Models

=> Saving models (pytorch, sklearn ...)

MLflow - Projects 4-11-2025

1 => python 1_tracking.py
2 => mlflow run . -e tracking --env-manager local
↳ local (can give github repo as well)

- Command 1 ran, but command 2 didn't. Since mlflow requires some more data which we didn't give in command.
- We actually defined experiment name in tracking.py but didn't tell the same name in command so it created another experiment & conflicted with file name.
- Add " --experiment-name Basic_Tracking_Demo " at the end of line 2.
- Can use remote github repos (mlflow project) & parameterize dataset (Maybe I wanna use my own data)
- The "local" in line # 2 tells mlflow to use current virtual environments. We can also set a default in mlflow project or give an Environment (as configured in yaml) as parameter.

MLflow - Model.

=> Takes the trained model (with however many parameters) and from whomever framework (sklearn, Pytorch) and wraps it in a single wrapper (with the information needed by mlflow) to make mlflow model.

=> After it is saved, all of every model comes down to a single interface for inference & whatnot.

---

infer - Signatures:

Models.py:

1) Made an sklearn model
2) Made a Pytorch model
3) Made a custom sklearn model with scaling. Same mlflow syntax.

4) load sklearn model & predict
5) " Pytorch (custom) model & predict.

MLflow - Model Registry

- Provides centralized
  1) Model store
  2) Versioning
  3) Life-cycle (Name -> Staging -> Production -> Archived)

(current) stage variable

=> Files till now have logged models which appears on the dashboard but that's not the same as register (model_registry)

=> Registering models (for UI & manging repository).
  1) Version
  2) Stage
  3) Tags
  4) Aliases

=> Compare models
-> fetch & do whatever you want with all ~~data~~ models, their ~~paran~~ versions, parameters, metrics etc...

MLFlow Basic Runs

1) mlflow.set_experiment("Experiment_name")
2) with mlflow.start_run("run_name = "basic_rf_model") as run:
3) ##
   # create params dictionary
3) mlflow.log_params(params_dict)
   # train model, predict, save metrics
4) mlflow.log_metrics(metrics_dict)

5) mlflow.set_tag("<tag_type>", "<tag_string>")
   # create artifacts
6) mlflow.log_artifact("<artifact_path>")

7) infer_signature(X_test, model.predict(X_test))

8) mlflow.sklearn.log_model(model, "model", signature=signature,
   input_example=X_test[:5]
   metadata={
   "model_type": "RandomForest",
   "task": "binary_classification",
   "framework": "sklearn",
   "version": "1.0.0"}

1) Simple tracking
2) Hyper-parameter tuning
3) Batch-logging
4) custom_artifacts

MLProject
conda.yaml
python_env.yaml
requirements.txt
train.py

MLProject:

name: mlflow_tutorial

python_env: python_env.yaml

entry_points:

main:
command: "python run_all.py"

tracking:
command: "python 1_tracking.py"

projects:
parameters:
n_estimators: {type: int, default: 10}
dep max_depth: {type: int, default: 5}
command: "python 2_projects.py {n_estimators} {max_depth}"

models:
command: "python 3models.py"

model_registry:
command: "python 4_model_registry.py"

python_env.yaml:

```
python: "3.8"
build-dependencies:
  - pip
dependencies: 2.8.0
  - mlflow >=
  - pandas >= 2.0.0
```

```
python: "3.8"
build-dependencies:
  - pip
dependencies:
  - mlflow >= 2.8.0
```

→ Once model is logged: run_id = run.info.run_id

model_uri = f"runs:/{run_id}/model"

1) loaded_sklearn_model = mlflow.sklearn.load_model(model_uri)

2) loaded_pyfunc_model = mlflow.pyfunc.load_model(model_uri)

// Predictions

1) loaded_sklearn_model.predict(X_test[:5])

2) loaded_pyfunc_model.predict(X_test[:5])

3) loaded_pyfunc_model.predict(X_test_df)

Inheritance allows for custom preprocessing & shit: mlflow.pyfunc.PythonModel

①

```
model_name = "classification_model"
model_info = mlflow.sklearn.log_model(
        model,
        "model",
        signature = signature,
        registered_model_name = model_name
)
```

↓ ②

Just use this again. If you want to edit, i.e ~~version~~ version 2.

↓ ③

If you keep the model name same you can even change the whole model, different algorithm and shize. It will just create a newer version.

↓ ④

```
client = MlflowClient()

client.search_model_versions(f"name = {model_name}")

client.transition_model_version_stage(
    name = model_name,
    version = 1
    stage = "Staging"
)

client.update_registered_model(    # adding description to model
    name = model_name,
    description = "...."
)
```

```
client.update_model_version(    //updating version descriptions.
    name = model_name,
    version = 1,
    description = "...."
)

client.set_model_version_tag(model_name, 1, "algorithm", "RandomForest")
                                            <variable> <value>

client.get_registered_model(model_name)

//loading models from registry: (by version)

model_version_uri = f"models:/{model_name}/2"

loaded_model_v2 = mlflow.sklearn.load_model(model_version_uri)

predictions = loaded_model_v2.predict(X_test[:5])

// (by stage)

model_stage_uri = f"models:/{model_name}/Production"

// (latest_version)
```

Compare models & aliasing!.

APACHE AIRFLOW

11-10-2025

1) A tool for authoring, scheduling, and monitoring

2) Directed Acyclic Graphs (DAG). Workflows. Airflow tasks:

=> Task: An instantiation of an operator.

=> Operator / Sensor / Hook : Operator performs actions (BashOperator, PythonOperator).
Sensors wait for external conditions.
Hooks implement connections to external service.

=> Scheduler: Schedules DAG runs & creates task instances.

=> Executor: Controls parallelism and how tasks run (Sequential, Local, Celery, K8s, etc.)

=> XCom: Lightweight per-task-instance communication system

=> Variables & Connections: Global config & credentials storage.

APACHE AIRFLOW

=> Better way to run is Docker-Compose.

Docker-Compose:

1) Postgres (for db)

2) x-airflow-common (does'nt get executed by docker-compose, to use somewhere else in the file)

3) airflow-scheduler

4) airflow-init

Dags Folder:

Logs FOLDER:

=> Every thing about dags is logged over here

Demo: (UI)

1) Added DAG file manually to dags folder (vscode) reflected on airflow UI.

2) DAG runs. Decoding the UI.

3) Run vs Parse DAG

4) x Communication.

DAG Code: (The file added manually to dag folder)

1) Imports
2) default_args.
3) with DAG (
    dag_id:
    default_args: default_args
    ...
) as dag:

    start_task = BashOperator (
        task_id = "start"
        bash_command = 'echo "Starting pipeline at $(date)"'
)

    process_task = BashOperator (
        ...
)

    complete_task = BashOperator (
        ...
)

start_task >> process_task >> complete_task # dependencies (linear)

Example_2 : Code file (DAG) (ETL example to show dependency)

1) Uses decorators @dag & @task.

2) Define tasks as functions using @task.

3) Define dependencies implicitly by just using tasks as functions and feeding the returned data forward.

4)

a) Seeing it on UI

b) x Can display on UI

c) Showing & explaining automatic & manual methods of x Can (on ReadMe).

AIRFLOW

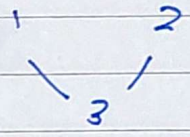
Task Dependencies:

- Linear

  task1 >> task2 >> task3

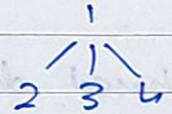
- Multiple upstream tasks

  task1 >> task3
  task2 >> task3



- Multiple downstream tasks:

  task1 >> [task2, task3, task4]

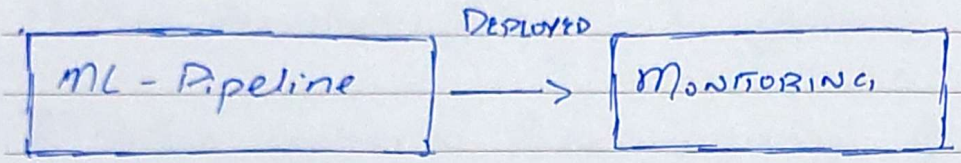


- Complex dependencies.

=> Connections
=> Hooks

PROMETHEUS & GRAPHANA - MONITORING

20-11-2025



PROMETHEUS: Getting data from the application / micro-service cluster

Graphana:

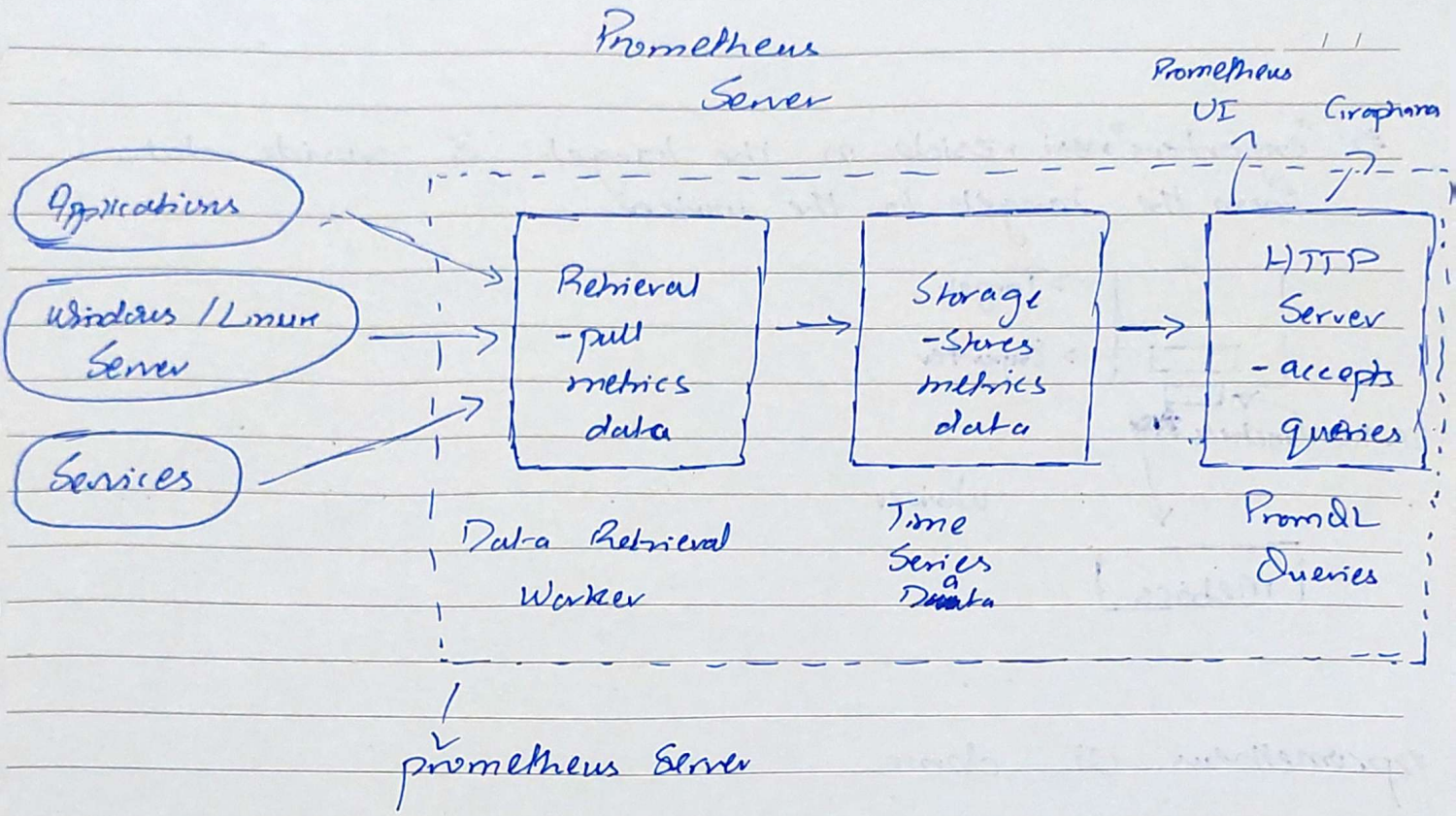
Prometheus:

- => Pull based data scraping
- => Multi-dimensional data labels
- => Uses Prom QL (The query language used by prometheus to query data)
- => Local timeseries storage
- => Integration with alert manager with alerts.

Graphana:

- => Interactive Dashboard
- => Time series Visualization
- => Alerts
- => Integrates with multiple data sources (including prometheus.)

=> We use docker-compose to run these together (prometheus, graphana, alert-manager)



Targets:

=> What Sources does prometheus monitor

1) Linux / Windows Server
2) Applications
3) Services

Metrics:

=> What units are monitored from those targets.

=> Format: Human readable text

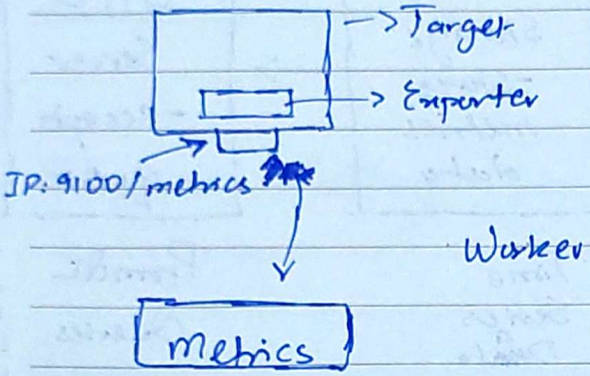
=> Metrics Entries: TYPE & HELP attributes.

↳ description of the unit being tracked

- Counter: How many times has X happened
- Guage: What is the current value of X
- Histogram: for how long X has occured (magnitude)

PAPERWORK

=> "exporters" reside in the target & provide data from the target to the worker.



=> prometheus UI demo

25-11-2025

Prometheus Code Demo:

1) docker-compose.yml

2) prometheus:
   prometheus.yml (all the endpoints running / exporters etc.)
   alert_rules.yml (conditions to generate alerts)

3) alert_manager
   alert_manager.yml (tells what to do when the alert has been generated

4) grafana

5)

Date

PromQL

1) variable_name (execute)

2) variable_name {mode = 'idle'} → filter by parameter value

3)

GRAPHANA

1) Install prometheus plugin.
2) Add new connection

unlimited res ...
cpu usage max.

/dev/null
yes > ~~kbuild~~

Date

alert_rules.yml:

=> Gives the PromQL query to run
=> The ~~time~~ time that will be waited before generating alerts (to remove false positives)
=> Gives summary & description of alerts.

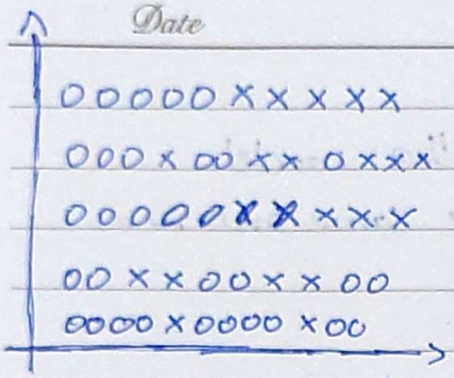
=> Blackbox: An alert definition that checks if endpoints are up.

alert_manager demo
connection with slack
details on github (will be uploaded by sir)

Date

ML Drift

2-12-2025



OOOOOXXXXX abrupt change X → label 2

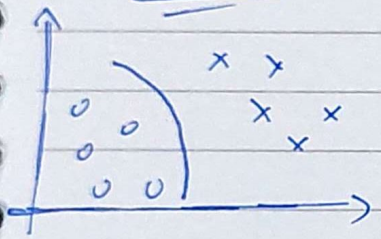
OOOXOOXXOXXX gradual O → label 1

OOOOOXXXXXX incremental

OOXXOOXXOO recurrent patterns (Seasonal - not weather wala, events that repeats)

OOOOXOOOOXOO blip (it may not be drift)

CONCEPT DRIFT:



=> For eg a model learnt this boundry based on the data / label values.

=> Now we change a concept / condition in the real world that changes the labels on those same data

=> So our data didn't change but the label formation did and hence we need a new ~~training~~ boundary line (retraining)

=> We change a condition that people below 200k instead of 150k will not be given a loan

DATA DRIFT:

1) LABEL DRIFT
2) FEATURE DRIFT

1) LABEL DRIFT:

=> We didn't change any concept

=> The values of the input didn't change

=> We noticed a change in the occurances of labels. Maybe because the instances of inputs increased.

=> More people applying for loan, (More rejected & accepted)

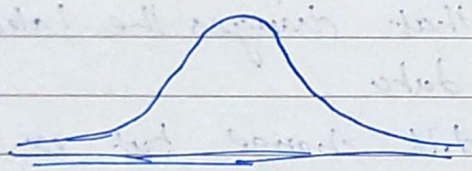
Labels for both will drift

Date

2) Feature drift.
   ⇒ Values of features change so the model predicts a different label for the same instance.

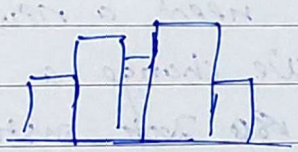
DETECTING DRIFT IN Unsupervised Learning

1) Numerical Variables



⇒ mean
⇒ Std deviation
⇒ wagners

2) Categorical
   ⇒ Frequency trends



3) Relations b/w numerical variables
   ⇒ pearson (r^2)

4) Relation b/w
   F_1 & F_2 two features can have low correlation (r^2) but still be strongly linked (quadratically or any other non-linear relation) since r^2 is just a linear correlation mapping.