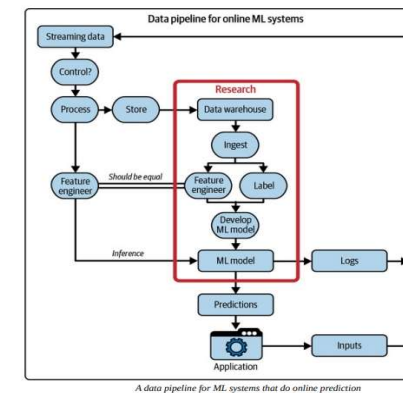


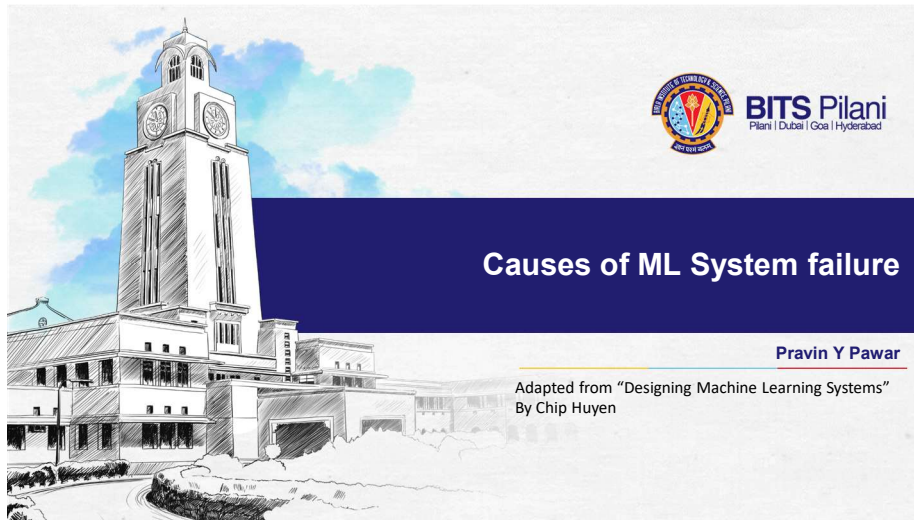
Online vs Batch prediction

Some key differences between batch prediction and online prediction

	Batch prediction (asynchronous)	Online prediction (synchronous)
Frequency	Periodical, such as every four hours	As soon as requests come
Useful for	Processing accumulated data when you don't need immediate results (such as recommender systems)	When predictions are needed as soon as a data sample is generated (such as fraud detection)
Optimized for	High throughput	Low latency

Unifying Batch Pipeline and Streaming Pipeline





ML system failure

- A failure happens when one or more expectations of the system is violated
- In traditional software, mostly care about a system's operational expectations:
 - whether the system executes its logic within the expected operational metrics,
 - e.g., latency and throughput
- For an ML system, care about both its operational metrics and its ML performance metrics
- For example, consider an English-French machine translation system
 - Operational expectation might be that, given an English sentence, the system returns a French translation within a one-second latency
 - ML performance expectation is that the returned translation is an accurate translation of the original English sentence 99% of the time

ML system failure types

- Operational expectation violations are easier to detect,
 - as usually accompanied by an operational breakage
 - such as a timeout, a 404 error on a webpage, an out-of-memory error, or a segmentation fault
- However, ML performance expectation violations are harder to detect
 - as doing so requires measuring and monitoring the performance of ML models in production
- In the English-French machine translation system, detecting whether the returned translations are correct 99% of the time is difficult
 - if don't know what the correct translations are supposed to be
- To effectively detect and fix ML system failures in production,
 - it's useful to understand why a model, after proving to work well during development, would fail in production
- Two types of failures: software system failures and ML-specific failures

Software System Failures

failures that would have happened to non-ML systems

- Dependency failure
 - A software package or a codebase that system depends on breaks, which leads system to break
 - common when the dependency is maintained by a third party
- Deployment failure
 - failures caused by deployment errors,
 - such as when accidentally deploy the binaries of an older version of model instead of the current version,
 - or when systems don't have the right permissions to read or write certain files
- Hardware failures
 - When the hardware that is used to deploy model, such as CPUs or GPUs, doesn't behave the way it should
 - For example, the CPUs used might overheat and break down
- Downtime or crashing
 - If a component of system runs from a server somewhere, such as AWS or a hosted service, and that server is down, system will also be down

Software System Failures(2)

- Addressing software system failures requires not ML skills, but traditional software engineering skills!
- Because of the importance of traditional software engineering skills in deploying ML systems,
 - ML engineering is mostly engineering, not ML!
- The reasons for the prevalence of software system failures:
 - ML adoption in the industry is still nascent,
 - tooling around ML production is limited
 - and best practices are not yet well developed or standardized
- However, as tooling's and best practices for ML production mature,
 - there are reasons to believe that the proportion of software system failures will decrease
 - and the proportion of ML-specific failures will increase

ML-Specific Failures

failures specific to ML systems

- Examples include
 - data collection and processing problems,
 - poor hyper parameters,
 - changes in the training pipeline not correctly replicated in the inference pipeline and vice versa,
 - data distribution shifts that cause a model's performance to deteriorate over time,
 - edge cases,
 - and degenerate feedback loops
- Even though they account for a small portion of failures, they can be more dangerous than non-ML failures
 - as they're hard to detect and fix, and they can prevent ML systems from being used altogether

Production data differing from training data

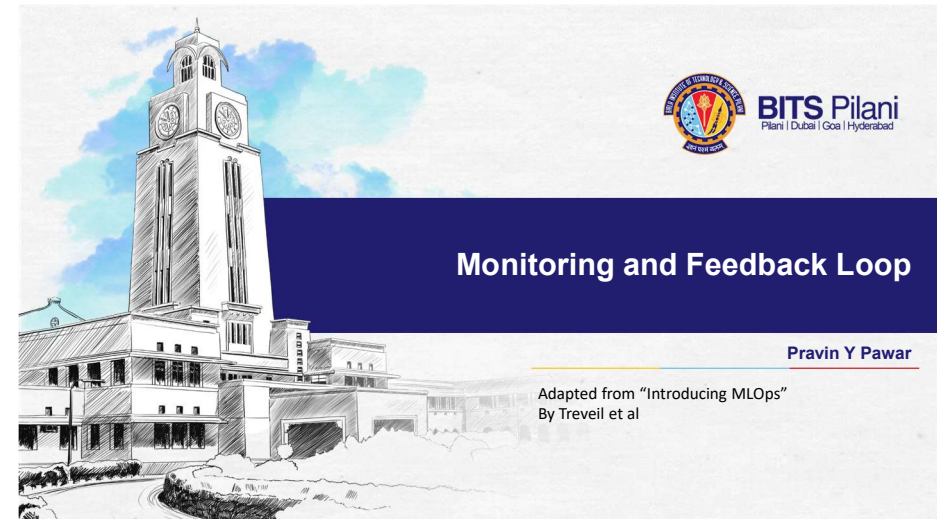
- ML model learns from the training data, means that the model learns the underlying distribution of the training data
 - with the goal of leveraging this learned distribution to generate accurate predictions for unseen data
 - when the model is able to generate accurate predictions for unseen data - model "generalizes to unseen data"
- The assumption
 - unseen data comes from a stationary distribution that is the same as the training data distribution
- This assumption is **incorrect** in most cases for two reasons!
- First, the **underlying distribution of the real-world data is unlikely to be the same as the underlying distribution of the training data**
 - Real-world data is multifaceted and, in many cases, virtually infinite,
 - whereas training data is finite and constrained by the time, compute, and human resources available during the dataset creation and processing
 - divergence leads to a common failure mode known as the train-serving skew: a model that does great in development but performs poorly when deployed
- Second, **the real world isn't stationary. Things change. Data distributions shift.**

Edge cases

- Edge cases are the data samples so extreme that they cause the model to make catastrophic mistakes
- An ML model that performs well on most cases but fails on a small number of cases
 - might not be usable if these failures cause catastrophic consequences
 - major self-driving car companies are focusing on making their systems work on edge cases
- Also true for any safety-critical application such as medical diagnosis, traffic control, e-discovery, etc.
- Can also be true for non-safety-critical applications
 - Imagine a customer service chatbot that gives reasonable responses to most of the requests,
 - but sometimes, it spits out outrageously racist or sexist content
 - will be a brand risk for any company that wants to use it, thus rendering it unusable

Degenerate feedback loops

- Feedback loop as the time it takes from when a prediction is shown until the time feedback on the prediction is provided
 - feedback can be used to
 - extract natural labels to evaluate the model's performance
 - train the next iteration of the model
- A degenerate feedback loop can happen when the predictions themselves influence the feedback,
 - which, in turn, influences the next iteration of the model
- A degenerate feedback loop is created when a system's outputs are used to generate the system's future inputs,
 - which, in turn, influence the system's future outputs
- Degenerate feedback loops are especially common in tasks with natural labels from users,
 - such as recommender systems and ads click-through-rate prediction
- Imagine building a system to recommend to users songs that they might like
 - songs that are ranked high by the system are shown first to users
 - because they are shown first, users click on them more,
 - which makes the system more confident that these recommendations are good



Model monitoring

Model monitoring is a crucial step in the ML model life cycle and a critical piece of MLOps

- When a machine learning model is deployed in production
 - it can start degrading in quality fast—and without warning—until it's too late
- Machine learning models need to be monitored at **two levels**:
- At the **resource level**, including ensuring the model is running correctly in the production environment.
 - Key questions include:
 - Is the system alive?
 - Are the CPU, RAM, network usage, and disk space as expected?
 - Are requests being processed at the expected rate?
- At the **performance level**, meaning monitoring the pertinence of the model over time
 - Key questions include:
 - Is the model still an accurate representation of the pattern of new incoming data?
 - Is it performing as well as it did during the design phase?

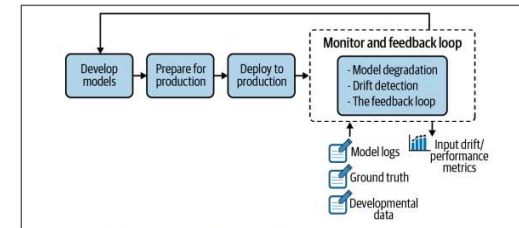
Model performance monitoring

- The first level is a traditional DevOps topic
- The latter is more complicated. Why?
 - Because how well a model performs is a reflection of the data used to train it
 - In particular, how representative that training data is of the live request data
- As the world is constantly changing,
 - a static model cannot catch up with new patterns that are emerging and evolving without a constant source of new data
- Model performance monitoring attempts to track this degradation,
 - At an appropriate time, it will also trigger the retraining of the model with more representative data

Model Retraining

- **Critical** for organizations to have a clear idea of **deployed models' drift and accuracy**
 - by setting up a **process** that allows for **easy monitoring and notifications**
- **Ideal** scenario would be a **pipeline** that **automatically triggers checks for degradation of model performance**
 - goal of notifications is not necessarily to kick off an automated process of retraining, validation, and deployment
 - but to alert the data scientist of the change; to **diagnose the issue** and evaluate the next course of action
- Critical that as part of MLOps and the ML model life cycle,
 - **data scientists** and their **managers** and the **organization** as a whole understand **model degradation**
- Every **deployed model** should come with **monitoring metrics and corresponding warning thresholds**
 - to detect meaningful business performance drops as quickly as possible

Monitoring and feedback loop



Monitoring and feedback loop highlighted in the larger context of the ML project life cycle

Model Degradation

- Once a machine learning model is trained and deployed in production
- Two approaches to **monitor its performance degradation**:
 - **ground truth evaluation**
 - **input drift detection**

Ground Truth Evaluation

Defined

- Ground truth retraining requires waiting for the label event
 - In a fraud detection model, the ground truth would be whether or not a specific transaction was actually fraudulent
 - For a recommendation engine, it would be whether or not the customer clicked on—or ultimately bought—one of the recommended products
- With the new ground truth collected,
 - next step is to compute the performance of model based on ground truth
 - compare it with registered metrics in the training phase
- **When the difference surpasses a threshold, the model can be deemed as outdated, and it should be retrained**
- The metrics to be monitored can be of two varieties:
 - Statistical metrics like accuracy, ROC AUC, log loss, etc.
 - domain agnostic
 - drawback is that the drop may be statistically significant without having any noticeable impact
 - cost of retraining and risk associated with a redeployment may be higher than expected benefits
 - Business metrics, like cost-benefit assessment such as the credit scoring
 - far more interesting because they ordinarily have a monetary value
 - enabling subject matter experts to better handle the cost-benefit trade-off of the retraining decision

Ground Truth Evaluation(2)

Challenges

- Ground truth is not always immediately, or even imminently, available
 - For some types of models, teams need to wait months (or longer) for ground truth labels to be available,
 - which can mean significant economic loss if the model is degrading quickly
 - Deploying a model for which the drift is faster than the lag is risky
- Ground truth and prediction are decoupled
 - To compute the performance of the deployed model on new data, it's necessary to be able to match ground truth with the corresponding observation
 - In many production environments, this is a challenging task because these two pieces of information are generated and stored in different systems and at different timestamps
- Ground truth is only partially available
 - In some situations, it is extremely expensive to retrieve the ground truth for all the observations,
 - which means choosing which samples to label and thus inadvertently introducing bias into the system

Input Drift

- Mathematically speaking, the samples of each dataset cannot be assumed to be drawn from the same distribution
 - i.e., they are not "identically distributed"
- Another mathematical property is necessary to ensure that ML algorithms perform as expected: independence
 - property is broken if samples are duplicated in the dataset or if it is possible to forecast the "next" sample given the previous one
- If train the algorithm on the first dataset and then deploy it on the second one
 - The **resulting distribution shift is called a drift**
- In wine quality prediction exercise,
- Called a **feature drift**
 - if the alcohol level is one of the features used by the ML model
 - or if the alcohol level is correlated with other features used by the model
- Called as a **concept drift** if it is not

Drift Detection in Practice

- To be able to react in a timely manner, model behavior should be monitored solely based on the feature values of the incoming data,
 - without waiting for the ground truth to be available
- The logic is that if the data distribution (e.g., mean, sd, correlations between features) diverges
 - between the training and testing phases on one side and the development phase on the other,
 - a **strong signal that the model's performance won't be the same!**
- Not the perfect mitigation measure, as retraining on the drifted dataset will not be an option,
 - but it can be part of mitigation measures (e.g., reverting to a simpler model, reweighting)

Causes of Data Drift

Two frequent root causes of data drift

- Sample selection bias
 - the training sample is not representative of the population
 - often stems from the data collection pipeline itself
 - For instance, building a model to assess the effectiveness of a discount program will be biased
 - if the best discounts are proposed for the best clients
- Non-stationary environment
 - where training data collected from the source population does not represent the target population
 - Often happens for time dependent tasks — such as forecasting with strong seasonality effects,
 - where learning a model over a given month won't generalize to another month

Input Drift Detection Techniques

Choice depends on the expected level of interpretability

- Two approaches
 - Univariate statistical tests
 - Domain classifier
- Should prefer univariate statistical tests
 - Organizations that need proven and explainable methods
- Domain classifier approach may be a good option
 - if complex drift involving several features simultaneously is expected
 - if the data scientists want to reuse what they already know and assuming the organization doesn't dread the black box effect

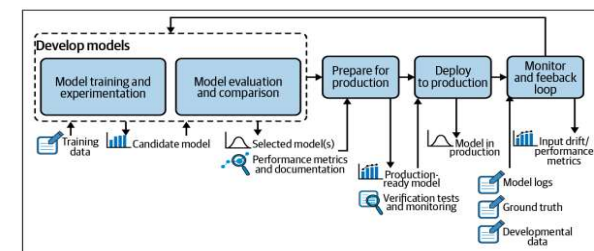
Univariate statistical tests

- Requires applying a statistical test on data from the source distribution and the target distribution for each feature
 - A warning will be raised when the results of those tests are significant
- Basic approaches rely on two tests:
 - For continuous features, the Kolmogorov-Smirnov test is a nonparametric hypothesis test that is used to check whether two samples come from the same distribution
 - For categorical features, the Chi-squared test is a practical choice that checks whether the observed frequencies for a categorical feature in the target data match the expected frequencies seen from the source data
- The main **advantage** of p-values is that they help **detect drift as quickly as possible**
- The main **drawback** is that they **do not quantify the level of the effect**
- If development datasets are very large, it is necessary to complement p-values with business-significant metrics
 - For example, on a sufficiently large dataset, the average age may have significantly drifted from a statistical perspective, but if the drift is only a few months, this is probably an insignificant value for many business use cases

Domain classifier

- Data scientists train a model that tries to discriminate between the original dataset (input features and, optionally, predicted target) and the development dataset
 - stack the two datasets and train a classifier that aims at predicting the data's origin
- The performance of the model (its accuracy, for example) can then be considered as a metric for the drift level
 - If this model is successful in its task, and thus has a high drift score,
 - implies that the data used at training time and the new data can be distinguished
 - fair to say that the new data has drifted
- To gain more insights, in particular to identify the features that are responsible for the drift
 - one can use the feature importance of the trained model

Continuous delivery for end-to-end machine learning process



Continuous delivery for end-to-end machine learning process

The Feedback Loop

- All effective machine learning projects implement a form of data feedback loop
 - information from the production environment flows back to the model prototyping environment for further improvement
- Continuous delivery for end-to-end machine learning process
 - Data collected in the monitoring and feedback loop is sent to the model development phase
 - From there, the system analyzes whether the model is working as expected
 - If it is, no action is required
 - If the model's performance is degrading, an update will be triggered, either automatically or manually by the data scientist
- In practice, usually means either retraining the model with new labeled data or developing a new model with additional features

The Feedback Loop(2)

- Goal of retraining is to be able to capture the emerging patterns and make sure that the business is not negatively impacted
- This infrastructure is comprised of three main components, which are critical to robust MLOps capabilities:
 - A logging system that collects data from several production servers
 - A model evaluation store that does versioning and evaluation between different model versions
 - An online system that does model comparison on production environments,
 - either with the shadow scoring (champion/challenger) setup or with A/B testing

Logging

- Monitoring a live system means collecting and aggregating data about its states
- Nowadays, as production infrastructures are getting more and more complex,
 - with several models deployed simultaneously across several servers,
 - an effective logging system is more important than ever!
- Data from these environments needs to be centralized to be analyzed and monitored,
 - either automatically or manually
 - will enable continuous improvement of the ML system
- An event log of a machine learning system is a record with a timestamp and information such as
 - Model metadata - Identification of the model and the version
 - Model inputs - Feature values of new observations
 - Model outputs - Predictions made by the model that
 - System action - an action taken based on model prediction
 - Model explanation - an explanation of prediction (i.e., which features have the most influence on the prediction)

Model Evaluation

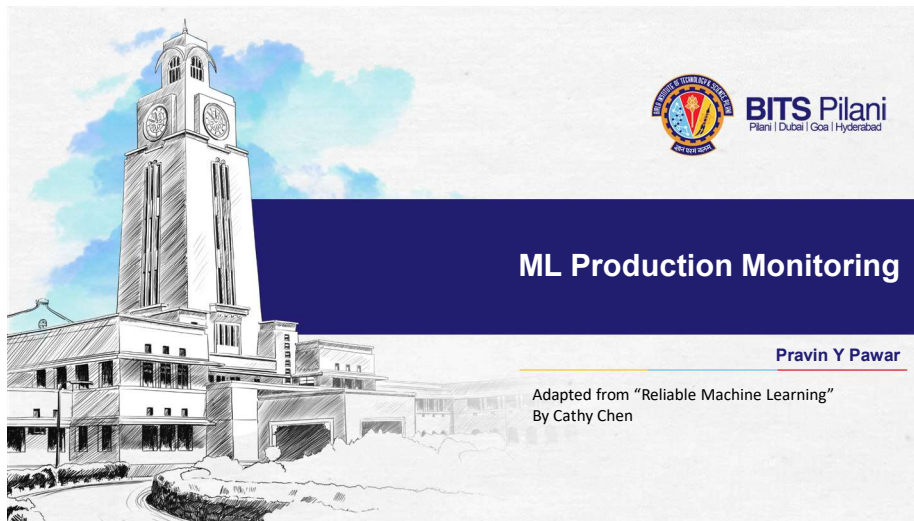
- If model performance is degrading, after review, data scientists decide to improve the model by retraining it,
 - With several trained candidate models, the next step is to compare them with the deployed model
 - means evaluating all the models (the candidates as well as the deployed model) on the same dataset
- If one of the candidate models outperforms the deployed model, there are two ways to proceed:
 - either update the model on the production environment
 - or move to an online evaluation via a champion/challenger or A/B testing setup
- In a nutshell, this is the notion of model store
- A structure that allows data scientists to:
 - Compare multiple, newly trained model versions against existing deployed versions
 - Compare completely new models against versions of other models on labeled data
 - Track model performance over time

Model evaluation store

- Formally, the model evaluation store serves as a structure that centralizes the data related to model life cycle to allow comparisons
- Two main tasks of a model evaluation store are:
 - Versioning the evolution of a logical model through time
 - Each logged version of the logical model must come with all the essential information concerning its training phase, including:
 - The list of features used
 - The preprocessing techniques that are applied to each feature
 - The algorithm used, along with the chosen hyperparameters
 - The training dataset
 - The test dataset used to evaluate the trained model (this is necessary for the version comparison phase)
 - Evaluation metrics
 - Comparing the performance between different versions of a logical model

Online Evaluation

- Online evaluation of models in production is critical from a business perspective,
 - but can be challenging from a technical perspective
- Two main modes of online evaluation:
 - Champion/challenger (otherwise known as shadow testing)
 - the candidate model shadows the deployed model and scores the same live requests
 - A/B testing
 - the candidate model scores a portion of the live requests and the deployed model scores the others
- Both cases require ground truth
 - evaluation will necessarily take longer than the lag between prediction and ground truth obtention
- Whenever shadow testing is possible, it should be used over A/B testing
 - because it is far simpler to understand and set up, and it detects differences more quickly



The basics

- Monitoring, at the most basic level, provides data about how your systems are performing
 - data is made storable, accessible, and displayable in some reasonable way
- Observability is an attribute of software, meaning that when correctly written, the emitted monitoring data
 - usually extended or expanded in some way, with labeling or tagging
 - can be used to correctly infer behavior of system
- Obviously, monitoring is hugely important in and of itself,
 - but an offshoot of monitoring is absolutely crucial: alerting
 - A useful simplification is that when things go wrong, humans are alerted to fix them
 - defining the conditions for "things going wrong," and being able to reliably notify the responsible folks that

What Does It Look Like?

- To do monitoring,
 - must have a monitoring system
 - as well as systems to be monitored (called the target systems)
- Today, target systems
 - emit metrics a series, typically of numbers, with an identifying name
 - which are then collected by the monitoring system
 - and transformed in various ways,
 - often via aggregation (producing a sum or a rate across multiple instances or machines)
 - or decoration (adding, say, event details onto the same data)
 - aggregated metrics are used for system analysis, debugging, and the alerting

Example

Web Server

- Web server emits a metric of the total number of requests it received
 - The monitoring system will obtain these metrics, usually via push or pull,
 - which refers to whether the metrics get pulled from the target systems or get pushed from them
 - These metrics are then collated, stored,
 - and perhaps processed in some way, generally as a time series
- Different monitoring systems will make different choices about how to receive, store, process,
 - but the data is generally queryable and often there's a graphical way to plot the monitoring data
 - to take advantage of our visual comparison hardware (eyes, retinas, optic nerves, and so on) to figure out what's actually happening

Problems with ML Production Monitoring

- ML model development is still in its infancy!
 - tools are immature, conceptual frameworks are underdeveloped
 - discipline is in short supply, as everyone scrambles to get some kind of model—any kind of model!
- The pressure to ship is real and has real effects!
- In particular, model development,
 - which is inherently hard because it involves reconciling a wide array of conflicting concerns
 - gets harder because that urgency forces developers and data science folks to focus on those hard problems
 - ignore the wider picture
- That wider picture often involves questions around monitoring and observability!

Difficulties of Development Versus Serving

The first problem is that effectively simulating production in development is extremely hard

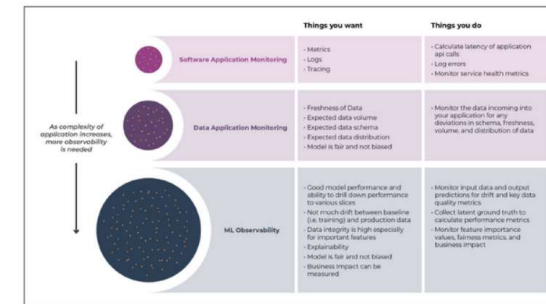
- Even if separate environments dedicated to that task (like test, staging, and so on.)
- Primarily Because of
- the wide variety of possible serving architectures
 - model pools, shared libraries, edge devices, etc., with the associated infrastructure running on)
- the invocation of the predictions
 - in development often invoke prediction methods directly or with a relatively small amount of code between developer and the model for velocity reasons
 - Running in production also generally means don't have the ability to manipulate input, logging level, processing, and so on
 - leading to huge difficulties in debugging, reproducing problematic configurations, etc.
- data in testing is not necessarily distributed like the data the model encounters in production
 - as always for ML, data distribution really matters

Difficulties of Development Versus Serving(2)

The second problem is about mature practices

- In conventional software delivery,
 - the industry has a good handle on work practices to improve throughput, reliability, and developmental velocity
 - most important is grouped concepts of continuous integration / continuous deployment (CI/CD), unit tests, small changes
- Unfortunately, today w/missing this equivalent of CI/CD for model development
 - not yet converged onto a good set of (telemetry-related, or otherwise) tools for model training and validation
- Expect this will improve over time as
 - existing tools (such as MLflow and Kubeflow) gain traction
 - vendors incorporate more of these concerns into their platforms
 - and the mindset of holistic, or whole-lifecycle monitoring gains more acceptance

Observability layers and system requirements

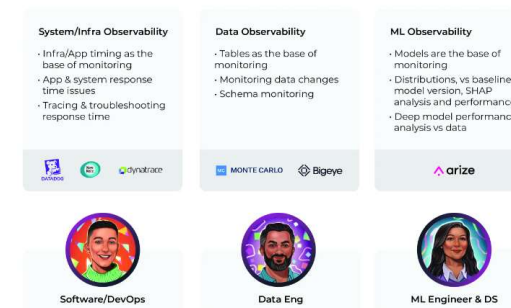


Observability layers and system requirements

Reasons for Continual ML Observability—in Production

- Observability data from models is absolutely fundamental to business
 - both tactical operations and strategic insights
- One example - connection between latency and online sales
 - In 2008, Amazon discovered that each additional 100 ms of latency lost 1% of sales, and also the converse
 - Similar results have been confirmed by Akamai Technologies, Google, Zalando, and others
- Without observability, there would be no way to have discovered this effect,
 - and certainly no way to know for sure that either making it better or worse!
- Ultimately, observability data is business outcome data
 - In the era of ML, this happily allows not just to detect and respond to outages,
 - but also to understand incredibly important things that are happening to business

ML observability in context

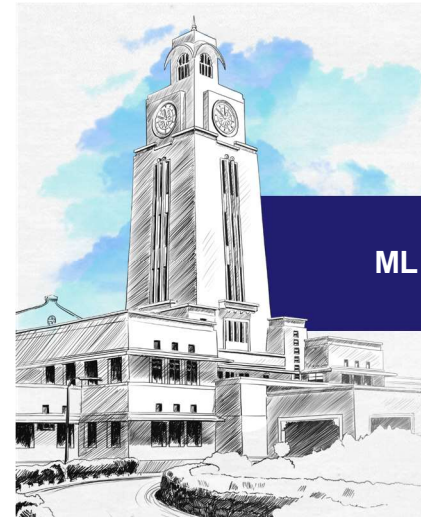



arize

Key Components of Observability

System vs Machine Learning

System Observability	ML Observability
Logs <ul style="list-style-type: none"> Records of an event that happened within an application. Typically not mutable by an event ID. Searchable by tags and unstructured indexes. 	Inference Store <ul style="list-style-type: none"> Records of ML prediction events that are logged from the model. Raw prediction events that hold granular context about the model's predictions. Mutable by prediction ID and dataset.
Metrics <ul style="list-style-type: none"> Measured values of system performance. Metrics comprise a set of attributes (i.e. value, label, and timestamp) that convey information about SLAs, SLOs, and SLIs. 	Model Metrics <ul style="list-style-type: none"> Calculated metrics on the prediction events. Provides ways to determine model health over time - this includes drift, performance, and data quality metrics. Metrics can be monitored. Metrics can be aggregate or slice-level.
Tracing <ul style="list-style-type: none"> Provides context for the other components of observability (logs, metrics). Follows the entire lifecycle of a request or action across distributed systems. 	ML Performance Tracing <ul style="list-style-type: none"> ML performance tracing is the methodology for pinpointing the source of a model performance problem. Involves mapping back to the data that caused the problem. Necessarily a distinct discipline because logs and metrics are rarely helpful for debugging model performance.




BITS Pilani
 Pilani | Dubai | Goa | Hyderabad

ML Monitoring - Metrics and Toolbox

Pravin Y Pawar

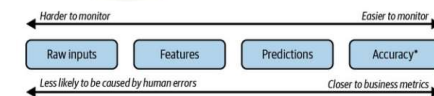
Adapted from "Designing Machine Learning Systems"
By Chip Huyen

ML Monitoring

- As the industry realized that many things can go wrong with an ML system,
 - many companies started investing in monitoring and observability for their ML systems in production
- Monitoring refers to the act of tracking, measuring, and logging different metrics that can help us determine when something goes wrong
- Observability means setting up our system in a way that gives us visibility into system to help us investigate what went wrong
- Monitoring is all about metrics!
 - Because ML systems are software systems, the first class of metrics need to monitor are the operational metrics
 - designed to convey the health of systems
- Generally divided into three levels:
 - the network the system is run on,
 - the machine the system is run on,
 - and the application that the system runs
- Examples
 - latency; throughput; the number of prediction requests model receives in the last minute, hour, day;
 - the percentage of requests that return with a 2xx code;
 - CPU/GPU utilization; memory utilization; etc.

ML-Specific Metrics

- Within ML-specific metrics, there are generally four artifacts to monitor:
 - a model's accuracy-related metrics,
 - predictions,
 - features,
 - and raw inputs
- Artifacts generated at four different stages of an ML system pipeline
 - The deeper into the pipeline an artifact is, the more transformations it has gone through,
 - which makes a change in that artifact more likely to be caused by errors in one of those transformations
 - However, the more transformations an artifact has gone through, the more structured it's become
 - closer it is to the metrics actually care about, which makes it easier to monitor



Monitoring accuracy-related metrics

- Accuracy-related metrics are the most direct metrics to help you decide whether a model's performance has degraded
- If system receives any type of user feedback for the predictions it makes
 - click, hide, purchase, upvote, downvote, favorite, bookmark, share, etc.
 - should definitely log and track it
- Some feedback can be used to infer natural labels,
 - which can then be used to calculate model's accuracy-related metrics
- Feedback can be used to detect changes in ML model's performance
 - For example, when building a system to recommend to users what videos to watch next on YouTube,
 - want to track not only whether the users click on a recommended video (click-through rate),
 - but also the duration of time users spend on that video and whether they complete watching it (completion rate)
 - If, over time, the clickthrough rate remains the same but the completion rate drops, it might mean that recommender system is getting worse
- Possible to engineer system to collect users' feedback
 - For example, Google Translate has the option for users to upvote or downvote a translation

Monitoring predictions

- Prediction is the most common artifact to monitor
 - If it's a regression task, each prediction is a continuous value (e.g., the predicted price of a house)
 - If it's a classification task, each prediction is a discrete value corresponding to the predicted category
- Each prediction is usually just a number (low dimension), predictions are easy to visualize
 - summary statistics are straightforward to compute and interpret
- Can monitor predictions for distribution shifts as they are low dimensional
 - Easier to compute two-sample tests to detect whether the prediction distribution has shifted
 - Prediction distribution shifts are also a proxy for input distribution shifts
- Can also monitor predictions for anything odd happening
 - such as predicting an unusual number of False in a row

Monitoring features

- ML monitoring solutions in the industry focus on tracking changes in features, both
- the features that a model uses as inputs
 - the intermediate transformations from raw inputs into final features
- Feature monitoring is appealing because
 - compared to raw input data, features are well structured following a predefined schema
- The first step of feature monitoring is feature validation:
 - ensuring that features follow an expected schema
- Things can be checked for a given feature:
 - If the min, max, or median values of a feature are within an acceptable range
 - If the values of a feature satisfy a regular expression format
 - If all the values of a feature belong to a predefined set
 - If the values of a feature are always greater than the values of another feature
- Many open source libraries that help in basic feature validation,
 - Two most common are Great Expectations and Deequ, which is by AWS

Monitoring features(2)

Four major concerns when doing feature monitoring

- A company might have hundreds of models in production, and each model uses hundreds, if not thousands, of features.
 - Even something as simple as computing summary statistics for all these features every hour can be expensive,
 - not only in terms of compute required but also memory used
 - Tracking, i.e., constantly computing, too many metrics can also slow down system
 - increase both the latency that users experience
- While tracking features is useful for debugging purposes, it's not very useful for detecting model performance degradation
 - In practice, an individual feature's minor changes might not harm the model's performance at all
 - Feature distributions shift all the time, and most of these changes are benign
 - If want to be alerted whenever a feature seems to have drifted, might soon be overwhelmed by alerts
 - realize that most of these alerts are false positives - "alert fatigue"
- Feature extraction is often done in multiple steps (such as filling missing values and standardization),
 - using multiple libraries (such as pandas, Spark),
 - on multiple services (such as BigQuery or Snowflake).
- Even if its detected a harmful change in a feature, it might be impossible to detect whether this change is
 - caused by a change in the underlying input distribution or whether it's caused by an error in one of the multiple processing steps
- The schema that features follow can change over time.
 - If don't have a way to version schemas and map each of features to its expected schema,
 - the cause of the reported alert might be due to the mismatched schema rather than a change in the data

Monitoring raw inputs

- A change in the features might be caused by problems in processing steps and not by changes in data
 - can monitor the raw inputs before they are processed
 - not be easier to monitor, as it can come from multiple sources in different formats, following multiple structures
- The way many ML workflows are set up today also makes it impossible for ML engineers to get direct access to raw input data,
 - as the raw input data is often managed by a data platform team who processes and moves the data to a location like a data warehouse,
- ML engineers can only query for data from that data warehouse where the data is already partially processed
- Monitoring raw inputs is often a responsibility of the data platform team, not the data science or ML team

Monitoring Toolbox

- Measuring, tracking, and interpreting metrics for complex systems is a nontrivial task
 - engineers rely on a set of tools to help them do so
- Common for the industry to herald **metrics, logs, and traces** as the three pillars of monitoring
- Seem to be generated from the perspective of people who develop monitoring systems:
 - traces are a form of logs and metrics can be computed from logs
- Focus on the set of tools from the perspective of users of the monitoring systems:
 - logs,
 - dashboards,
 - alerts

Monitoring Toolbox(2)

Logs

- Traditional software systems rely on logs to record events produced at runtime
 - An event is anything that can be of interest to the system developers,
 - either at the time the event happens or later for debugging and analysis purposes
- Examples of events
 - a container starts,
 - the amount of memory it takes,
 - when a function is called,
 - when that function finishes running,
 - the other functions that this function calls,
 - the input and output of that function,
 - log crashes, stack traces, error codes, and more.
- The number of logs can grow very large very quickly.
 - need to query your logs for the sequence of events that caused it, a process that can feel like searching for a needle in a haystack
- Because logs have grown so large and so difficult to manage,
 - there have been many tools developed to help companies manage and analyze logs
 - The log management market is estimated to be worth USD 2.3 billion in 2021, and it's expected to grow to USD 4.1 billion by 2026

Monitoring Toolbox(3)

Dashboards

- A picture is worth a thousand words!
 - A series of numbers might mean nothing, but visualizing them on a graph might reveal the relationships among these numbers
 - Dashboards to visualize metrics are critical for monitoring
- Another use of dashboards is to make monitoring accessible to non-engineers.
 - Monitoring isn't just for the developers of a system, but also for non-engineering stakeholders
 - including product managers and business developers
- Dashboard rot
 - Excessive metrics on a dashboard can also be counterproductive
 - important to pick the right metrics
 - abstract out lower-level metrics to compute higher-level signals that make better sense for specific tasks

Monitoring Toolbox(4)

Alerts

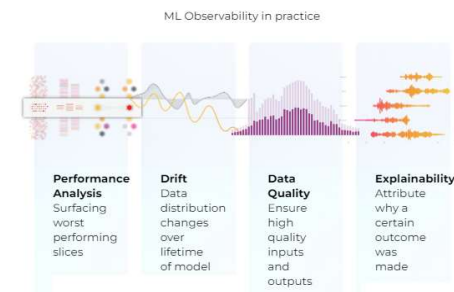
- When monitoring system detects something suspicious, it's necessary to alert the right people about it
- An alert consists of the following three components:
 - An alert policy
 - describes the condition for an alert
 - might want to create an alert when a metric breaches a threshold, optionally over a certain duration
 - Notification channels
 - describe who is to be notified when the condition is met
 - A description of the alert
 - helps the alerted person understand what's going on with a detailed description
 - often necessary to make the alert actionable
 - by providing mitigation instructions or a runbook, a compilation of routine procedures and operations that might help with handling the alert
- Alert fatigue is a real phenomenon!
 - can be demoralizing—nobody likes to be awakened in the middle of the night for something outside of their responsibilities
 - can be dangerous —being exposed to trivial alerts can desensitize people to critical alerts
 - important to set meaningful conditions so that only critical alerts are sent out



What is ML Observability?

- ML Observability is a tool used to monitor, troubleshoot, and explain machine learning models
 - as they move from research to production environments
- An effective observability tool should not only automatically surface issues,
 - but drill down to the root cause of your ML problems and act as a guardrail for models in production

4 Pillars of ML Observability



Performance Analysis

- ML observability enables fast actionable performance information on models deployed in production
- While performance analysis techniques vary on a case-by-case basis depending on model type and its use case in the real world,
 - common metrics include: Accuracy, Recall, F-1, MAE, RMSE, and Precision
- Performance analysis in an ML observability system ensures that performance has not degraded drastically from when it was trained or when it was initially promoted to production.

Drift

- ML observability encompasses drift to monitor for a change in distribution over time,
 - measured for model inputs, outputs, and actuals of a model
- Measure drift to identify if models have grown stale, have data quality issues, or if there are adversarial inputs in model
- Detecting drift in models will help protect models from performance degradation and allow to better understand how to begin resolution

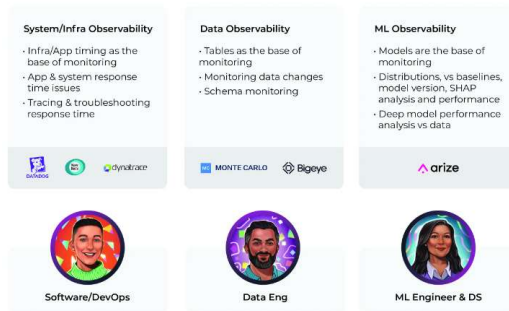
Data Quality

- Data quality checks in an ML observability system identify hard failures
 - within data pipelines between training and production
 - that can negatively impact a model's end performance
- Data quality includes
 - monitoring for cardinality shifts,
 - missing data,
 - data type mismatch,
 - out-of-range,
 - and more to better gauge model performance issues and ease RCA

Explainability

- Explainability in ML observability uncovers feature importance
 - across training, validation, and production environments
 - which provides the ability to introspect and understand why a model made a particular prediction
- Explainability is commonly achieved by
 - calculating metrics such as SHAP and LIME
 - to build confidence and continuously improve machine-learned models

ML observability in context



[arize](https://arize.com)

How Can I Achieve ML Observability?

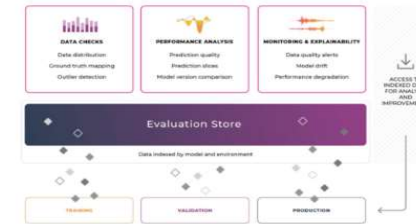
ML Observability with an Evaluation Store

Monitor drift, data quality issues, or anomalous performance degradations using baselines.

Analyze performance metrics in aggregate (or slice) for any model, in any environment — production, validation, training.

Root Cause Analysis to connect changes in performance to why they occurred.

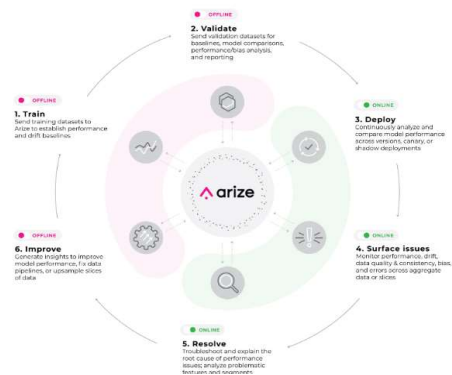
Enable feedback loop to **actively improve** model performance.



[arize](https://arize.com)

ML Observability with Arize

- ML Observability is the practice of obtaining a deep understanding into model's data and performance across its lifecycle
 - Observability doesn't just stop at surfacing a red or green light,
 - but enables ML practitioners to root cause/explain why a model is behaving a certain way in order to improve it



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Continual Learning

Pravin Y Pawar

Adapted from "Designing Machine Learning Systems"
By Chip Huyen

Continual Learning

- How do adapt models to data distribution shifts?
 - The answer is by continually updating our ML models
- Continual learning is largely an infrastructural problem
 - Setting up infrastructure allows to update models as frequently as required
- The goal of continual learning is to safely and efficiently automate the update
 - allows to design an ML system that is maintainable and adaptable to changing environments

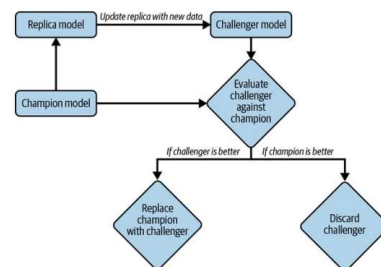
What is continual learning?

- Many people think of the training paradigm where a model updates itself with every incoming sample in production
 - people imagine updating models very frequently, such as every 5 or 10 minutes
- Very few companies actually do that!
 - makes model susceptible to catastrophic forgetting - the tendency of a neural network to completely and abruptly forget previously learned information upon learning new information
 - make training more expensive - most hardware backends today were designed for batch processing, so processing only one sample at a time causes a huge waste of compute power and is unable to exploit data parallelism
- Most companies don't need to update their models that frequently because of two reasons
 - First, they don't have enough traffic (i.e., enough new data) for that retraining schedule to make sense
 - Second, their models don't decay that fast
 - If changing retraining schedule from a week to a day gives no return and causes more overhead, there's no need to do it.

A simplification of how continual learning work in production

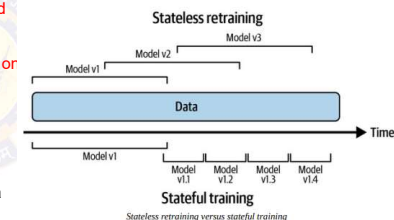
Champion-Challenger

- The updated model shouldn't be deployed until it's been evaluated
 - means that shouldn't make changes to the existing model directly
- Instead,
 - should create a replica of the existing model
 - update this replica on new data
 - only replace the existing model with the updated replica if the updated replica proves to be better
- The existing model - champion model
- The updated replica - the challenger
- An oversimplification of the process for the sake of understanding!
 - In reality, a company might have multiple challengers at the same time



Stateless Retraining Versus Stateful Training

- Continual learning isn't about the retraining frequency,
 - but the manner in which the model is retrained
- Most companies do **stateless retraining** — the model is trained from scratch each time
- Some follow **stateful training** — the model continues training on new data
 - also known as fine-tuning or incremental learning
- Stateful training allows to update model with less data
- Training a model from scratch tends to require a lot more data
- One beautiful property that is often overlooked is that with stateful training,
 - it might be possible to avoid storing data altogether



Stateless Retraining Versus Stateful Training(2)

- The companies that have most successfully used stateful training
 - also occasionally train their model from scratch on a large amount of data to calibrate it
- Once infrastructure is set up to allow both stateless retraining and stateful training,
 - the training frequency is just a knob to twist
 - can update models once an hour, once a day, or whenever a distribution shift is detected
- Continual learning is about **setting up infrastructure** in a way that
 - allows a data scientist or ML engineer,
 - to **update models whenever it is needed**,
 - whether **from scratch or fine-tuning**,
 - to **deploy** this update **quickly**

Model iteration vs Data iteration

- Stateful training sounds cool, but **how does this work if needs to add a new feature or another layer to model?**
- Must differentiate **two types of model updates**:
 - **Model iteration** - A new feature is added to an existing model or the model architecture is changed
 - **Data iteration** - Model architecture and features remain the same, but **refresh this model with new data**.
- As of today, stateful training is mostly applied for data iteration!
- Changing model architecture or adding a new feature still requires training the resulting model from scratch
 - research shows that it might be possible to bypass training from scratch for model iteration
 - by using techniques such as knowledge transfer (Google, 2015) and model surgery (OpenAI, 2019)

Why Continual Learning?

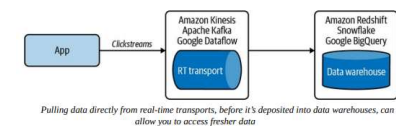
Why would you need the ability to update your models as fast as you want?

- The first use case of continual learning is to combat data distribution shifts,
 - especially when the shifts happen suddenly
- Another use case of continual learning is to adapt to rare events
- Can help overcome is the continuous cold start problem
- "Why continual learning?" should be rephrased as "why not continual learning?"
 - Continual learning is a **superset of batch learning** - allows to do everything traditional batch learning can do
 - If continual learning **takes the same effort to set up and costs the same to do as batch learning**,
 - there's no reason not to do continual learning!
- **MLOps tooling for continual learning is maturing**, which means, one day not too far in the future,
 - it might be as easy to set up continual learning as batch learning

Continual Learning Challenges

Fresh data access challenge - Data Sources

- **If want to update model every hour, need new data every hour!**
- Data Sources
 - Currently, many companies pull new training data from their data warehouses
 - **The speed at which data can be pulled from data warehouses depends on the speed at which this data is deposited into data warehouses**
 - The speed can be slow, especially if data comes from multiple sources
 - An alternative is to allow pull data before it's deposited into data warehouses,
 - e.g., **directly from real-time transports** such as Kafka and Kinesis that transport data from applications to data warehouses



Continual Learning Challenges(2)

Fresh data access challenge - Labelling

- If model needs labeled data to update, data will need to be labeled as well!
 - In many applications, the speed at which a model can be updated is bottlenecked by the speed at which data is labeled
- The best candidates for continual learning are tasks where can get natural labels with short feedback loops
 - dynamic pricing, estimating time of arrival, stock price prediction,
 - ads click-through prediction, and recommender systems
- If model's speed iteration is bottlenecked by labeling speed,
 - possible to speed up the labeling process by leveraging programmatic labeling tools like Snorkel to generate fast labels
 - possible to leverage crowdsourced labels to quickly annotate fresh data
- Given that tooling around streaming is still nascent,
 - architecting an efficient streaming-first infrastructure for accessing fresh data and extracting fast labels from real-time transports can be engineering-intensive and costly

Continual Learning Challenges(3)

Evaluation challenge

- ML systems make catastrophic failures in production,
 - from millions of minorities being unjustly denied loans,
 - to drivers who trust autopilot too much being involved in fatal crashes
- The biggest challenge is in making sure that this update is good enough to be deployed!
- The risks for catastrophic failures amplify with continual learning
 - More frequently models are updated, the more opportunities there are for updates to fail
 - Makes models more susceptible to coordinated manipulation and adversarial attack
- To avoid similar or worse incidents, it's crucial to thoroughly test each of model updates to ensure its performance and safety before deploying the updates to a wider audience.
- When designing the evaluation pipeline for continual learning, keep in mind that evaluation takes time,
 - which can be another bottleneck for model update frequency.

Continual Learning Challenges(4)

Algorithm challenge - "softer" challenge

- Challenge as it only affects certain algorithms and certain training frequencies.
 - only affects matrix-based and tree-based models that want to be updated very fast (e.g., hourly)
- Consider two different models:
 - a neural network and a matrix-based model, such as a collaborative filtering model
 - The collaborative filtering model uses a user-item matrix and a dimension reduction technique
- A neural network model
 - Can update model with a data batch of any size
 - Can even perform the update step with just one data sample
- Collaborative filtering model
 - For updating model, first need to use the entire dataset to build the user-item matrix before performing dimensionality reduction on it
 - if matrix is large, the dimensionality reduction step would be too slow and expensive to perform frequently
 - less suitable for learning with a partial dataset than the preceding neural network model

Four Stages of Continual Learning

- How to overcome these challenges and make continual learning happen?
- The move toward continual learning happens in four stages
 - Stage 1: Manual, stateless retraining
 - Stage 2: Automated retraining
 - Stage 3: Automated, stateful training
 - Stage 4: Continual learning

Four Stages of Continual Learning(2)

Stage 1: Manual, stateless retraining

- In the beginning, the ML team often focuses on developing ML models to solve as many business problems as possible
 - Because team is focusing on developing new models, updating existing models takes a backseat
 - No fixed frequency to update the models
- Update an existing model only when the following two conditions are met:
 - the model's performance has degraded to the point that it's doing more harm than good
 - team has time to update it
- The process of updating a model is manual and ad hoc
 - Someone, usually a data engineer, has to query the data warehouse for new data.
 - Someone else cleans this new data, extracts features from it, retrains that model from scratch on both the old and new data, and then exports the updated model into a binary format.
 - Someone else takes that binary format and deploys the updated model.
- A vast majority of companies outside the tech industry—
 - e.g., any company that adopted ML less than three years ago and doesn't have an ML platform team—are in this stage

Four Stages of Continual Learning(3)

Stage 2: Automated retraining

- After a few years, team has managed to deploy models to solve most of the obvious problems
 - Priority is no longer to develop new models, but to maintain and improve existing models
 - The ad hoc, manual process of updating models mentioned from the previous stage has grown into a pain point too big to be ignored
- Team decides to write a script to automatically execute all the retraining steps
 - run periodically using a batch process such as Spark
- Status
 - Most companies with somewhat mature ML infrastructure are in this stage
 - Some sophisticated companies run experiments to determine the optimal retraining frequency
 - For most companies in this stage, the retraining frequency is set based on gut feeling
 - e.g., "once a day seems about right" or "let's kick off the retraining process each night when we have idle compute"

Four Stages of Continual Learning(4)

Stage 2: Automated retraining - Requirements for setup

- If company has ML models in production,
 - likely that company already has most of the infrastructure pieces needed for automated retraining
- The feasibility of this stage revolves around the feasibility of writing a script to automate workflow and configure infrastructure to automatically:
 1. Pull data.
 2. Downsample or upsample this data if necessary.
 3. Extract features.
 4. Process and/or annotate labels to create training data.
 5. Kick off the training process.
 6. Evaluate the newly trained model.
 7. Deploy it.
- In general, the three major factors that will affect the feasibility of this script are:
 - scheduler,
 - data,
 - and model store

Four Stages of Continual Learning(4)

Stage 3: Automated, stateful training

- Need to reconfigure automatic updating script so that, when the model update is kicked off
 - it first locates the previous checkpoint and loads it into memory before continuing training on this checkpoint
- Requirements
 - The main thing needed at this stage is a way to track data and model lineage
- Imagine you first upload model version 1.0
 - is updated with new data to create model version 1.1, and so on to create model 1.2
 - Another model is uploaded and called model version 2.0
 - is updated with new data to create model version 2.1
 - After a while, you might have model version 3.32, model version 2.11, model version 1.64
- Might want to know
 - how these models evolve over time,
 - which model was used as its base model,
 - which data was used to update it so that you can reproduce and debug it
- Need model store that has this model lineage capacity

Four Stages of Continual Learning(5)

Stage 4: Continual learning

- At stage 3, models are still updated based on a fixed schedule set out by developers
 - Finding the optimal schedule isn't straightforward and can be situation-dependent
 - might want models to be automatically updated whenever data distributions shift and the model's performance plummets
 - The move from stage 3 to stage 4 is steep!
- Requirements
 - First need a mechanism to trigger model updates
 - Time-based - every five minutes
 - Performance-based - whenever model performance plummets
 - Volume-based - whenever the total amount of labeled data increases by 5%
 - Drift-based - whenever a major data distribution shift is detected
 - For this trigger mechanism to work, need a solid monitoring solution
 - Hard part is not to detect the changes, but to determine which of these changes matter
 - If monitoring solution gives a lot of false alerts, model will end up being updated much more frequently than it needs to be
 - Need a solid pipeline to continually evaluate model updates.
 - Writing a function to update models isn't much different from stage 3
 - Hard part is to ensure that the updated model is working properly

How Often to Update Your Models?

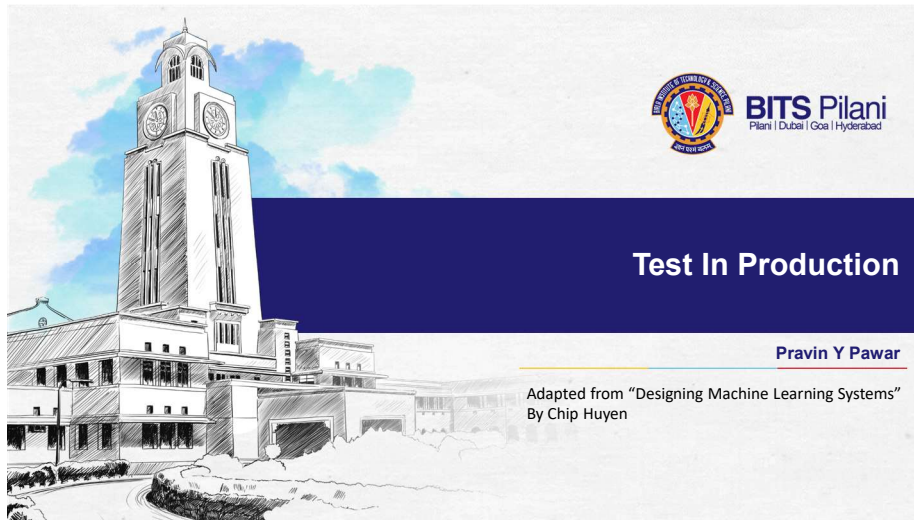
- Depends on how much gain model will get from being updated with fresh data
 - The more gain model can get from fresher data, the more frequently it should be retrained
- The question on how often to update your model is a difficult one to answer!
 - In the beginning, when infrastructure is nascent and the process of updating a model is manual and slow, the answer is: as often as you can.
 - As infrastructure matures and the process of updating a model is partially automated and can be done in a matter of hours, if not minutes,
 - the answer to this question is contingent on the answer to the following question:
 - "How much performance gain would I get from fresher data?"
- Points to be considered
 - Value of data freshness
 - Model iteration versus data iteration

Value of data freshness

- The question of how often to update a model becomes a lot easier
 - if knows how much the model performance will improve with updating
- For example,
 - If switch from retraining model every month to every week, how much performance gain can we get?
 - What if we switch to daily retraining?
- People keep saying that data distributions shift, so fresher data is better,
 - but how much better is fresher data?
 - One way to figure out the gain is by training model on the data from different time windows in the past and evaluating it on the data from today to see how the performance changes.

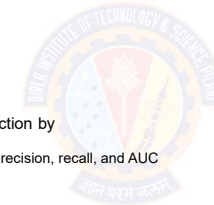
Model iteration versus data iteration

- Not all model updates are the same - can be model iteration or data iteration!
- Might wonder not only how often to update model, but also what kind of model updates to perform
 - In theory, can do both types of updates
 - In practice, should do both from time to time
- The more resources spend in one approach, the fewer resources can be spent in another
 - if iterating on data doesn't give much performance gain, then should spend resources on finding a better model
 - if finding a better model architecture requires 100X compute for training and gives 1% performance
 - whereas updating the same model on data from the last three hours requires only 1X compute and also gives 1% performance gain, will be better off iterating on data
- Currently no book can give the answer on which approach will work better for specific model on specific task
 - have to do experiments to find out.



Offline Evaluation

- An offline model evaluation happens when the model is being trained by the analyst
- The analyst tries out different
 - features,
 - models,
 - algorithms,
 - and hyperparameters
- Model training is guided in the right direction by
 - Tools like confusion matrix
 - Various performance metrics, such as precision, recall, and AUC
- Process
 - First, validation data is used to assess the chosen performance metric and compare models
 - Once the best model is identified, the test set is used, also in offline mode, to again assess the best model's performance
 - This final offline assessment guarantees post-deployment model performance



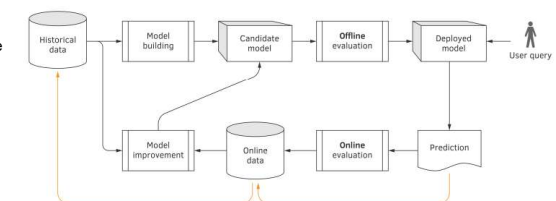
Offline Evaluation(2)

Two major test types for offline evaluation

- Test splits
 - are usually static and have to be static so that you have a trusted benchmark to compare multiple models
 - will be hard to compare the test results of two models if they are tested on different test sets
- Backtests
 - method of testing a predictive model on data from a specific period of time in the past
 - If model is updated to adapt to a new data distribution, it's not sufficient to evaluate this new model on test splits from the old distribution
 - one idea is to test model on the most recent data that you have access to - after updated model on the data from the last day, might want to test this model on the data from the last hour
- The question is whether backtests are sufficient to replace static test splits. Not quite!
 - If something went wrong with data pipeline and some data from the last hour is corrupted, evaluating model solely on this recent data isn't sufficient.
 - With backtests, should still evaluate model on a static test set that have been extensively studied and (mostly) trust as a form of sanity check.

Offline and Online Evaluation

- Historical data is first used to train a deployment candidate
 - Then it is evaluated offline
 - If the result is satisfactory, deployment candidate becomes the deployed model, and starts accepting user queries
- User queries and the model predictions are used for an online evaluation of the model
 - The online data is then used to improve the model
 - To close the loop, the online data is permanently copied to the offline data repository



The placement of offline and online model evaluations in a machine learning system.

Source: Machine Learning Engineering by Burkov

Why do we evaluate both offline and online?

- The offline model evaluation reflects how well the analyst succeeded
 - in finding the right features, learning algorithm, model, and values of hyperparameters
 - reflects how good the model is from an engineering standpoint
- Online evaluation, focuses on measuring business outcomes,
 - such as customer satisfaction, average online time, open rate, and click-through rate
 - may not be reflected in historical data, but it's what the business really cares about
- Offline evaluation doesn't allow us to test the model in some conditions that can be observed online,
 - such as connection and data loss, and call delays

Test in production

Aka Online Evaluation

- The only way to know whether a model will do well in production is to deploy it
 - led to one seemingly terrifying but necessary concept
- Techniques to help to evaluate models in production (mostly) safely
 - Shadow deployment
 - A/B testing
 - Canary analysis
 - Interleaving experiments
 - Bandits
- To sufficiently evaluate models, first need a mixture of offline evaluation and online evaluation!

Shadow Deployment

Might be the safest way to deploy model or any software update

- Works as follows
 1. Deploy the candidate model in parallel with the existing model
 2. For each incoming request, route it to both models to make predictions, but only serve the existing model's prediction to the user
 3. Log the predictions from the new model for analysis purposes.
- Only when found that the new model's predictions are satisfactory do replace the existing model with the new model!
- The risk of this new model doing something funky is low
 - as don't serve the new model's predictions to users until made sure that the model's predictions are satisfactory
- Always not favorable because it's expensive
 - doubles the number of predictions system has to generate, which generally means doubling inference compute cost

A/B Testing

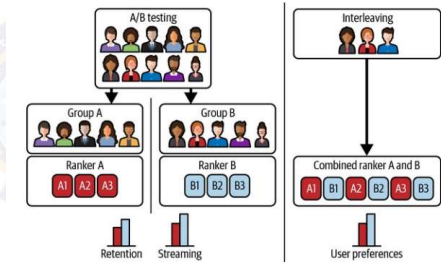
- A way to compare two variants of an object, typically by testing responses to these two variants, determining which of the two variants is more effective
 - the existing model as one variant, and the candidate model (the recently updated model) as another variant
- Works as follows:
 1. Deploy the candidate model alongside the existing model
 2. A percentage of traffic is routed to the new model for predictions; the rest is routed to the existing model for predictions.
 3. Monitor and analyze the predictions and user feedback, if any, from both models to determine whether the difference in the two models' performance is statistically significant.
- Two important things:
 - First, A/B testing consists of a randomized experiment: the traffic routed to each model has to be truly random.
 - If not, the test result will be invalid
 - Second, A/B test should be run on a sufficient number of samples to gain enough confidence about the outcome.
 - How to calculate the number of samples needed for an A/B test is a simple question with a very complicated answer.
- Often, in production, you don't have just one candidate but multiple candidate models
 - possible to do A/B testing with more than two variants, which means can have A/B/C testing or even A/B/C/D testing

Canary Release

- A technique to reduce the risk of introducing a new software version in production by slowly rolling out the change to a small subset of users
 - before rolling it out to the entire infrastructure and making it available to everybody
- Works as follows:
 1. Deploy the candidate model alongside the existing model. The candidate model is called the canary
 2. A portion of the traffic is routed to the candidate model.
 3. If its performance is satisfactory, increase the traffic to the candidate model. If not, abort the canary and route all the traffic back to the existing model.
 1. Stop when either the canary serves all the traffic (the candidate model has replaced the existing model) or when the canary is aborted
- The candidate model's performance is measured against the existing model's performance according to the metrics you care about
 - If the candidate model's key metrics degrade significantly, the canary is aborted and all the traffic will be routed to the existing model
- Canary releases can be used to implement A/B testing due to the similarities in their setups
 - don't have to randomize the traffic to route to each model
 - first roll out the candidate model to a less critical market before rolling out to everybody

Interleaving Experiments

- Imagine two recommender systems, A and B, and want to evaluate which one is better
 - Each time, a model recommends 10 items users might like
- With A/B testing, divide users into two groups: one group is exposed to A and the other group is exposed to B.
 - Each user will be exposed to the recommendations made by one model
- What if instead of exposing a user to recommendations from a model, we expose that user to recommendations from both models and see which model's recommendations they will click on?



An illustration of interleaving versus A/B testing. Source: Adapted from an image by Parks.

Interleaving Experiments(2)

- In A/B testing, core metrics like retention and streaming are measured and compared between two groups
- In interleaving, two algorithms can be compared by measuring user preferences,
 - no guarantee that user preference will lead to better core metrics
- Netflix found that interleaving "reliably identifies the best algorithms with considerably smaller sample size compared to traditional A/B testing."
- When recommendations from multiple models are shown to users,
 - the position of a recommendation influences how likely a user will click on it
 - users are much more likely to click on the top recommendation than the bottom recommendation
- For interleaving to yield valid results, must ensure that at any given position,
 - a recommendation is equally likely to be generated by A or B.

Bandits

- Bandit algorithms originated in gambling!
- A slot machine is also known as a one-armed bandit
 - A casino has multiple slot machines with different payouts
 - don't know which slot machine gives the highest payout
 - Can experiment over time to find out which slot machine is the best while maximizing payout
- Multi-armed bandits are algorithms
 - allow you to balance between
 - exploitation (choosing the slot machine that has paid the most in the past)
 - and exploration (choosing other slot machines that may pay off even more)
- When multiple models need to be evaluated, each model can be considered a slot machine whose payout (i.e., prediction accuracy) is unknown
- Bandits allow to determine how to route traffic to each model for prediction to determine the best model while maximizing prediction accuracy for users

Bandits(2)

- Bandit is stateful: before routing a request to a model, need to calculate all models' current performance
- Requires three things:
 - Model must be able to make online predictions
 - Preferably short feedback loops
 - A mechanism to collect feedback, calculate and keep track of each model's performance, and route prediction requests to different models based on their current performance
- Bandits are well-studied in academia and have been shown to be a lot more data efficient than A/B testing
 - require less data to determine which model is the best
 - at the same time, reduce opportunity cost as they route traffic to the better model more quickly
 - But a lot more difficult to implement than A/B testing because it requires computing and keeping track of models' payoffs



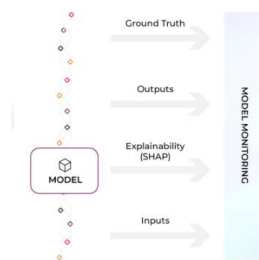
Model Monitoring

Pravin Y Pawar

Adapted from ["What is Model Monitoring?"](#)

What is Model Monitoring?

- A series of techniques used to detect and measure issues that arise with machine learning models
 - Once configured, monitors fire when a model metric crosses a threshold
- Areas of focus for monitoring include
 - model performance,
 - data quality,
 - drift detection
 - and embedding analysis



Why Is Model Monitoring Important?

- A lot can go wrong with a production model
 - navigating model issues can be challenging for even the most seasoned machine learning (ML) practitioner
- With model monitoring,
 - can immediately know when issues arise in machine learning models
 - better insights empower data scientists and ML engineers to pinpoint where to begin further analysis
- Model Monitoring Improves Business Outcomes
- Despite the multitude of problems a model can encounter in the real world,
 - over half of ML teams lack a reliable way to proactively surface when something is going wrong with a model in production
 - Many rely on
 - batch in-house solutions or dashboards that may not catch issues in time
 - tools that are not purpose-built for machine learning
- In an era where ML models relied on to increase profitability and even save lives, it's clear that better model monitoring is critical.

Model Performance Management

- Model performance indicates how model performs in production
 - Measure model performance with an evaluation metric,
 - which can be evaluated with daily or hourly checks
 - on metrics such as accuracy, recall, precision, F1, MAE, MAPE, and more
 - model type directs which performance metrics are applicable to model
- Performance monitor recommendations:
 - Performance monitors measure performance based on an evaluation metric
 - Can use performance metrics to compare model behavior between different environments
 - Use those insights to drill into the root cause of performance degradation
 - Important to look at the performance of models across various cohorts and slices of predictions

Drift Monitoring

- Drift monitors measure distribution drift, which is the difference between two statistical distributions
- Models are trained with polished data to represent production environments
 - it's common for real-world production data to deviate from training parameters over time
 - need to measure drift to identify if models have grown stale, have data quality issues, or if there are adversarial inputs in model
- Drift monitor recommendations:
 - To detect drift over time, set baseline using training data to identify how model changes between features, predictions, and actuals
 - To detect short-term drift, set baseline using historical production data (i.e. two weeks).
 - With proactive monitoring, detecting drift should be easy with automatic alerts
 - Bulk-create monitors with multiple baselines, view feature performance at a glance, access a historical view of drift, and access the distribution view associated with drift metric


Data Quality Monitoring

- ML models rely on upstream data to train and make predictions
 - Data is commonly collected from multiple systems, vendors, or can be owned by another team,
 - making it difficult to ensure always have high-quality data
- Model health depends on high-quality data that powers model features!
 - Important to immediately surface data quality issues to identify how data quality maps to model's performance
- Data quality monitors help identify key data quality issues
 - such as cardinality shifts, data type mismatch, missing data, and more
- Data quality monitor recommendations:
 - Use data quality monitors to detect shifts in upstream data and alert underlying changes
 - Configure data quality monitors to detect data issues like change in cardinality and change in percent of missing values

Monitoring Unstructured Data

- Most companies building computer vision (CV) or natural language processing (NLP) models lack a window into their models
 - are performing in production, running the risk of models impacting earnings or acting in unfair ways
- Since deep learning models rely on human labeling teams,
 - identifying new patterns in production and troubleshooting performance can be tricky
- CV and NLP Model Monitoring Recommendations:
 - Visualizing and monitoring embeddings — vector representations of data where linear distances capture structure
 - By monitoring embeddings, ML teams can proactively identify when their unstructured data is drifting

Date: 31st March 2024



BITS Pilani
Pilani Campus

Presentation

Model Monitoring, Drift Detection, Re-training and Continuous Learning

Abhishek Singh
Rajendra Mishra

Introduction

In machine learning system lifecycle, an important step begins, once we deploy our models to production.

Traditionally, with rule based, deterministic, software, the majority of the work occurs at the initial stage and once deployed, our system works as we've defined it.

But with machine learning, we haven't explicitly defined how something works but used data to architect a *probabilistic solution*.

This approach is subject to **natural performance degradation over time**, as well as unintended behaviour, **since gradually the data exposed to the model will be different from what it has been trained on**. This isn't something we should be trying to avoid but rather **understand and mitigate as much as possible**.

In addition, there are **many aspects of the machine learning system that we need to monitor** to ensure that our machine learning use cases works as expected.

- Infrastructure monitoring
- ML Model Monitoring
 1. Model Training Monitoring
 2. Model Serving Monitoring
- Model Serving resources monitoring

Agenda

- Introduction
- Dashboard Tools
- Infrastructure monitoring
- ML Model Monitoring
- Model Training Monitoring
- Model Tuning Monitoring
- Model Serving resources monitoring
- Model Serving Monitoring
- Model re-training and deployment
- Model Re-training Policy
- Stateless Vs Stateful model re-training
- Four Stages of Continuous Learning
- Summary
- Python Packages available to detect drifts
- References
- Demo

Dashboard Tools

Dashboard tools play a crucial role in the MLOps lifecycle by providing insights into model performance, detecting potential issues, and ensuring that machine learning systems remain robust and efficient.

Prometheus

Prometheus is an open-source monitoring and alerting toolkit designed for reliability and scalability. It is widely used for monitoring both machine learning models and infrastructure components, making it an essential tool for MLOps. Key features of Prometheus include:

- Flexible data model and powerful query language (PromQL), enabling users to define custom metrics and perform complex queries to gain insights into their systems
- Pull-based data collection, allowing Prometheus to scrape metrics from multiple sources, such as applications, databases, and other infrastructure components
- Built-in alerting system, enabling users to define custom alert rules and receive notifications when specific conditions are met
- Integration with popular visualization tools, such as Grafana, providing users with customizable dashboards and visualizations of their monitoring data

By incorporating Prometheus into their MLOps workflows, data scientists and engineers can effectively monitor their machine learning models and infrastructure components, ensuring the reliability and performance of their systems.



Dashboard Tools

Grafana

Grafana is an open-source analytics and monitoring platform that allows users to create, explore, and share dashboards and visualizations of their data. Grafana supports a wide range of data sources, including Prometheus, Elasticsearch, and InfluxDB, making it a versatile option for monitoring and performance management in MLOps. Key features of Grafana include:

- Customizable dashboards, allowing users to create and share visualizations of their monitoring data, such as model performance metrics and infrastructure metrics
- Flexible alerting system, enabling users to define custom alert rules and receive notifications through various channels, such as email, Slack, or PagerDuty
- Extensive plugin ecosystem, providing users with additional data sources, panels, and themes to customize their monitoring experience
- Integration with popular authentication and authorization systems, such as OAuth, LDAP, and GitHub, ensuring secure access to monitoring data

By using Grafana in their MLOps workflows, data scientists and engineers can gain valuable insights into their models' performance and infrastructure, identify potential issues, and optimize their systems for better results



ML Model Monitoring

Just monitoring the system's health won't be enough to capture the underlying issues with our model.

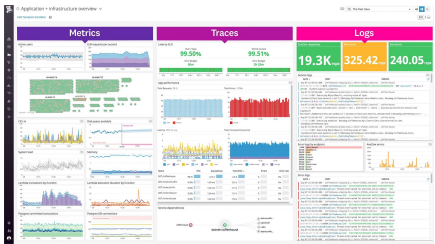
So we need various levels of model monitoring.

We need to **monitor training and tuning metrics**, **monitoring the model serving resources** e.g. throughput & latency, and finally, **model monitoring for issues like data drift, model decay** to take a decision to trigger re-training of the model in case the prediction drifts from ground truth beyond a threshold.



Infrastructure monitoring

The first step to ensure that our model is performing well is to ensure that the actual system is up and running as it should. This can include metrics specific to service requests such as latency, throughput, error rates, etc. as well as infrastructure utilization such as CPU/GPU utilization, memory, etc. Fortunately, most MLOps providers and even CaaS layers will provide this insight into our system's health through a dashboard. In the event we don't, we can easily use [Grafana](#), [Datadog](#), etc. to ingest system performance metrics from logs to create a customized dashboard and set alerts. (we need to rely on the underlying platform for this)



Model Training Monitoring

The first level of metrics to monitor and log involves the model training and tuning performance. These would be quantitative evaluation metrics that we used during model training and evaluation e.g. accuracy, precision, f1, etc. There are different metrics to be evaluated in case of regression or classification cases.

Regression Metrics

The most common metrics for evaluating predictions on regression machine learning problems:

- Mean Absolute Error.
- Mean Squared Error.
- R².
- Training parameters

Classification Metrics

Classification problems are perhaps the most common type of machine learning problem and as such there are a lots of metrics that can be used to evaluate predictions for these problems:

- Classification Accuracy.
- Logarithmic Loss.
- Area Under ROC Curve.
- Confusion Matrix.
- Classification Report.
- Training parameters



Model Tuning Monitoring

Once we've done model training and evaluation, it's possible that we want to see if you can further improve our training in any way. We can do this by **tuning hyper-parameters**. There were a few parameters we implicitly assumed when we did our training, and now is a good time to go back and test those assumptions and try other values. the following parameters can be monitored and logged:

- Fine-tuning parameters
- Metric related to model accuracy

In machine learning, the terms **model parameters** and **hyperparameters** are often used interchangeably. However, there is a subtle difference between the two.

•**Model parameters** are the values that are learned by the model during training. They are typically represented by a vector of numbers. The number of model parameters depends on the complexity of the model. For example, a simple linear regression model may have only a few model parameters, while a complex deep learning model may have millions or even billions of model parameters.

•**Hyperparameters** are the values that are set before training the model. They control the learning process and the behavior of the model. For example, a hyperparameter might be the number of iterations to train the model, the learning rate, or the batch size. Hyperparameters can have a significant impact on the performance of the model.

In general, **model parameters** are learned from data, while **hyperparameters** are set by the user. However, there are some cases where **model parameters** can be set by the user, such as when using a **pre-trained model**.



Model Serving Monitoring

There are 2 questions which we need to answer:

1. Does new incoming data reflect the same patterns as the data on which model was originally trained on ?
2. Is the model performing as well in development as it did in design phase ? If not, why ?

Machine Learning Model Drift

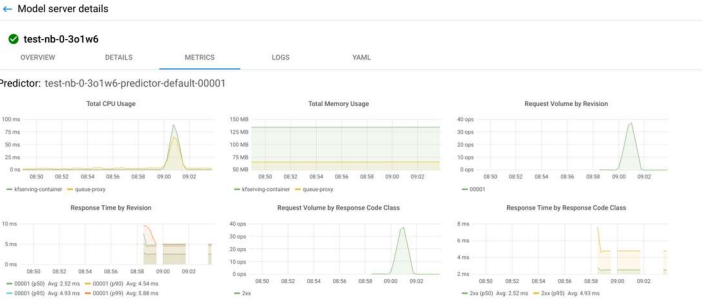
How well a model performs is the reflection of the data used to train it. If there is a significant change in distribution or composition of values of input variables or the target variables, it could lead to data drift which could cause the model to degrade. Model degradation can lead to inaccurate predictions and insights, therefore we want to carefully monitor model degradation. *To track model degradation there are 2 approaches to consider:*

1. **Based on Ground Truth** - Labelled data is compared to prediction
2. **Based on Data Drift** - Instead of waiting for the labelled data, the training data and recent data are compared statistically.



Model Serving resources monitoring

An important part of monitoring system resources is monitoring the performance of model serving infrastructure. Once the model is trained and served, we are going to use the model serving for quiet sometime. So its important to ensure that the performance of model serving is as per expectation..



Model Serving Monitoring

Model Monitoring based on Ground Truth vs Data Drift

The ground truth is the correct answer that the model was asked to solve -- when we know the ground truth for all the predictions a model has made, we can judge with certainty how well the model is performing.

Ground truth monitoring requires waiting for the label event and then computing the performance of the model based on these ground truth observations. When the ground truth is available quickly it can be the best solution to monitor model degradation, however obtaining ground truth can be slow and costly.

If our use cases requires rapid feedback, or if the ground truth is not available or hard to compute, input drift evaluation may be the way to go. The basis for input drift, is that the model is only going to predict accurately, if the data it was trained on is an accurate reflection of the real world, so if a comparison of the recent requests to a deployed model against the training data, shows distinct differences, then there is a strong likelihood that the model performance may be compromised.

Unlike for ground truth evaluation, the data required for input drift evaluation already exists, so there is no need to wait for any other information.

Note: If we wait to catch the model decay based on the ground-truth performance, it may have already caused significant damage to downstream business pipelines that are dependent on it. We need to employ more fine-grained monitoring to identify the sources of model drift prior to actual performance degradation.

Model Serving Monitoring



Types of Drift

We need to first understand the different types of issues that can cause our model's performance to decay (model drift). The best way to do this is to look at all the moving pieces of what we're trying to model and how each one can experience drift.

Drift Type	Description
Feature/Input Drift $\rightarrow (X)$	Input feature(s) distributions deviate. Also known as Covariate Shift .
Label/Target Drift $\rightarrow (y)$	Label/Target distribution deviates. Also known as Prior Probability Shift .
Prediction Drift $\rightarrow P(y)$	Model prediction distribution deviates. Also known as Model Drift .
Concept Drift $\rightarrow P(y X)$	External factors cause labels to evolve. Also known as Task Drift .

Model Serving Monitoring



Drift types and actions to take:

Feature/Data Drift	<ul style="list-style-type: none"> Investigate feature generation process Retrain using new data
Label/Target Drift	<ul style="list-style-type: none"> Investigate label generation process Retrain using new data
Prediction Drift	<ul style="list-style-type: none"> Investigate Model training process Access business impact of changes in prediction
Concept Drift	<ul style="list-style-type: none"> Investigate additional feature engineering Consider alternative approach/solution Retrain/tune using new data

Model re-training and deployment



- It's usually a good idea to establish policies around when we're going to retrain our model.
- There's no right or wrong answer here, so it will depend on what works in our particular situation.
- We could simply choose to retrain our model whenever it seems to be necessary.
- That includes situations where we detect a drift, but it also includes situations where we always retrain the models according to a schedule, or on demand.
- **In practice, this is what many companies do, because it's simple to understand and in many domains, it works fairly well.**
- It can, however, incur higher training and data gathering costs unnecessarily.
- If we can automate the process of detecting the conditions which require model retraining, that will be ideal.
- That includes being able to detect model performance degradation or when we detect significant data drift, and triggering retraining.
- In both cases, in order to automate retraining, we should have data gathered and labelled automatically and only retrain when sufficient data is available.
- We need to have continuous training, integration and deployment setup to make the process fully automated.

Model Re-training Policy



Policy	Intuition
On Demand	Re-train the model on need basis (e.g. when ACTS detects that the deployed model is old, it triggers re-training and deployment)
On Schedule	Re-train at a fixed interval on a daily, weekly or monthly basis.
On Data Drift	We find significant changes in the input data and trigger model retraining.
On Model Performance Degradation	Prediction Drift based on ground truth, trigger model retraining.
On New data availability	When new data is available on ad-hoc basis and there is no trend. Re-train model when new labelled data is collected and available in source database.

Stateless Vs Stateful model re-training



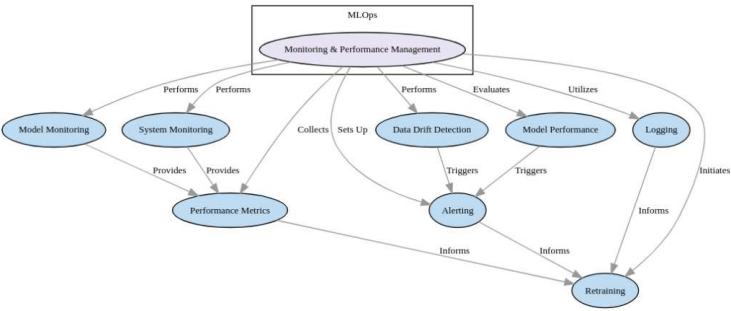
Policy	Intuition
Stateless	<ul style="list-style-type: none">New model is retrained from scratch each time.
Stateful	<ul style="list-style-type: none">The existing model continues to be trained on new dataStateful re-training is also known as fine-tuning or incremental learning

Four Stages of Continuous Learning



Stage	Description
Stage 1: On-Demand, Stateless retraining	This is the first stage of continuous learning. In this stage we update the models on demand. In this stage the models are updated on need basis or as deemed suitable.
Stage 2: Automated, Stateless retraining	In this stage we write automated and recurring pipelines which rely mostly on schedule to trigger re-training. Most companies with somewhat mature ML infrastructure are in this stage.
Stage 3: Automated, Stateful retraining	In this stage we update existing model which continues to be trained on new data. Stateful re-training is also known as fine-tuning or incremental learning.
Stage 4: Continual learning	Till Stage 3 our models are still updated based on a fixed schedule. This might not be optimal always as quite a few models might not show any performance degradation and hence the stage 3 might lead to higher compute costs. On the flip side some of our models could decay/degrade much faster and require a much faster and automated retraining. In stage 4 our re-training trigger can be combination of one or more factors e.g. data drift, model performance degradation (beyond threshold) or arrival of new data.

Summary



Python Packages available to detect drifts



Some of the open-source packages available to detect drift are:

Evidently AI: Open-Source Tool To Analyze Data Drift. Evidently is an open-source Python library for data scientists and ML engineers. It helps evaluate, test, and monitor the performance of ML models from validation to production.

drift_detection: This package contains some developmental tools to detect and compare statistical differences between 2 structurally similar pandas dataframes. The intended purpose is to detect data drift — where the statistical properties of an input variable change over time. It provides a class DataDriftDetector which takes in 2 pandas dataframes and provides a few useful methods to compare and analyze the differences between the 2 datasets.

skmultiflow: One of the key features of skmultiflow is its ability to handle concept drift, which is the change in the underlying distribution of the data over time. It includes algorithms for detecting concept drift and adapting the machine learning model in real-time to account for the changes in the data. This makes it an ideal tool for building machine learning models that can adapt and continue to perform well as the data changes over time.

Deepchecks: Deepchecks Open Source is a python library for data scientists and ML engineers. The package includes extensive test suites for machine learning models and data, built in a way that's flexible, extendable, and editable.

NannyML: NannyML is an open-source Python library that helps you monitor and improve the performance of your machine learning models

References



- [An overview of unsupervised drift detection methods](#)
- [Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift](#)
- [Monitoring and explainability of models in production](#)
- [Detecting and Correcting for Label Shift with Black Box Predictors](#)
- [Outlier and anomaly pattern detection on data streams](#)

Thank You !



Demo