MLOps AML notes(Arka's(Borda's) (DEEP)research work)

1 Introduction to MLOps

- In MLOps, **compute and storage are decoupled**, meaning that storage and computation are independent.
- This ensures that **data persists even if the system is turned off**, preventing data loss.

2 System Architecture Overview

The architecture typically consists of:

- **Data Store** → Stores raw data.
- **Server** → Performs computations.
- **Model Store** → Stores trained models.

Diagram Representation:

```
Data Store  -->  Server  -->  Model Store
```

3 Scalability in MLOps

- **Scaling Storage:** If additional storage is required, more data storage can be added.
- **Scaling Computation:** If additional computing power is required, more servers can be added.
- The system is designed to scale up or down **based on demand** dynamically.

4 Computation and Performance Optimization

- **Data Loading Process:**
  - Data is first loaded from the **data store to the server**.
  - Computation happens at **RAM level** instead of disk level.
  - This results in a **10x increase in speed** and **lower latency** compared to disk-level computation.

5 Evolution from Hadoop to Spark

5.1 Traditional Systems (Hadoop)

- Hadoop partially **decoupled computation and storage**.
- However, **most computations happened on disk**, leading to slower performance.

5.2 Modern Systems (Spark)

- Spark introduced a shift by **moving most computations to RAM**, significantly improving speed and efficiency.

6 Introduction to MLOps

- In MLOps, **compute and storage are decoupled**, meaning that storage and computation are independent.
- This ensures that **data persists even if the system is turned off**, preventing data loss.

7 System Architecture Overview

The architecture typically consists of:

- **Data Store** → Stores raw data.
- **Server** → Performs computations.
- **Model Store** → Stores trained models.

Diagram Representation:

```
Data Store  -->  Server  -->  Model Store
```

8 Scalability in MLOps

- **Scaling Storage:** If additional storage is required, more data storage can be added.
- **Scaling Computation:** If additional computing power is required, more servers can be added.
- The system is designed to scale up or down **based on demand** dynamically.

9 Computation and Performance Optimization

- **Data Loading Process:**
  - Data is first loaded from the **data store to the server**.
  - Computation happens at **RAM level** instead of disk level.
  - This results in a **10x increase in speed** and **lower latency** compared to disk-level computation.

10 Evolution from Hadoop to Spark

10.1 Traditional Systems (Hadoop)

- Hadoop partially **decoupled computation and storage**.
- However, **most computations happened on disk**, leading to slower performance.

10.2 Modern Systems (Spark)

- Spark introduced a shift by **moving most computations to RAM**, significantly improving speed and efficiency.

11 Experiment Tracking in MLOps

- **Git is not designed to handle ML systems** due to data versioning complexities.
- **Experiment Tracking** involves:
  - Taking data.
  - Training the model.
  - Adding more data or modifying weights/hyperparameters.

Key Concept: Experiment tracking allows recording of all changes to ensure **reproducibility**.

11.1 Popular Experiment Tracking Tools

- **MLflow**
- **Weights and Biases (W&B)**
- **OpenAI Tracking System**

Example MLflow command:

```
mlflow.log_metrics({"train_loss": train_loss, "val_loss": val_loss}, step=epoch)
```

12 Principles of MLOps

MLOps follows principles similar to DevOps:

- **Consideration of ML Systems**: - Whether predictions are given in batch mode or real-time stream mode determines the system architecture.
- **Version Control**: - Writing and deploying multiple versions of ML code requires version control. - **Git** is commonly used for this.

12.1 ML Version Control Challenges

- Version controlling just the code is insufficient.
- ML systems also require **data** and **model** versioning.

Comparison of Version Control Tools:

Data	Model	Code
DVC (Data Version Control)	MLflow	Git

13 Challenges in Using Git for ML

- Storing large datasets in Git is impractical.
- As data grows, Git repositories become too large.

14 Data Version Control (DVC)

14.1 Handling Large Datasets

- Data is often very large, making it impractical to push directly to Git.
- Instead of pushing large files, we use `data.xml.dvc`, which is a lightweight pointer to the actual dataset stored in the DVC repository.
- The actual data is stored in a **DVC remote storage** (e.g., Google Drive, AWS S3).

14.2 Ignoring Large Files in Git

- Use `.gitignore` to specify files that should be ignored by Git.
- This helps avoid pushing large dataset files.

14.3 Versioning Datasets

- Copy datasets for version control:

```
cp data/data.xml /tmp/data.xml
```

- View commit history:

```
git log --oneline
```

- Checkout a previous dataset version:

```
git checkout HEAD~1 data/data.xml.dvc
```

14.4 Basic Idea of DVC

- **DVC extends Git to track large files** outside of the Git repository, similar to how Google Drive works.
- Git handles version control, while DVC manages large dataset files.

15 Steps to Set Up Data Version Control

1. Initialize Git:

   ```
   git init
   ```

2. Initialize DVC:

   ```
   git:(master) dvc init
   ```

3. Create a data directory:

   ```
   mkdir data
   ```

4. Use `dvcget` to download data from a registry to local storage:

   - `dvcget` is similar to `curl` for fetching remote files.

5. Add dataset to DVC:

   ```
   dvc add data/filename
   git add data/filename.dvc
   git commit -m "Add raw data"
   ```

6. Push dataset to remote storage:

   ```
   dvc remote add -d storage gdrive://ID_of_folder
   ```

16 Code Versioning and Git Remote

16.1 Git Remote Repository

- A Git remote repository is stored in the cloud, not on a local PC.
- The remote repository is linked to a local repository for syncing.

16.2 Git Workflow

Git Commands Workflow:

```
git add  ->  git commit  ->  git push
```

Data Flow:

```
Working Directory  ->  Staging Area  ->  Local Repo  ->  Remote Repo
```

Restoring Previous Versions:

```
git restore  ->  git checkout / revert
```

17 Artifacts and Data Tracking

- Large datasets grow over time, but most changes are small.
- Git and DVC maintain **hash values** to track incremental changes efficiently.
- Maintaining hash-based tracking helps in:
  - Efficient storage management.
  - Versioning of datasets with minimal redundancy.
  - Faster rollback and recovery of data.

18 Request-Response Model and APIs

18.1 What is an API?

- API stands for **Application Programming Interface**.
- It allows different systems to communicate with each other.
- An API endpoint is a URL used to send and receive requests.

Example API Endpoint:

`http://www.xyz.com/recommendation/predict`

18.2 Public vs Local API

- APIs can be either **public** (accessible over the internet) or **local** (restricted within a network).

18.3 Request and Response Format

- The API follows a **request-response model**:
  - Request is sent in **JSON format**.
  - Response is received in **JSON format**.

19 Continuous Integration and Continuous Deployment (CI/CD)

19.1 Definitions

- **CI (Continuous Integration)**: The practice of automatically integrating code changes.
- **CD (Continuous Deployment)**: The practice of automatically deploying code changes to production.

19.2 CI Pipeline Stages

- **Build** → **Test** → **Merge**.
- Automated testing ensures the code passes before merging.

CI Workflow:

```
Build  -->  Test  -->  Merge
```

19.3 Continuous Delivery and Deployment

- **Continuous Delivery**: Automatically releases code to a repository.
- **Continuous Deployment**: Automatically deploys code to production.

20 Distributed Training in MLOps

20.1 What is Distributed Training?

- Distributed training distributes computations across multiple machines to speed up model training.
- This is essential for training large models in a reasonable amount of time.

20.2 Ray Train Framework

- **Ray Train** is a framework for distributed training.
- It consists of:
  - **Head node**: Orchestrates the training process.
  - **Worker nodes**: Train the model and send results back to the head node.

Distributed Training Workflow:

```
Head Node
    |
    |--> Worker Node
    |--> Worker Node
    |--> Worker Node
```

21 Cloning a Repository

21.1 What is Cloning?

- Cloning copies a remote Git repository to a local computer.

21.2 Command to Clone a Repository

```
git clone <repository-url>
```

22 MLflow Experiment Tracking

22.1 Tracking Dataset Versions

- MLflow logs references to dataset versions or data markers to track which dataset was used in each experiment.
- This ensures reproducibility by maintaining dataset logs.

22.2 Model Performance Comparison

- Different datasets or subsets can be compared in MLflow to evaluate model performance.

22.3 Logging Experiment Parameters

- MLflow logs:
  - Hyperparameters
  - Training metrics
  - Model artifacts

22.4 Model Registry and Deployment

- A model is stored in a registry to maintain version control.
- Production-ready models are logged and monitored.

23 Git Hooks and CI/CD Pipeline Integration

23.1 Git Hooks for Automation

- **DVC (Data Version Control)** maintains dataset versions.
- **Git** maintains pointers to dataset versions.
- Any change in a data pointer can trigger a CI pipeline.

23.2 Triggering Model Training

- A push or merge in the Git repository can trigger a CI/CD pipeline.
- The pipeline runs an integration test before retraining the model.

24 Data and Model Updates in MLOps

24.1 Data Updates

- Dataset versions are updated via DVC.
- Version updates trigger workflow events such as:
  - Data cleaning
  - Validation
  - Feature engineering

24.2 Model Updates

- When raw data is ingested or code changes, an automated pipeline retrains the model.
- The pipeline computes evaluation metrics.
- If the model exceeds performance thresholds, it is automatically marked for production.

24.3 Handling Data Drift

- Automated data profiling detects data drift.
- If data drift is detected:
  - A data drift alert is generated.
  - The pipeline may trigger a new model training cycle.

25 CI/CD Pipeline for Software Production

25.1 Overview of CI/CD

- CI/CD (Continuous Integration and Continuous Deployment) automates the software release process.

25.2 CI/CD Workflow

- **Software Development:** - Version control maintains code changes in a repository. - A CI/CD pipeline is triggered.
- **Build Phase:** - Merges source code with dependencies. - Creates a runnable software instance.
- **Testing Phase:** - Unit tests validate individual components. - Integration tests ensure system-wide compatibility. - Cross-platform testing verifies performance across environments.
- **Deployment Phase:** - Deploys the system to a production-like environment. - Automation testing ensures reliability. - The final step is **production validation**.

CI/CD Pipeline Steps:

```
Software Developed  ->  CI/CD Pipeline Triggered
   |
   v
Build Phase  ->  Test Phase  ->  Deployment Phase
```

26 Model Testing and Monitoring

26.1 Distribution Checks

- Check if the dataset distribution is balanced before training.

26.2 Model Considerations

- **Tolerance**: Slight changes in data should not drastically change predictions.
- The model should maintain **minimum functionality**.
- **Behavioral Check**: Does changing certain words (e.g., *"amazing"* to *"horrible"*) change sentiment predictions accordingly?

26.3 Testing vs Monitoring

Testing (Pre-deployment)

- Offline validation.
- Fixed dataset.
- Checks known requirements.

Monitoring (Post-deployment)

- Live data checks.
- Detects data drift.
- Identifies edge cases.
- Tracks prediction quality.

27 Types of Testing in CI/CD Pipelines

27.1 Testing in CI/CD

- Each successful CI/CD pipeline run can build a new Docker image and push it to a registry.

27.2 Types of Tests

- **Unit Tests**: Check functionality of individual components.
- **Integration Tests**: Verify if different parts work together across platforms.
- **System Tests**: Validate the entire system for failure points.
- **Acceptance Tests**: Ensure the system satisfies basic user requirements.

Regression Testing

- Re-tests previously fixed bugs after code updates.

27.3 What to Test?

Data Schema Validation

- Check every aspect of the data schema.
- Validate columns, row formats, and data structure.

Data Quality Check

- Detect missing values in critical fields.

28 Model Deployment

28.1 Deploying Models as Microservices

- Deployed using **Flask** as a web service.

28.2 Blue-Green Deployment

- Strategy used by cloud services like AWS to deploy new models while keeping the old model live.

28.3 Web Serving with Flask

- A Flask app can host an endpoint that:
  - Receives feature data.
  - Returns real-time predictions.

28.4 Containerization for Deployment

- Containers help package model inference and preprocessing pipelines.
- Docker is used to containerize the model.

Deployment Workflow:

```
Model + Inference Code  -->  Docker Image  -->  Deploy via Kubernetes
```

29 Eyeballing Predictions

- Manually inspect correct/incorrect predictions to catch subtle issues that automated tests might miss.
- Example: Confusing *"apple"* with *"apple inc."* due to entity recognition errors.

29.1 Automated Testing

- Code testing across environments can be automated using `pytest`.

29.2 Profiling for Optimization

- Measure memory and time consumption of critical steps.
- Identify and optimize performance bottlenecks.

30 Out-of-Range Value Checks

30.1 What?

- Validate numerical ranges and categorical values.

30.2 Why?

- Catch errors such as:
  - Negative ages.
  - Invalid categorical values.

Example check for valid age range:

```
assert df["age"].between(0, 120).all()
```

31 Model Overfitting Check

31.1 What?

- Compare training vs. validation metrics (loss/accuracy).

31.2 Why?

- Ensure the model generalizes well beyond the training dataset.

Example assertion for loss comparison:

```
assert val_loss < 1.2 * train_loss
```

32 Target Distribution Check

32.1 What?

- Verify class balance of target features (for classification) or output value range (for regression).

32.2 Why?

- Prevents bias toward dominant classes.

Example check for class imbalance:

```
assert df["label"].value_counts(normalize=True).min() > 0.1
```

33 Key Predictor Distributions

33.1 What?

- Compare feature distributions between training and production data.

33.2 Why?

- Detect data drift.

33.3 Tools for Distribution Checks

- Kullback-Leibler (KL) Divergence Test.
- **Great Expectations** for data validation.