ML lifecycle III

Big data management

Recap ML lifecycle

Part 1: Data preprocessing

Part 2: Model building

Part 3: Parameter tuning, model monitoring and deployment (today)

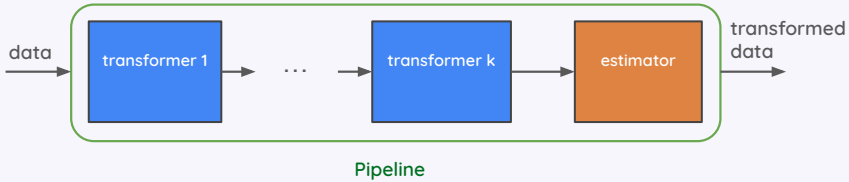
Recap ML lifecycle

Part 1: Data preprocessing

- combine datasets, missing data, scaling, feature transformations
- training, testing, cross-validation
- pipelines (preprocessing or prediction)

```python
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

```python
from sklearn.compose import ColumnTransformer

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
        ("num", num_pipeline, num_attribs),
        ("cat", OneHotEncoder(), cat_attribs),
    ])

housing_prepared = full_pipeline.fit_transform(housing)
```



Recap pipelines

```
t1 = Transformer1()
t2 = Transformer2()
clf = Classifier()
```

```
train_t1 = t1.fit_transform(X_train)
train_t1_t2 = t2.fit_transform(train_t1)
clf.fit(train_t1_t2, y_train)
```

```
test_t1 = t1.fit_transform(X_test)
test_t1_t2 = t2.fit_transform(test_t1)
test_predicted = clf.predict(test_t1_t2)
```

Any errors?

Recap pipelines

```
t1 = Transformer1()
t2 = Transformer2()
clf = Classifier()
```

```
train_t1 = t1.fit_transform(X_train)
train_t1_t2 = t2.fit_transform(train_t1)
clf.fit(train_t1_t2, y_train)
```

```
test_t1 = t1.transform(X_test)
test_t1_t2 = t2.transform(test_t1)
test_predicted = clf.predict(test_t1_t2)
```

Recap pipelines

```
t1 = Transformer1()
t2 = Transformer2()
clf = Classifier()
```

```
train_t1 = t1.fit_transform(X_train)
train_t1_t2 = t2.fit_transform(train_t1)
clf.fit(train_t1_t2, y_train)
```

```
test_t1 = t1.transform(X_test)
test_t1_t2 = t2.transform(test_t1)
test_predicted = clf.predict(test_t1_t2)
```

```
pipeline = Pipeline([
       ('t1',  Transformer1()),
       ('t2',  Transformer2()),
       ('clf', Classifier()),
       ])
```

```
pipeline.fit(X_train,y_train)
```

```
test_predicted = pipeline.predict(X_test)
```

Recap ML lifecycle

Part 2: Model building

- decision trees
- ensembles of models
- Xgboost - ensemble of trees; optimisations for big data
- SVM - computationally expensive for big data, but some optimisations for big data
- neural networks:
  - series of linear transformations with added non-linearity through activation functions
  - distributed deep learning
    - data parallelism vs fully sharded data parallelism

Quiz ML recap

Quick quiz to recap what you learnt so far regarding the ML lifecycle.

Check LearnIT page and search for ML lifecycle recap (Lecture 9).

ML challenges

Software development

meet functional requirements

ML models

optimize a specific metric (e.g. prediction accuracy)

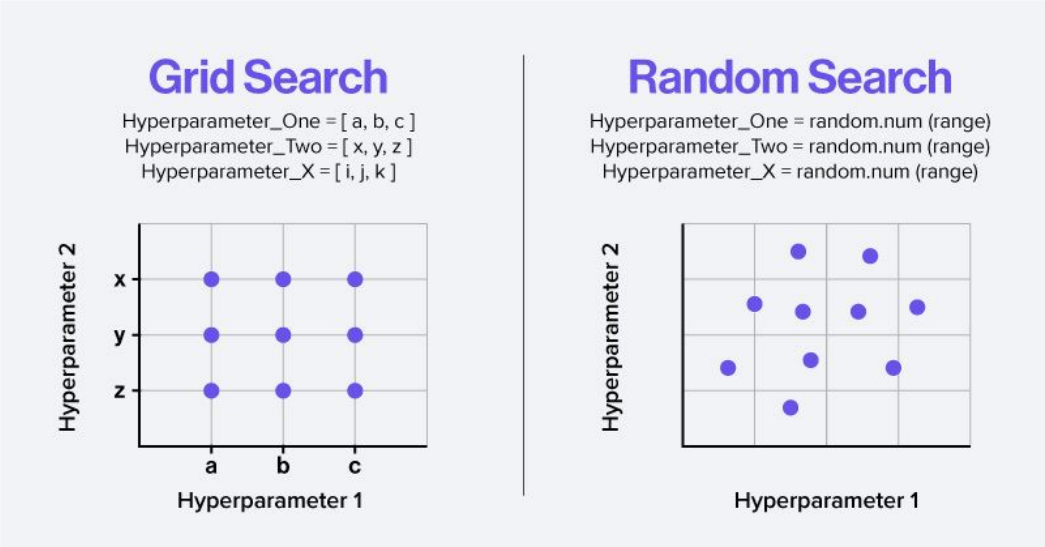
performance depends on training data

development revolves around **experimentation**

Experimentation

- Try out different transformations (scalers, data imputation etc.)
- Tuning hyperparameters
  - for example with NN: define number of layers, neurons, activation function etc.

Finding the best parameters is a complex process.



Experimentation with grid search and cross-validation

```
pipeline = Pipeline([
    ('scaler', StandardScaler()),  # This will be replaced during grid search
    ('mlp', MLPClassifier())
])

# Define parameter grid for hyperparameter tuning
param_grid = {
    # Try different scalers
    'scaler': [
        StandardScaler(),
        MinMaxScaler()
    ],

    # Try different network architectures (hidden layer sizes)
    'mlp__hidden_layer_sizes': [
        (50,),           # 1 hidden layer with 50 neurons
        (100,100)          # 2 hidden layers with 100 neurons each
    ]
}

# Create GridSearchCV object
grid_search = GridSearchCV(
    pipeline,
    param_grid,
    cv=3,  # 3-fold cross-validation
    scoring='accuracy',
)

# Fit the grid search
grid_search.fit(X_train, y_train)
```

define pipeline

parameters to experiment with and their possible values

define grid search

find the parameters that give highest score (here: accuracy)

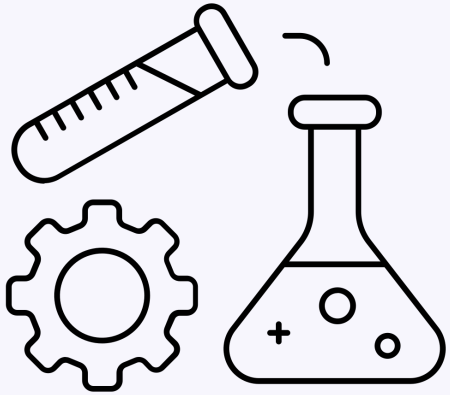
Model development

Traditional software often has a defined set of product features to be built.

ML development tends to revolve around experimentation.

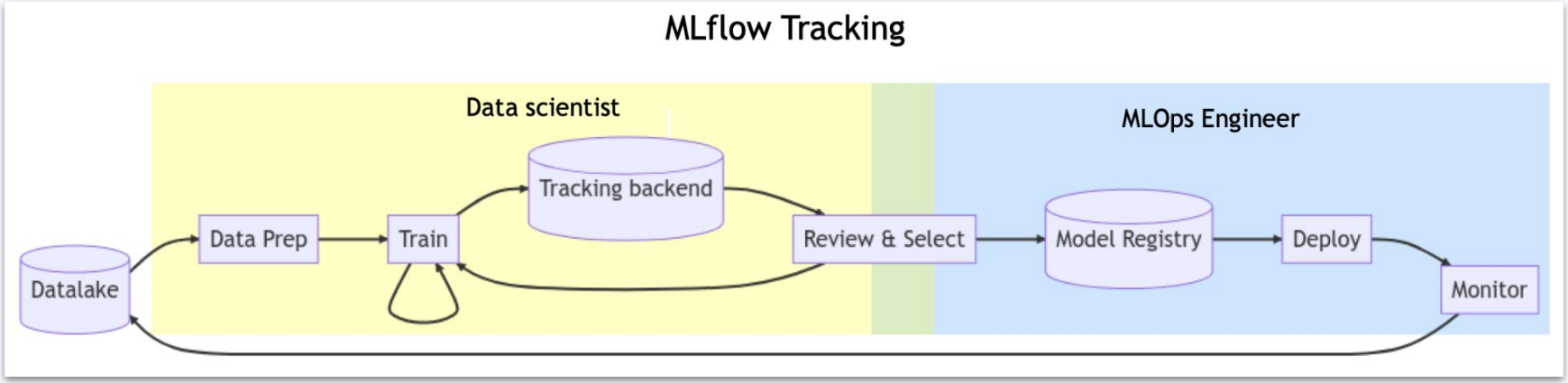
Experiment with:

- new datasets
- models
- software libraries
- tuning parameters



Open source platform for managing the end-to-end machine learning lifecycle:

- Track **experiments**
- Package ML code in a **reproducible** form
- **Deploy** models
- Collaboratively manage the full lifecycle of an MLflow Model, including model versioning, stage transitions, and annotations



MLflow is an open source platform to manage the ML lifecycle, including experimentation, reproducibility, deployment, and a central model registry. MLflow currently offers four components:

MLflow Tracking

Record and query experiments: code, data, config, and results

Read more

MLflow Projects

Package data science code in a format to reproduce runs on any platform

Read more

MLflow Models

Deploy machine learning models in diverse serving environments

Read more

Model Registry

Store, annotate, discover, and manage models in a central repository

Read more

MLflow Tracking - runs

MLflow Tracking is organized around the concept of **runs** (executions of data science code) which record:

- code version
- start and end time of the run
- source (eg name of the file to launch the run)
- input parameters
- metrics (eg loss)
- artifacts = output files (eg images, models, data files)

Runs can be recorded locally, in a database, remotely (tracking server).

You can set the name of a run:

```
with mlflow.start_run(run_name = "name")
```

or

```
mlflow.set_tag("mlflow.runName", "name")
```

Tracking runs

```python
import mlflow

with mlflow.start_run():
    mlflow.log_param("lr", 0.001)
    # Your ml code
    ...
    mlflow.log_metric("val_loss", val_loss)
```

```python
import mlflow

mlflow.autolog()

# Your training code...
```

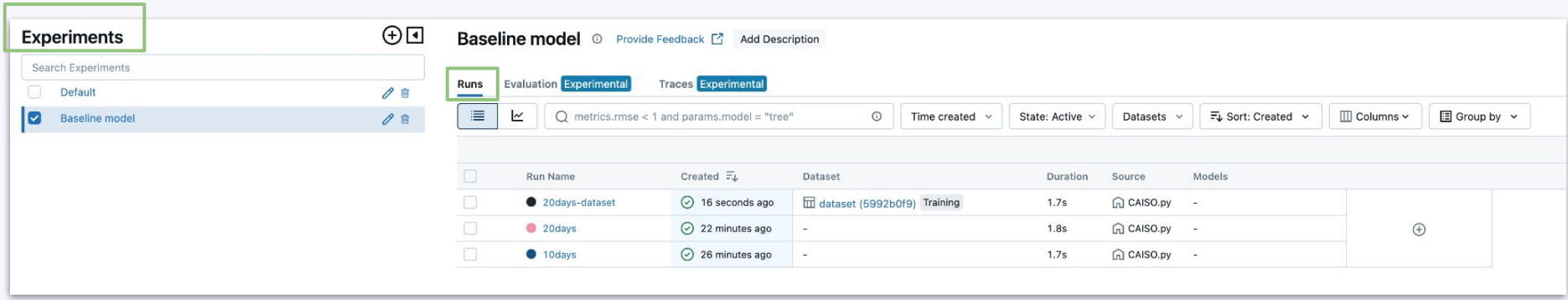
- Initiate an MLflow run context to start a new run that we will log the model and metadata to.
- Log model parameters and performance metrics.

MLflow Tracking - experiment

An **experiment** groups together runs for a specific task.

You can create an experiment using the CLI, API, or UI.

The MLflow API and UI also let you create and search for experiments.



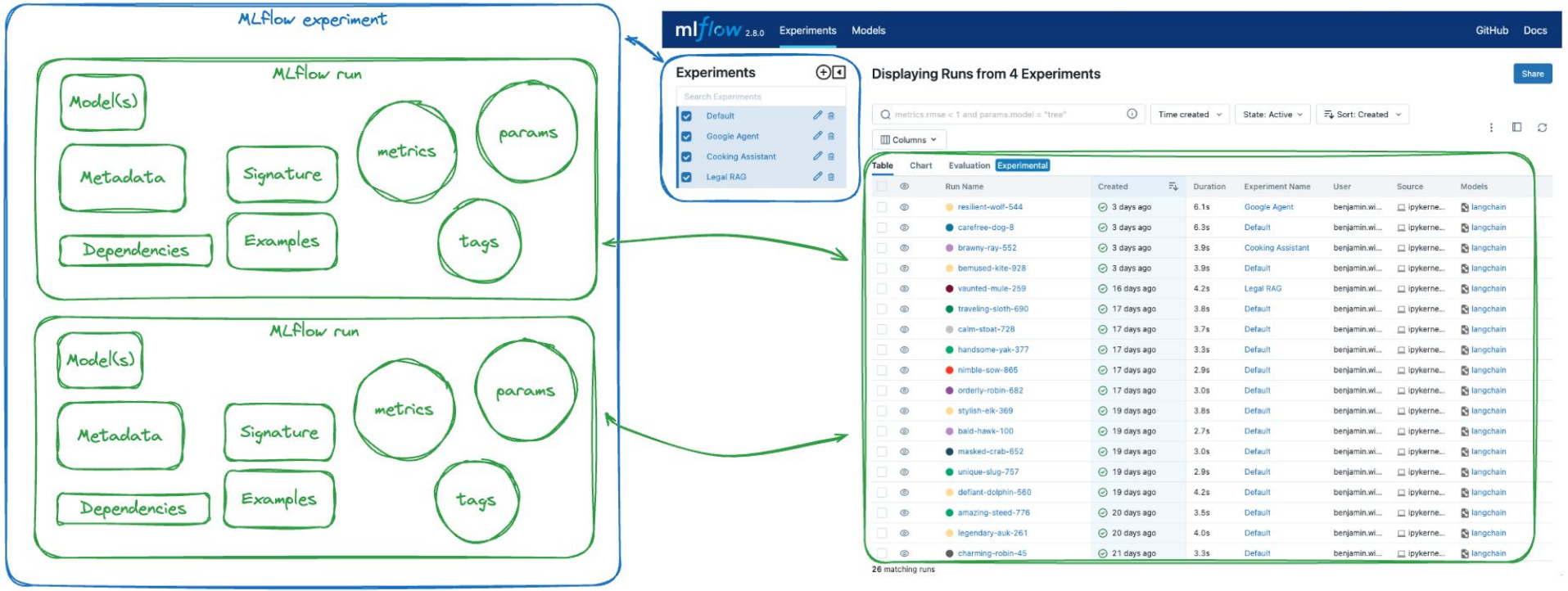
You can set the name of an experiment:

```
mlflow.set_experiment("Baseline model")
```

or

```
mlflow.create_experiment(name, artifact_location="...")
```

MLflow Tracking - overview



MLflow Tracking - overview

1. Install MLflow
   - if using virtual environments you need to install pyenv, instructions here
2. Set a tracking server
   - not strictly necessary: you can simply use local files
   - but useful to test development workflow
   - if not set, MLflow uses by default *http://127.0.0.1:5000*
3. Create/reactivate an experiment and/or a run
   - not strictly necessary: MLflow can generate these
   - if no active run, calling a logging function will start a run
4. Train a model and prepare metadata for logging
   - define model hyperparameters, calculate accuracy etc.
5. Log the model and its metadata
6. View the run in the UI (user interface)

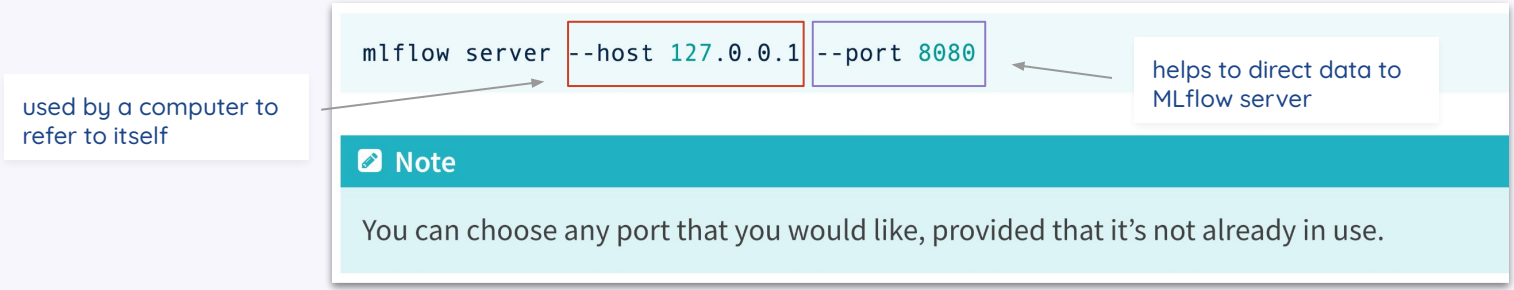
Mlflow tracking server

A **server** is a hardware device or piece of software which manages access to a resource or service in a network.

Examples:

- web server - hosts websites and serves web pages through web browsers
- database server - manages data in a database
- mail server - stores and manages emails
- Mlflow tracking server - tracks ML experiments and serves model metadata

Local tracking server



The host (--host 127.0.0.1) and port (--port 8080) are network settings that determine where the MLflow tracking server runs and how it can be accessed.

127.0.0.1 = **loopback address/localhost:** This IP address is reserved for loopback purposes and is not used to communicate with other devices like a real IP address.

Port = a number that identifies a connection endpoint; it help to direct data to a particular service.

Ports help differentiate multiple services running on the same machine.

Logging

Parameters:

```
dataset_time = "2025-02-13 19:02:14.214940"
mlflow.log_param("dataset_time", dataset_time)
```

Metrics:

```
MAE = sklearn.metrics.mean_absolute_error(test_y, preds)
mlflow.log_metric("MAE", MAE)
print("MAE", MAE)
```

Datasets:

```
dataset = mlflow.data.from_pandas(df)
mlflow.log_input(dataset, context="training")
```

Log artifacts:

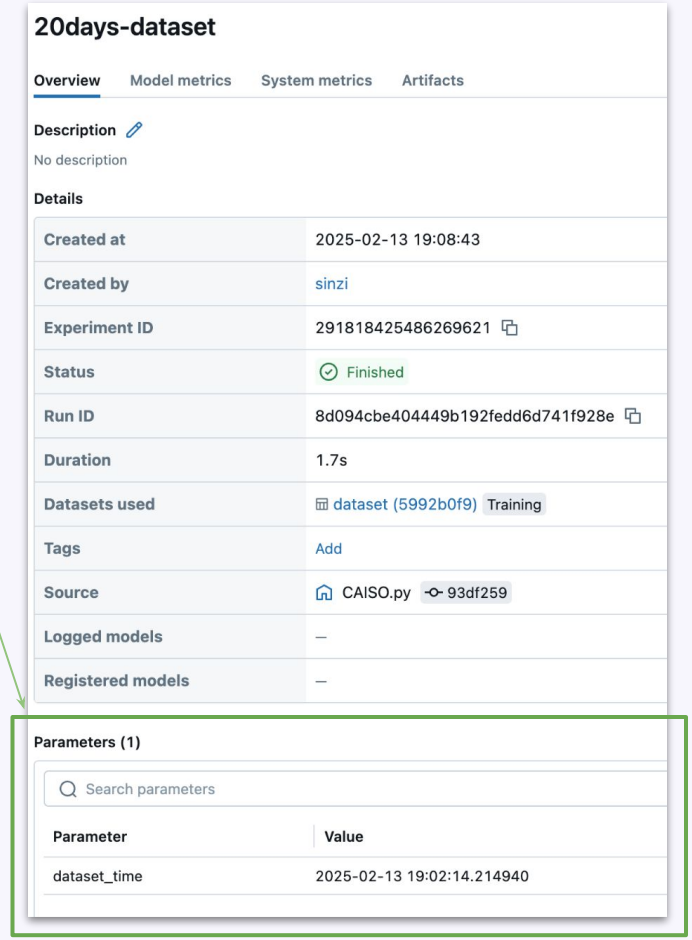
```
df.describe().to_html("dataset-describe.html")
mlflow.log_artifact("dataset-describe.html")
```

Log models:

```
mlflow.pyfunc.log_model(artifact_path="model-updated",python_model=CAISO())
```

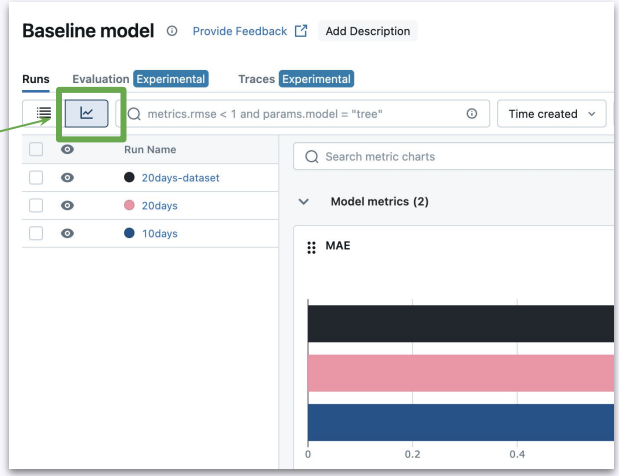
Log parameters

```
dataset_time = "2025-02-13 19:02:14.214940"
mlflow.log_param("dataset_time", dataset_time)
```

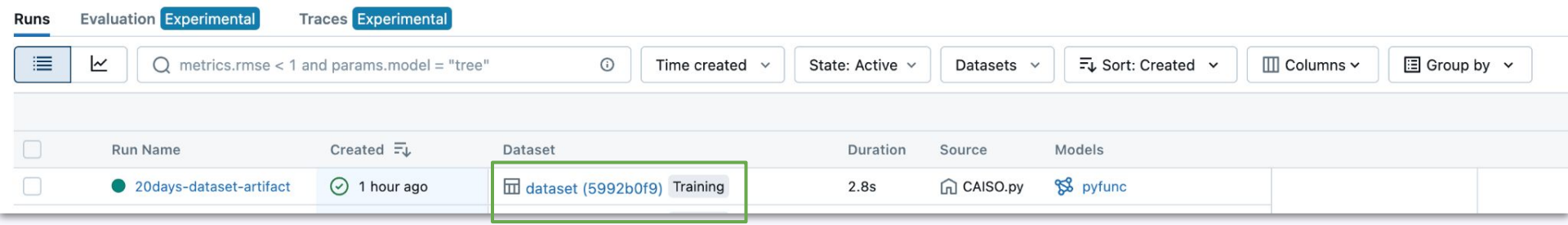


Log metrics

```
MAE = sklearn.metrics.mean_absolute_error(test_y, preds)
mlflow.log_metric("MAE", MAE)
print("MAE", MAE)
```

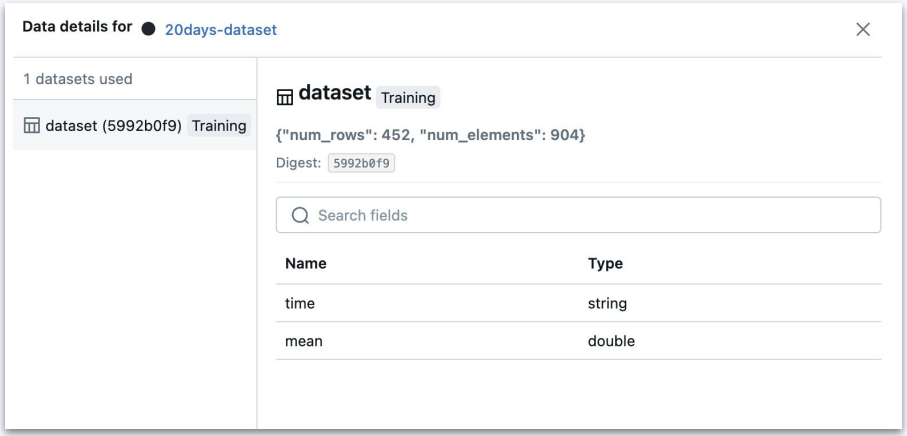


Log datasets




```
dataset = mlflow.data.from_pandas(df)
mlflow.log_input(dataset, context="training")
```

Record and retrieve dataset *information*



In order to load the dataset itself, you need to set a dataset source, which can be a URL or cloud instance, or log the dataset path.

Log artifacts

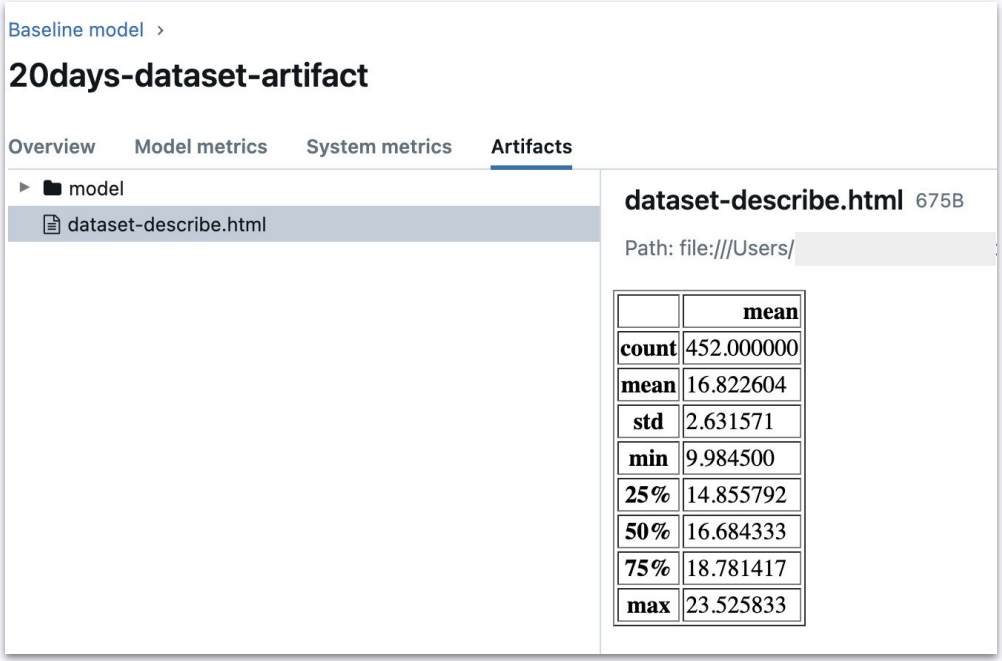
```
df.describe().to_html("dataset-describe.html")
mlflow.log_artifact("dataset-describe.html")
```

Artifacts = output files from a run such as:

- model weights
- images
- datasets

Can be saved either locally or remotely.

When you log a model it appears as an artifact (but not when you save it).



	mean
count	452.000000
mean	16.822604
std	2.631571
min	9.984500
25%	14.855792
50%	16.684333
75%	18.781417
max	23.525833

[Logging Visualizations with MLflow]

Log models

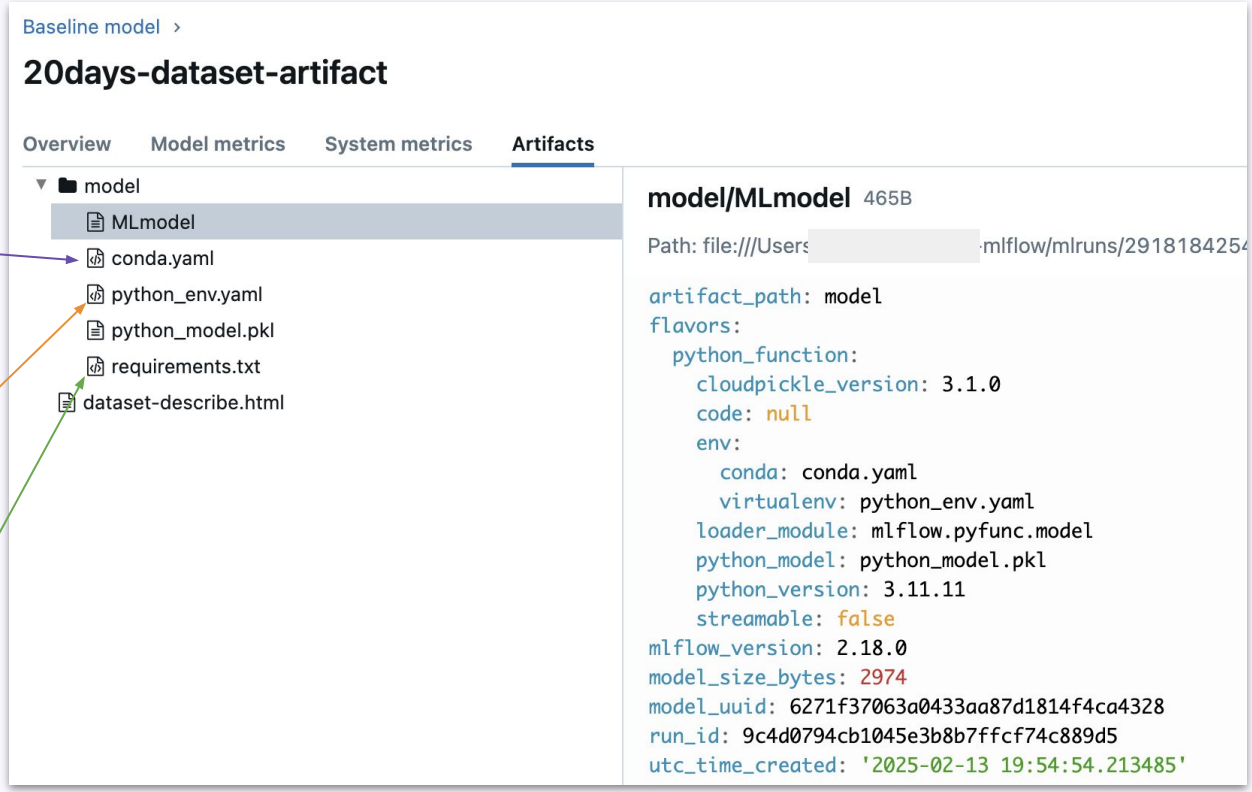
```
mlflow.pyfunc.log_model(artifact_path="model-updated",python_model=CAISO())
```

Dependencies

the conda environment required to run the model

information required to restore the model environment using virtualenv (1) python version (2) build tools like pip (3) ref to requirements

the set of pip dependencies required to run the model

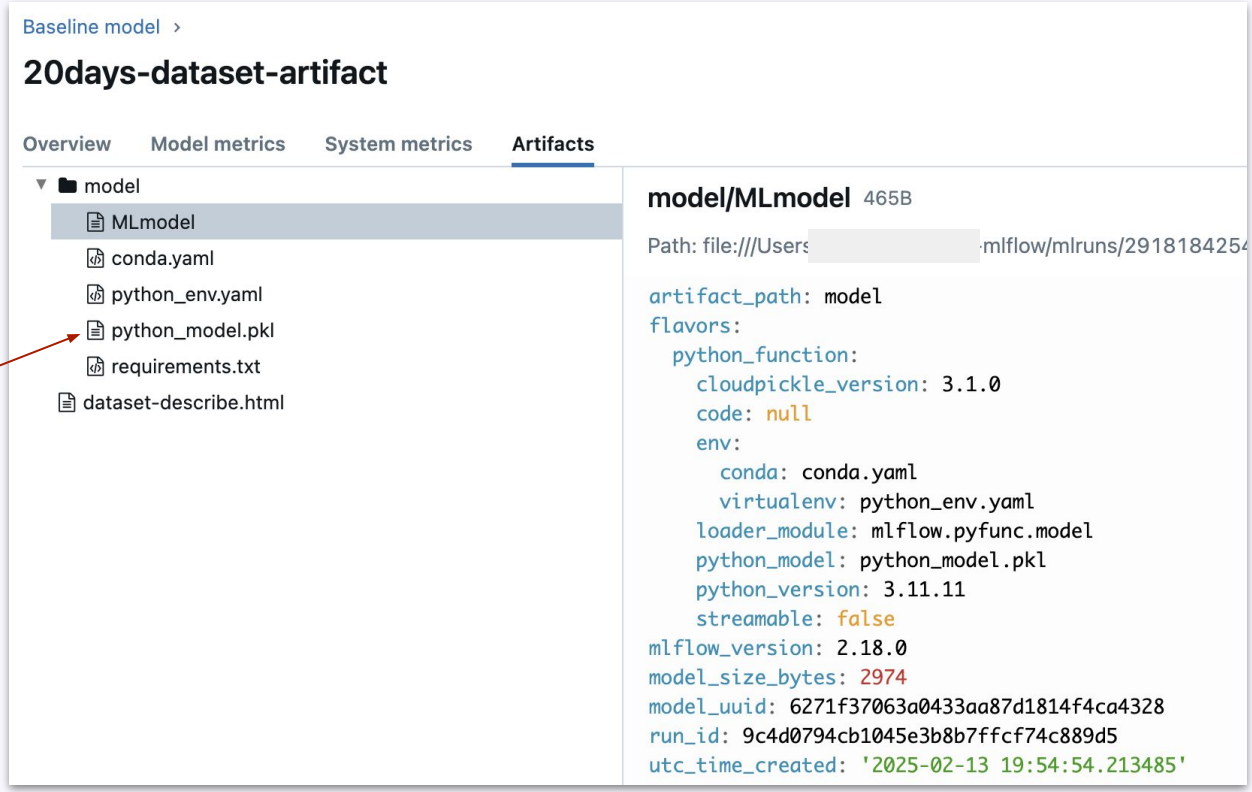


Log models

```
mlflow.pyfunc.log_model(artifact_path="model-updated",python_model=CAISO())
```

Storage

The stored model can either be a *serialized object* (e.g., a pickled scikit-learn model) or a *Python script* (or notebook, if running in Databricks) that contains the model instance.



Log models

```
mlflow.pyfunc.log_model(artifact_path="model-updated",python_model=CAISO())
```

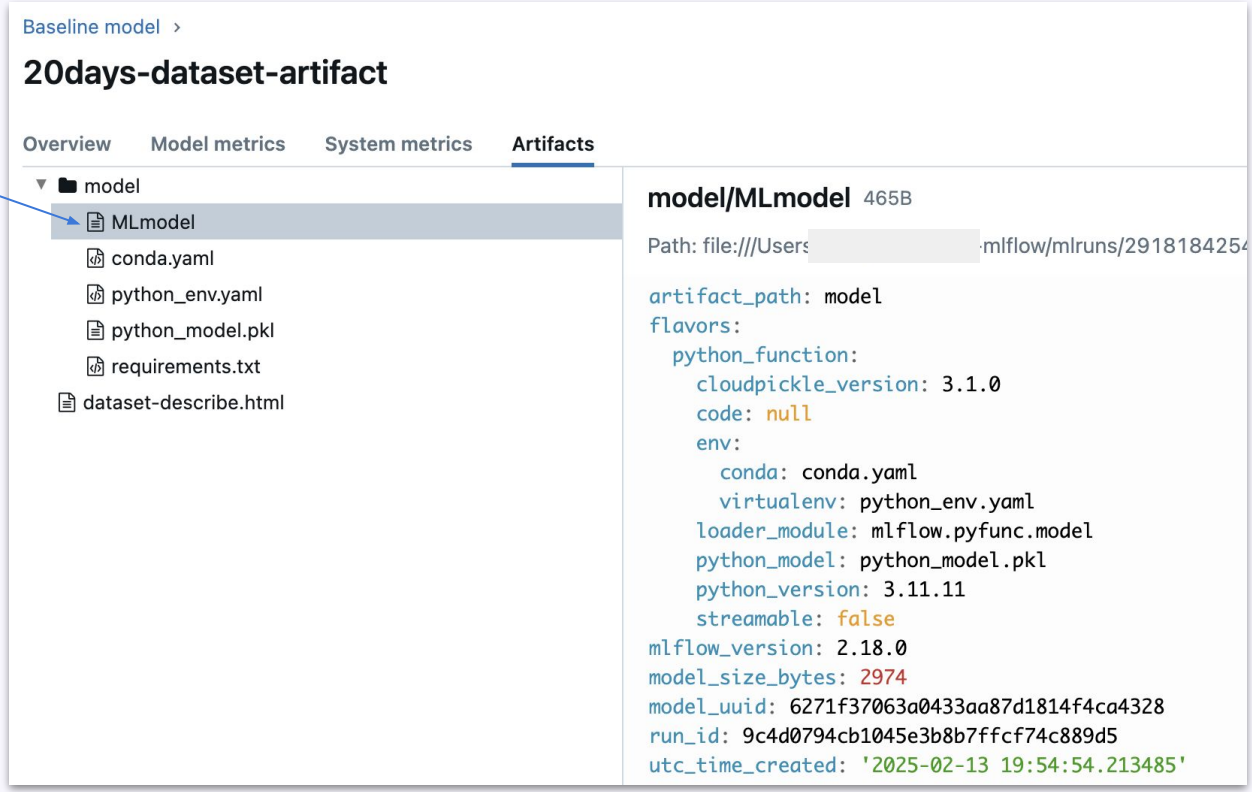
MLmodel file

when the model was created, the ID of the run, mlflow version

signature: expected format for input and output, additional parameters

input example: reference to an artifact with a data example

\+ model *flavours*



MLflow model flavors

= conventions that help understand the model

⇒ allow to write tools that work with models from *any ML library* without having to *integrate* each tool with each library

MLflow model flavors

= conventions that help understand the model

⇒ allow to write tools that work with models from any ML library without having to integrate each tool with each library

Built-In Model Flavors:

- Python Function (python_function)
- Scikit-learn (sklearn)
- XGBoost (xgboost)
- Spark MLlib (spark)
- Keras (keras)
- Transformers (transformers) (experimental)

Using MLflow flavors

Each flavor defines standardized `load_model()`, `predict()` methods regardless of their underlying library.

Make Predictions

Predict on a Pandas DataFrame:

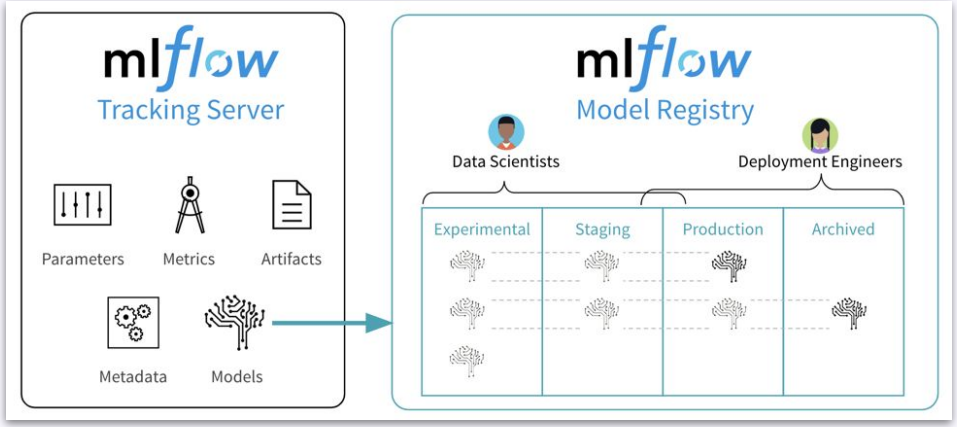
```
import mlflow
logged_model = 'runs:/1eaa1ea9bfc04276be0a5b4256fae4fb/model-updated'

# Load model as a PyFuncModel.
loaded_model = mlflow.pyfunc.load_model(logged_model)

# Predict on a Pandas DataFrame.
import pandas as pd
loaded_model.predict(pd.DataFrame(data))
```

Sklearn example

```
import mlflow
from mlflow.models import infer_signature
import numpy as np
from sklearn.linear_model import LogisticRegression

with mlflow.start_run():
    X = np.array([-2, -1, 0, 1, 2, 1]).reshape(-1, 1)
    y = np.array([0, 0, 1, 1, 1, 0])
    lr = LogisticRegression()
    lr.fit(X, y)
    signature = infer_signature(X, lr.predict(X))

    model_info = mlflow.sklearn.log_model(
        sk_model=lr, artifact_path="model", signature=signature
    )

sklearn_pyfunc = mlflow.pyfunc.load_model(model_uri=model_info.model_uri)

data = np.array([-4, 1, 0, 10, -2, 1]).reshape(-1, 1)

predictions = sklearn_pyfunc.predict(data)
```

MLflow Registry

You can register MLflow Models into the MLflow Registry = a centralized **model store** that provides a UI and set of APIs to manage the full lifecycle of MLflow Models.



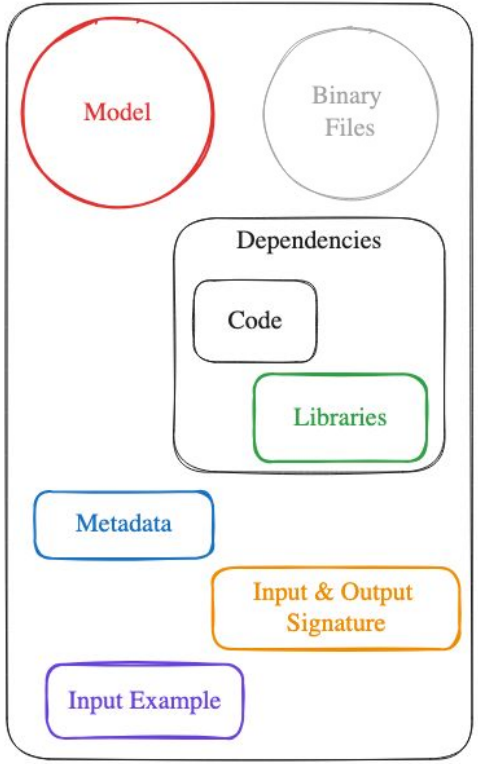
Register model

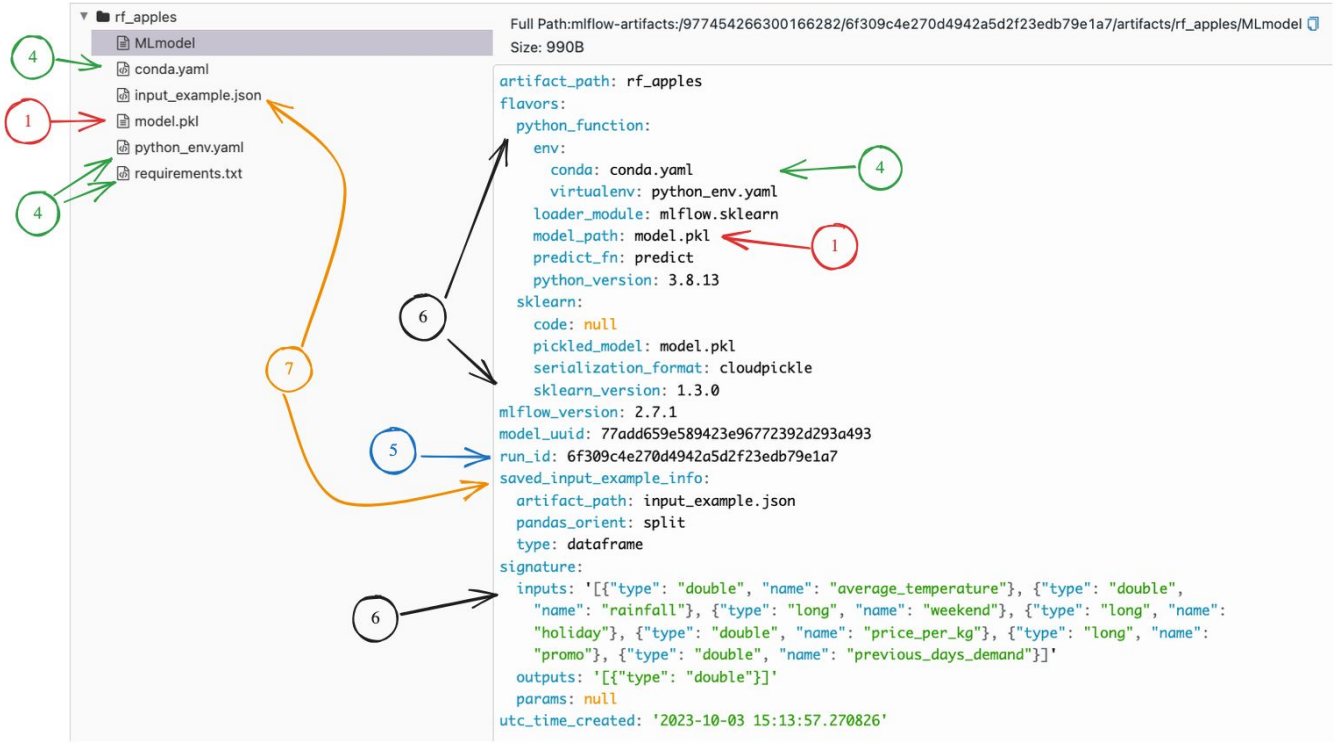
```
# Log the sklearn model and register as version 1
mlflow.sklearn.log_model(
    sk_model=model,
    artifact_path="sklearn-model",
    signature=signature,
    registered_model_name="sk-learn-random-forest-reg-model",
)
```

Load model

```
import mlflow.pyfunc

model_name = "sk-learn-random-forest-reg-model"
model_version = 1

model = mlflow.pyfunc.load_model(model_uri=f"models:/{model_name}/{model_version}")

model.predict(data)
```

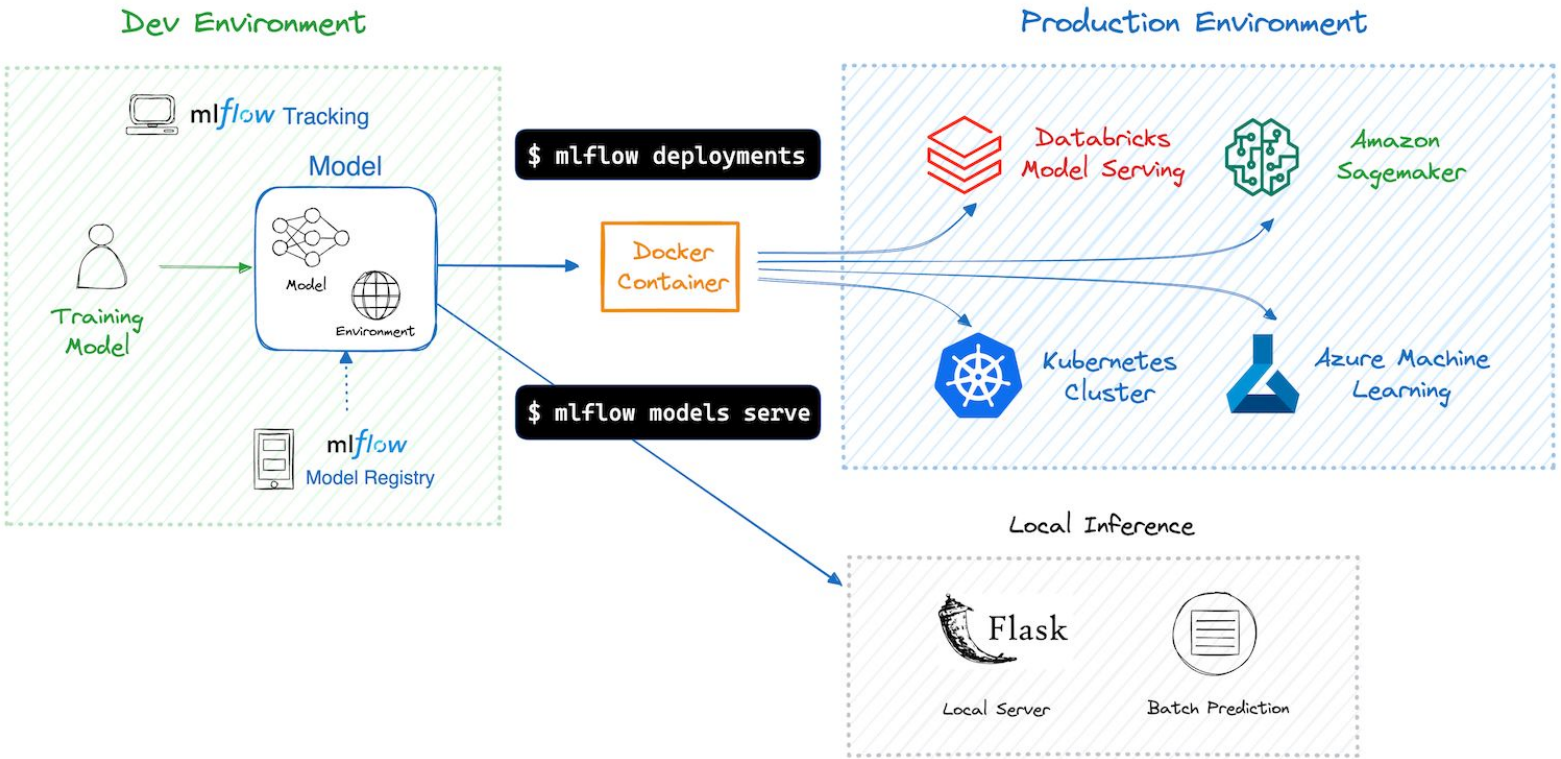
Overview of MLflow Model





Deployment

After training your machine learning model and ensuring its performance, the next step is to make the models available.



Serving and deployment an MLflow model

Serving is the process of making a model accessible - for example with a REST API or a web service.

Representational State Transfer (REST) is a software architecture that imposes conditions on how an API should work (statelessness, cacheability etc).

An application programming interface (API) is a connection between computers or between computer programs. It is a type of software interface, offering a service to other pieces of software.

Serving a MLflow Model

To serve the model to a local REST API endpoint, run the following command:

```
mlflow models serve -m model
```

starts a local server and listens on the specified port

To use the model, we send data (in json format) to the server running MLFlow:

endpoint: a URL where you want to send data to or retrieve data from

-H = header of the request: data being sent is in json format

```
curl http://127.0.0.1:5000/invocations -H 'Content-Type: application/json'
-d '{"dataframe_records":[{"Time":"2025-01-10T15:0:00Z"}]}'
```

data to run inference on

cURL = client URL: open-source command line tool for exchanging data with a server

Serving a MLflow Model

To serve the model to a local REST API endpoint run the following command:

```
mlflow models serve -m model
```

To use the model, we send data (in json format) to the server running MLFlow:

```
curl http://127.0.0.1:5000/invocations -H 'Content-Type: application/json'
-d '{"dataframe_records":[{"Time":"2025-01-10T15:0:00Z"}]}'
```

This approach is ideal for lightweight applications or for testing your model locally before moving it to a staging or production environment.

Serving and deployment an MLflow model

Serving is the process of making of making predictions available to end users or applications (for example with REST API or web service).

Deploying is the overall process of integrating a trained machine learning model into a production environment.

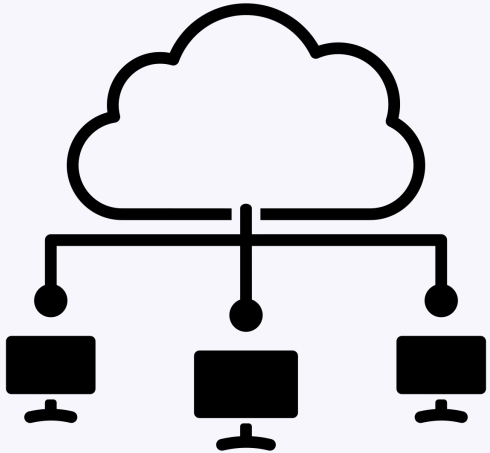
Goal: To get the model into an environment where it's ready to start serving predictions, but not necessarily actively doing so.

Deployment target refers to the destination environment for your model. MLflow supports various targets, including local environments, cloud services (AWS, Azure), Kubernetes clusters, and others.

Deploying a MLflow Model to a remote server

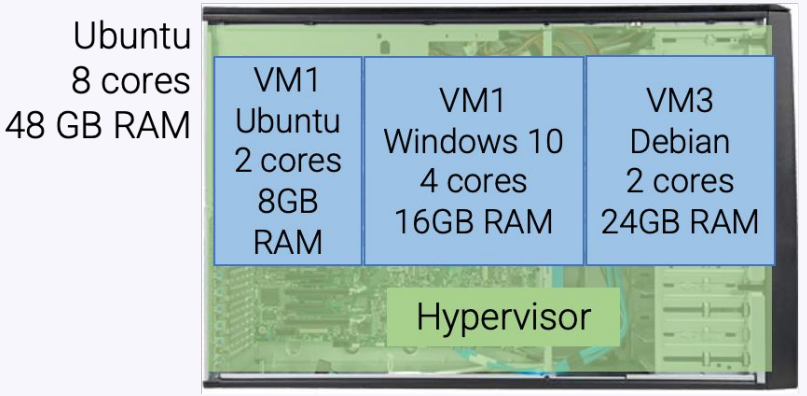
Deploying is the overall process of integrating a trained machine learning model into an environment accessible across teams and users.

For example: you can deploy your model to a cloud-based virtual machine with a public IP address.



Virtual machines

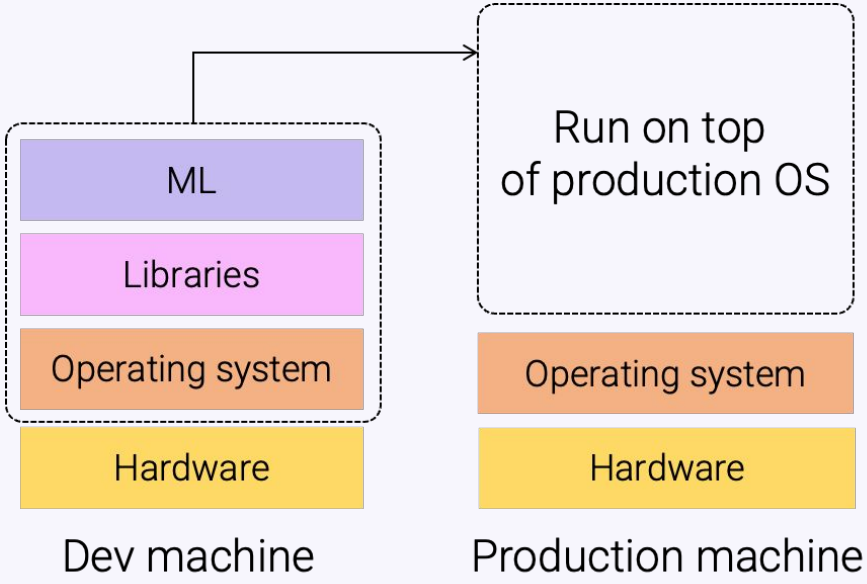
- Traditionally, one operating systems is installed on one piece of hardware
- Virtualization allows to run multiple OS on the same hardware
- Virtual machine is an emulation of a computer system (OS, hardware)



Hypervisor (or Virtual Machine Monitor) = software layer that runs the virtual machines

Virtualization

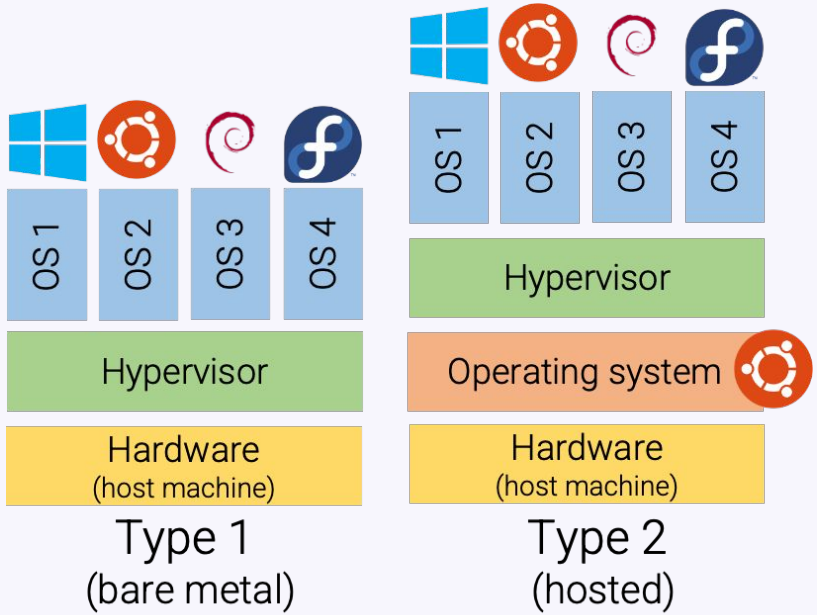
- Virtualization: “freeze” the whole system (including the OS) into an isolated “package”
- Run the whole package on top of the operating system of the production machine



Virtual machines

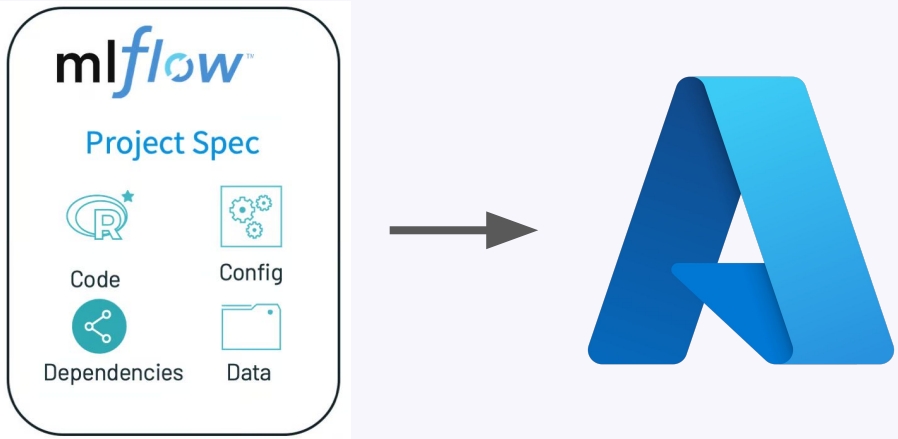
- The Hypervisor splits the hardware resources of the machine across multiple virtual machines

- Hypervisor can be installed directly on Hardware (Type 1) or on host OS (Type 2)



MLflow deployment with VMs

For the assignment you will deploy your model using MLflow projects and a virtual machine on Azure.



MLflow Projects

An MLflow Project = a convention for organizing and describing code to let others run it.

Each project is a directory of files, or a Git repository, containing code and it can specify several properties:

- name of the project
- entry points (commands that can be run within the project, and information about their parameters)
- environment (conda, virtualenv, docker containers)

MLproject file - adds more details about how the project can be ran.

MLproject file

caiso example from the exercises

```
name: CAISO_model

python_env: python_env.yaml

entry_points:
  main:
    command: "python CAISO.py"
```

a more complex example

```
name: My Project

python_env: python_env.yaml
# or
# conda_env: my_env.yaml
# or
# docker_env:
#    image:  mlflow-docker-example

entry_points:
  main:
    parameters:
      data_file: path
      regularization: {type: float, default: 0.1}
    command: "python train.py -r {regularization}
{data_file}"
```

Running MLflow projects

There are two ways to run a project:

```
mlflow run project_URI
```

```
mlflow.projects.run(project_URI)
```

MLflow deployment with VMs on Azure

Push a locally developed MLflow project to Github.

Create a VM on Azure and ssh into it.

Install Python, pip, libssl-dev, pyenv. Reload shell.

Create a virtual environment, where you install MLflow.

Clone the GitHub repository containing your model.

Run and then serve the model.

Query the served model from your machine using the VM's public IP address.

MLflow overview

MLflow provides a unified platform to navigate model development, deployment, and management.

Experiment Management

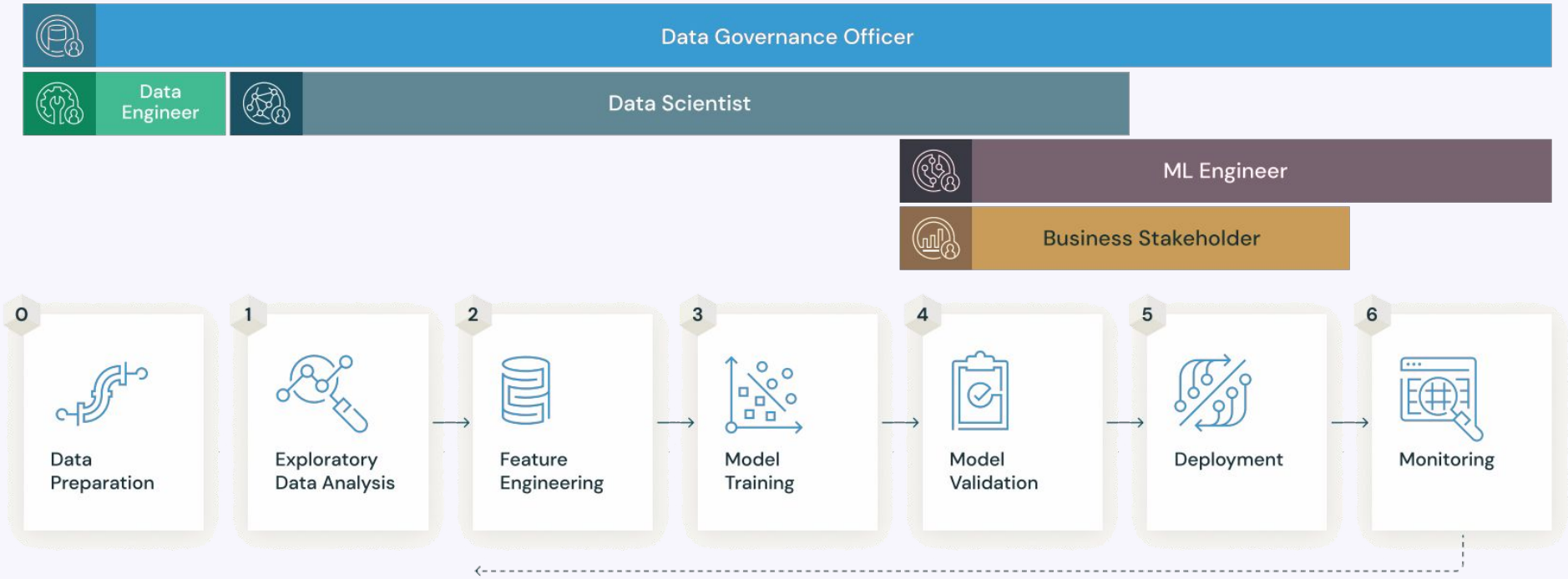
Reproducibility

Deployment Consistency

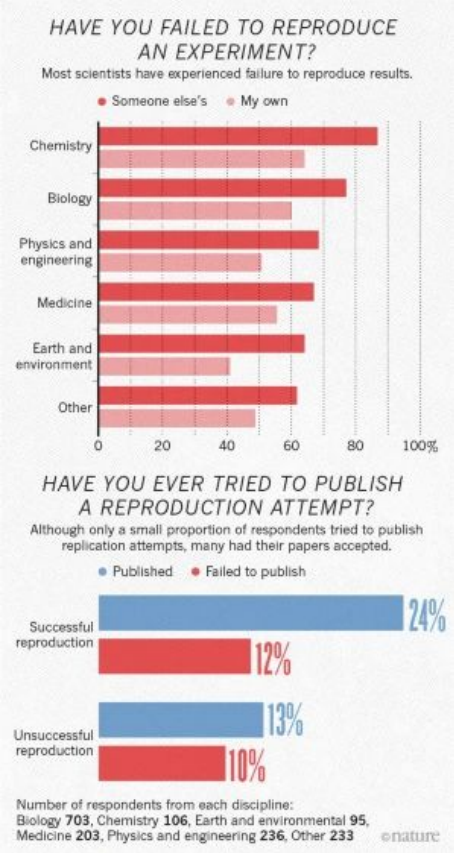
Model Management

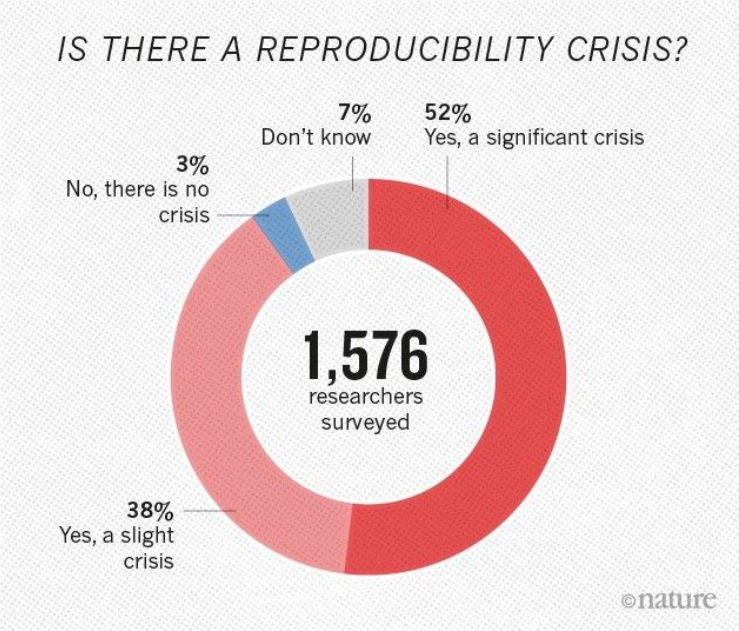
Library Agnosticism

ML WORKFLOW AND PERSONAS



Reproducibility crisis in science





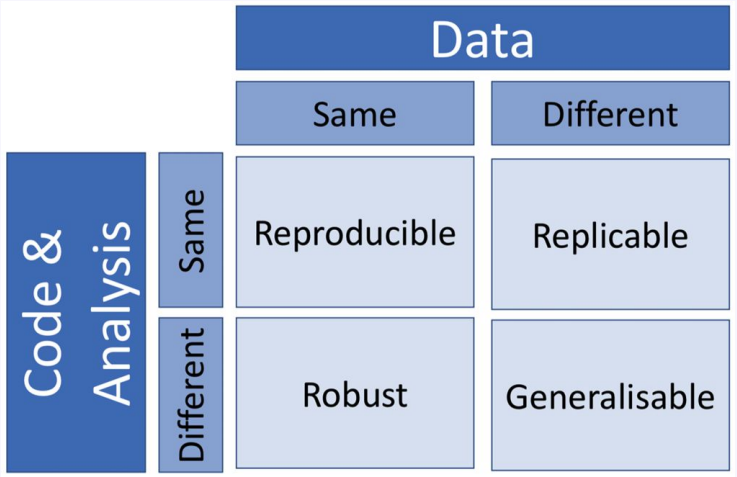
Link to article

Reproducibility gap in ML research

- Underspecification of the **metrics** used to report results
- Improper use of **statistics** to analyze results
- **Selective** reporting of results
- Lack of access to the same **training data** / differences in data distribution
- Lack of availability of the **code** necessary to run the experiments, or errors in the code

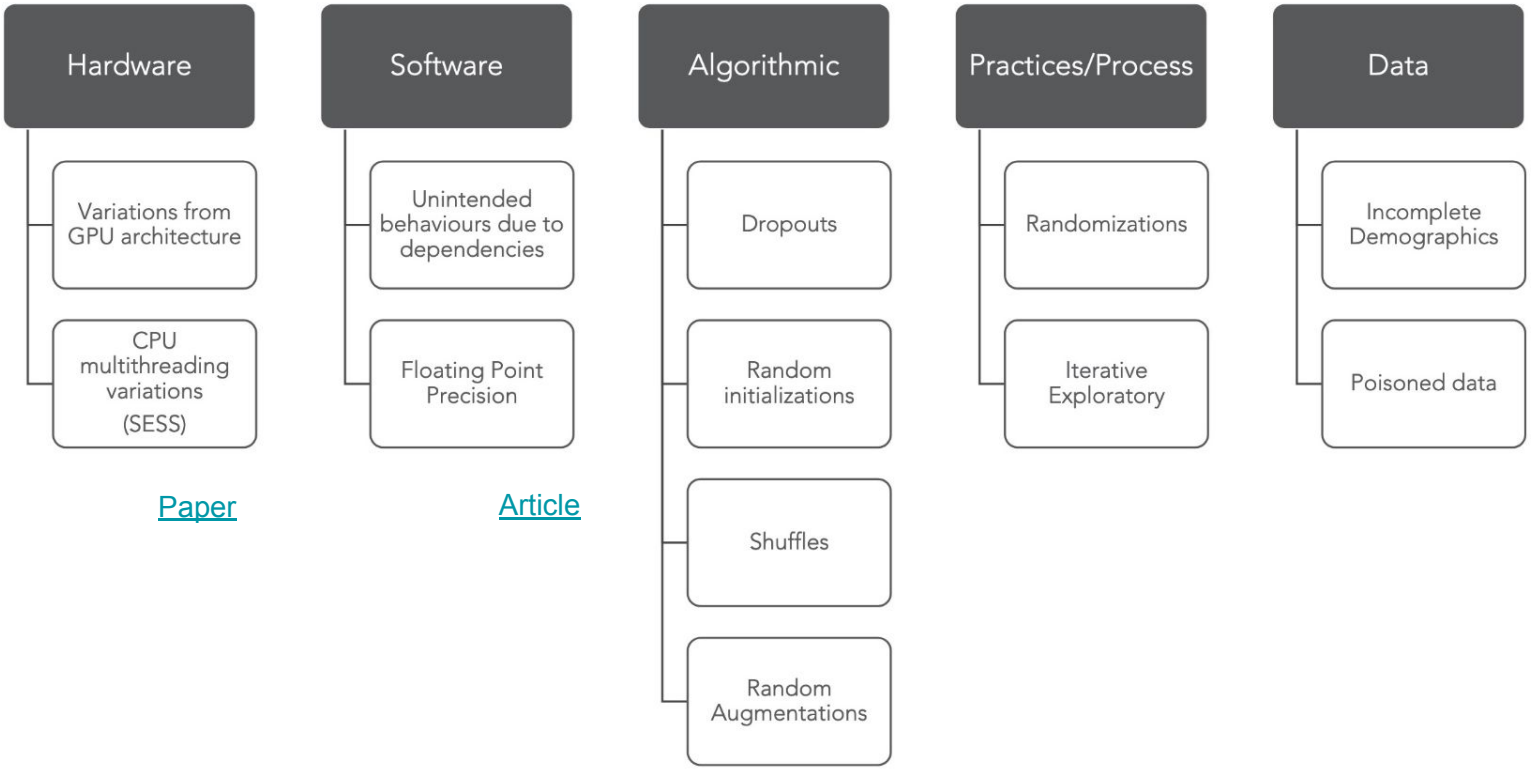
Reproducibility gap in ML research

- Underspecification of the **metrics** used to report results
- Improper use of **statistics** to analyze results
- **Selective** reporting of results
- Lack of access to the same **training data** / differences in data distribution
- Lack of availability of the **code** necessary to run the experiments, or errors in the code



Training the same model, with the same hyperparameters can give different results.

CHALLENGES IN REPRODUCIBLE AI/ML

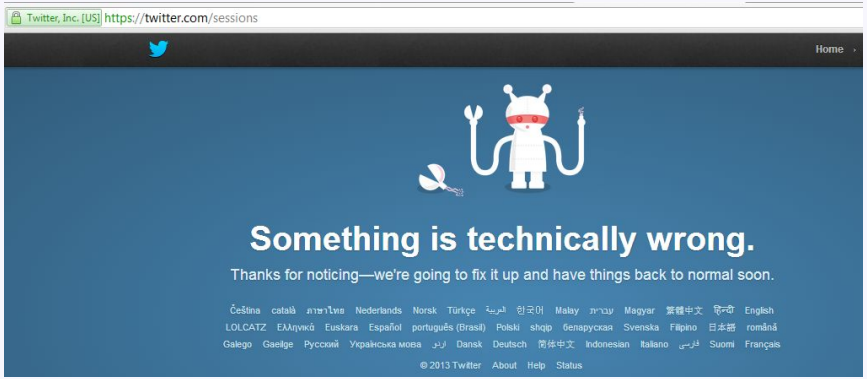


Paper

Article

ML systems fail silently

Normal software fails



ML systems fails

Monitoring ML systems

Monitoring = the act of tracking, measuring, and logging different metrics that can help us determine when something goes wrong.

Monitoring ML systems

Monitoring = the act of tracking, measuring, and logging different metrics that can help us determine when something goes wrong.

Metrics that convey the health of an ML system:

- operational metrics (latency, throughput, the number of prediction requests, CPU/GPU utilization, memory utilization etc)

Monitoring ML systems

Monitoring = the act of tracking, measuring, and logging different metrics that can help us determine when something goes wrong.

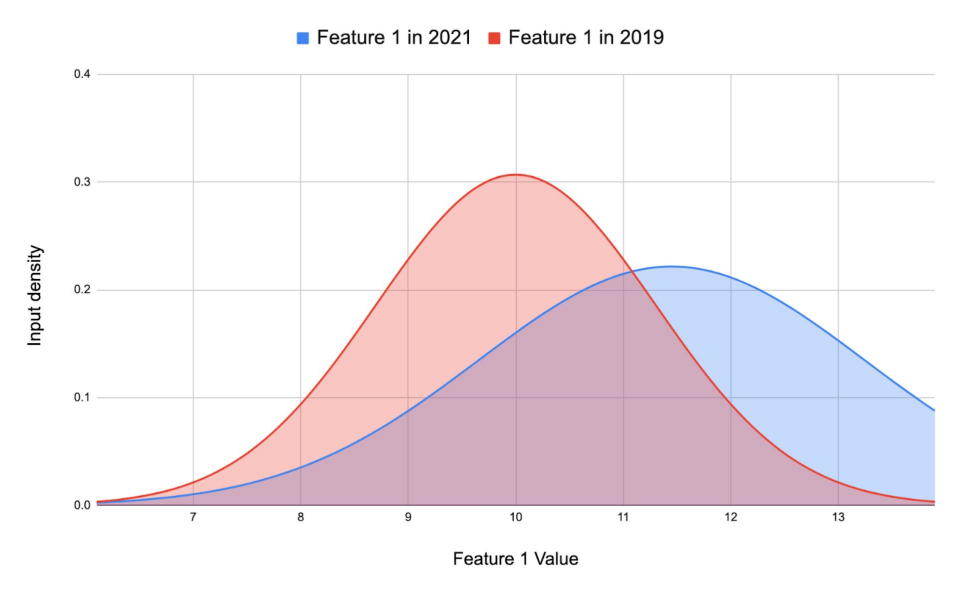
Metrics that convey the health of an ML system:

- operational metrics (latency, throughput, the number of prediction requests, CPU/GPU utilization, memory utilization etc)
- ML specific metrics - 4 artifacts:
  - accuracy related (eg user feedback)
  - predictions
  - features
  - raw inputs

How ML Breaks: A Decade of Outages for One Large ML Pipeline

Data distribution shift

= the data a model works with changes over time



Data distribution shift

Source distribution = data the model is trained on

Target distribution = data the model runs inference on

Data distribution shift

X = inputs to a model

Y = outputs

In supervised learning, the training data can be viewed as a set of samples from the joint distribution P(X, Y). We model P(Y|X)

The joint distribution P(X, Y) can be decomposed in two ways:

$P(X, Y) = P(Y|X)P(X)$

$P(X, Y) = P(X|Y)P(Y)$

Data distribution shift

Covariate shift - P(X) changes, but P(Y|X) remains the same

= the distribution of the input changes, but the conditional probability of a label given an input remains the same

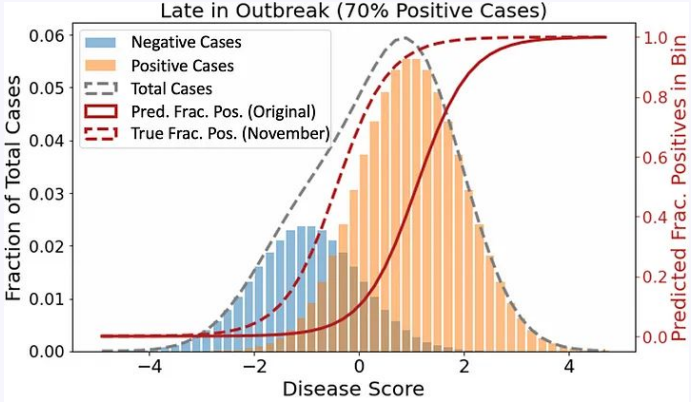
Data distribution shift

Covariate shift - P(X) changes, but P(Y|X) remains the same

= the distribution of the input changes, but the conditional probability of a label given an input remains the same

Label shift (prior shift) - P(Y) changes but P(X|Y) remains the same

= the output distribution changes, but for a given output, the input distribution stays the same



Data distribution shift

Covariate shift - P(X) changes, but P(Y|X) remains the same

= the distribution of the input changes, but the conditional probability of a label given an input remains the same

Label shift (prior shift) - P(Y) changes but P(X|Y) remains the same

= the output distribution changes but for a given output, the input distribution stays the same

Concept drift (posterior shift) - P(Y|X) changes, P(X) remains the same

= same input, different output

Detecting data distribution shifts

How to determine that two distributions are different?

1. Compare statistics: mean, median, variance, quantiles, skewness, …
    - Compute these stats during training and compare these stats in production

Detecting data distribution shifts

How to determine that two distributions are different?

1. Compare statistics: mean, median, variance, quantiles, skewness, …
    - Not universal: only useful for distributions where these statistics are meaningful

Detecting data distribution shifts

How to determine that two distributions are different?

1. Compare statistics: mean, median, variance, quantiles, skewness, …
    - Not universal: only useful for distributions where these statistics are meaningful
    - Inconclusive: if statistics differ, distributions differ. If statistics are the same, distributions can still differ.

Detecting data distribution shifts

How to determine that two distributions are different?

1. Compare statistics: mean, median, variance, quantiles, skewness, ...
2. Two-sample hypothesis test
   - Determine whether the difference between two populations is statistically significant
   - If yes, likely from two distinct distributions

E.g.
1. Data from yesterday
2. Data from today

Two-sample test: KS test (Kolmogorov–Smirnov)

- Pros
  - Doesn’t require any parameters of the underlying distribution
  - Doesn’t make assumptions about distributions
- Cons
  - Only works with one-dimensional data

- Useful for prediction & label distributions
- Not so useful for features

Two-sample test

Drift Detection

Detector	Tabular	Image	Time Series	Text	Categorical Features	Online	Feature Level
Kolmogorov-Smirnov	✓	✓		✓	✓		✓
Maximum Mean Discrepancy	✓	✓		✓	✓	✓	
Learned Kernel MMD	✓	✓		✓	✓		
Least-Squares Density Difference	✓	✓		✓	✓	✓	
Chi-Squared	✓				✓		✓
Mixed-type tabular data	✓				✓		✓
Classifier	✓	✓	✓	✓	✓		
Spot-the-diff	✓	✓	✓	✓	✓		✓
Classifier Uncertainty	✓	✓	✓	✓	✓		
Regressor Uncertainty	✓	✓	✓	✓	✓		

alibi-detect

Most tests work better on low-dim data, so dim reduction is recommended beforehand!

ML in production: expectation

1. Collect data
2. Train model
3. Deploy model
4. 

ML in production: reality

1. Choose a metric to optimize
2. Collect data
3. Train model
4. Realize many labels are wrong -> relabel data
5. Train model
6. Model performs poorly on one class -> collect more data for that class
7. Train model
8. Model performs poorly on most recent data -> collect more recent data
9. Train model
10. Deploy model
11. Dream about $$$
12. Wake up at 2am to complaints that model biases against one group -> revert to older version
13. Get more data, train more, do more testing
14. Deploy model
15. Pray
16. Model performs well but revenue decreasing
17. Cry
18. Choose a different metric
19. Start over

Value of data

Integrating ML/AI models in production is difficult:

- **only one in two** organizations has moved beyond pilots and proofs of concept
- **72%** of a cohort of organizations that began AI pilots before 2019 have not been able to deploy even a single application in production

Issues:

- teams engage in a *high degree of manual* and one-off work
- no *reusable or reproducible* components
- processes involve difficulties in *handoffs between data scientists and IT*
- challenges in *deployment, scaling, and versioning* efforts still hinder teams from getting value from their investments in ML

ML systems

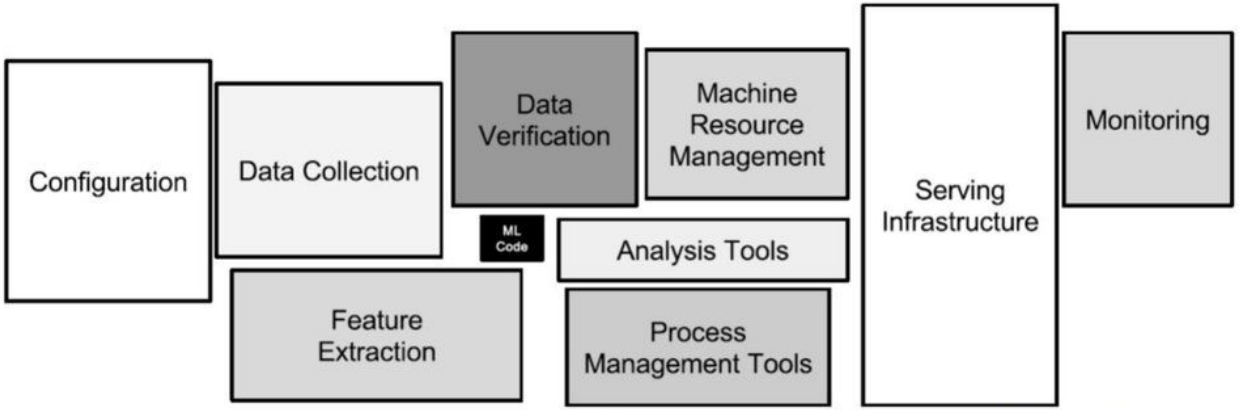
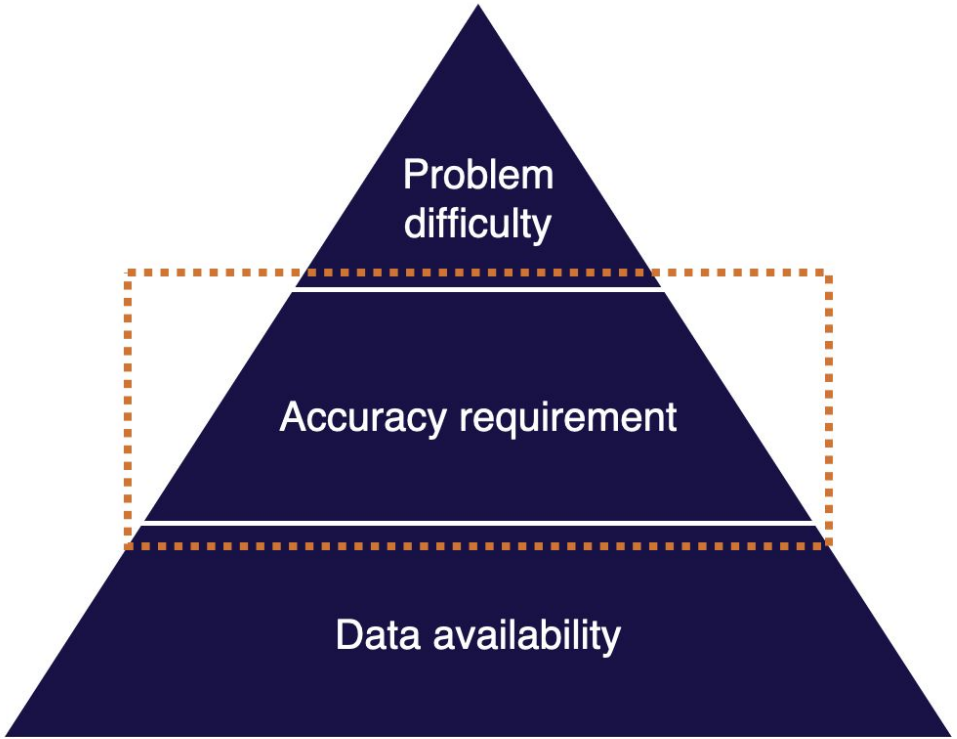


Figure 1: Only a small fraction of real-world ML systems is composed of the ML code, as shown by the small black box in the middle. The required surrounding infrastructure is vast and complex.

Before designing an ML system:

Cost drivers



Main considerations

- Is the problem well-defined?
- Good published work on similar problems? (newer problems mean more risk & more technical effort)
- Compute requirements?
- Can a human do it?

- How costly are wrong predictions?
- How frequently does the system need to be right to be useful?
- Ethical implications?

- How hard is it to acquire data?
- How expensive is data labeling?
- How much data will be needed?
- How stable is the data?
- Data security requirements?