



2.3 Reproducibility in MLOps

POSGRAD
MNA

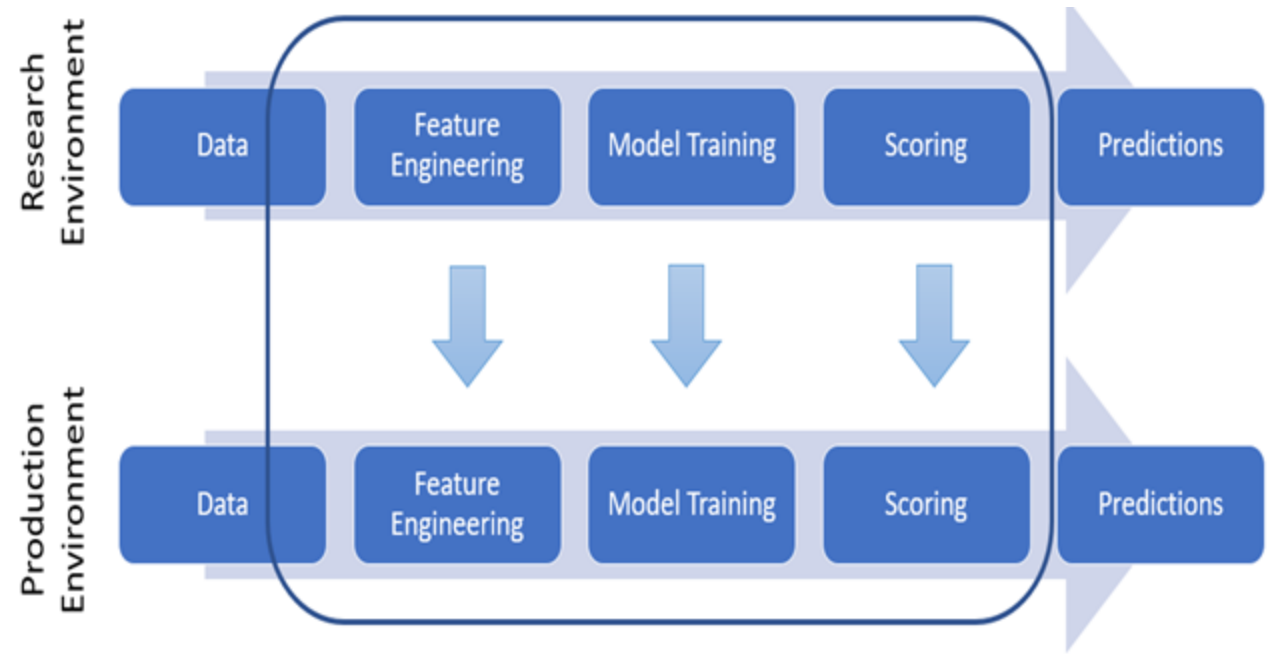
Introduction




Reproducibility is the capacity to exactly replicate the results of a process or model such that, when provided with the identical raw data input, the duplicated process or model produces the same output.

If the model cannot be reliably reproduced, users can lose trust in the results.


In the process of constructing a machine learning (ML) model, we operate across several distinct environments, usually encompassing research/development, and production environments.





Beyond the initial research, each stage aims to mimic the established set of processes, but the methods to achieve this vary significantly between environments.

This variation in methods introduced numerous challenges to maintaining reproducibility at every stage of the production line, from data collection to generating predictions with the deployed model.



A man with a beard and glasses, wearing large white headphones, is seated at a desk. He is looking towards the right side of the frame. He is wearing a blue button-down shirt. In the background, there is a bookshelf filled with books. The image has a teal overlay with abstract geometric shapes in the corners and a pattern of concentric circles on the right side.

Challenges of Reproducibility in ML

Data & Input — Reproducibility Challenges

- Data drift or change over time affecting model reproducibility.
- Lack of data versioning and provenance tracking.
- Data leakage between training and testing datasets.
- Preprocessing inconsistency across different experiments.

Data & Input — Mitigation Strategies

- Version and snapshot datasets using tools like DVC or Delta Lake.
- Freeze and serialize preprocessing pipelines to ensure consistency.
- Use strict data partitioning logic and schema validation.
- Maintain lineage metadata and perform regular data validation checks.

Feature Engineering — Reproducibility Challenges

- Non-deterministic transformations introducing randomness.
- Inconsistent feature transformation logic between training and inference.
- Library version drift altering preprocessing behavior.

Feature Engineering — Mitigation Strategies

- Fix random seeds and ensure deterministic feature transformations.
- Serialize and reuse trained feature transformers (e.g. joblib artifacts).
- Lock dependency versions and containerize preprocessing environments.
- Implement tests to confirm transformation parity between training and inference.

Model Training — Reproducibility Challenges

- Random initialization and stochastic optimization yield non-deterministic results.
- Non-repeatable hyperparameter search processes.
- Library or algorithmic updates alter model performance.
- Hardware and floating-point nondeterminism (especially on GPUs).
- Overfitting to validation or test data through repeated tuning.

Model Training — Mitigation Strategies

- Fix random seeds across all frameworks (NumPy, TensorFlow, PyTorch).
- Enable deterministic training modes when available.
- Log hyperparameters and results in experiment tracking tools (e.g., MLflow).
- Containerize environments to preserve library and hardware consistency.
- Use multiple runs and report statistical variance across experiments.

Model Evaluation — Reproducibility Challenges

- Differences in metric implementations or thresholds.
- Cross-validation variability due to random splits.
- Selective reporting and metric cherry-picking.

Model Evaluation — Mitigation Strategies

- Use standardized, verified metric libraries (e.g., `sklearn.metrics`).
- Fix evaluation splits and random seeds for reproducibility.
- Save predictions and confusion matrices for auditing.
- Report averaged results across multiple runs rather than single best outcomes.

Deployment & Inference — Reproducibility Challenges

- Training-serving skew: mismatch in preprocessing logic.
- Model version drift leading to outdated deployments.
- Hardware/environment differences causing inference variability.
- Lack of inference logging and monitoring.
- Concept drift in data after deployment.

Deployment & Inference — Mitigation Strategies

- Embed preprocessing logic within deployed models or reuse transformers.
- Use a model registry to control deployment versions.
- Containerize inference environments (Docker, ONNX Runtime).
- Implement full audit logging of inputs, outputs, and model versions.
- Monitor data drift and trigger retraining pipelines as needed.

Maintenance & Evolution — Reproducibility Challenges

- Untracked modifications causing experiment irreproducibility.
- Missing regression testing for new versions of code or models.
- Feature schema drift breaking backward compatibility.
- Loss of institutional knowledge and design rationale.

Maintenance & Evolution — Mitigation Strategies

- Introduce ML regression tests in CI/CD pipelines.
- Track infrastructure and environments as code (i.e. Dockerfiles).
- Maintain comprehensive documentation and model information.
- Use version control for all artifacts and schema validation over time.

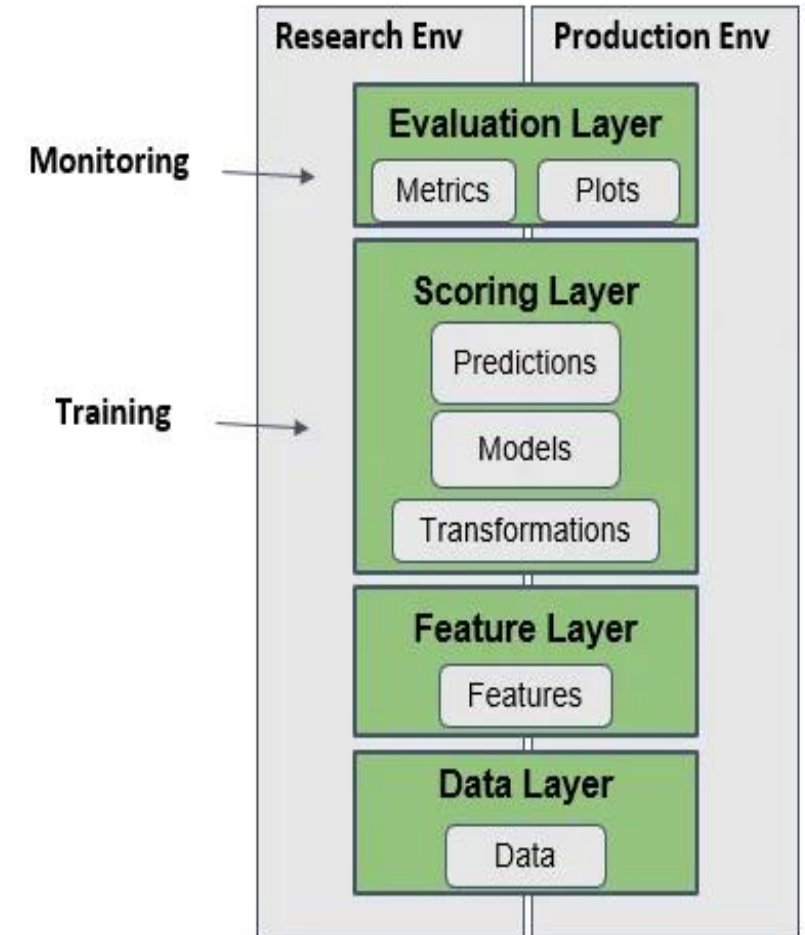


Architecture for reproducible ML Pipelines.

Architecture for Reproducibility

Adopting modularity by dividing the project into components enhances generality and scalability. The architecture will align with Scikit-learn's API conventions for compatibility, recognized as industry standards.

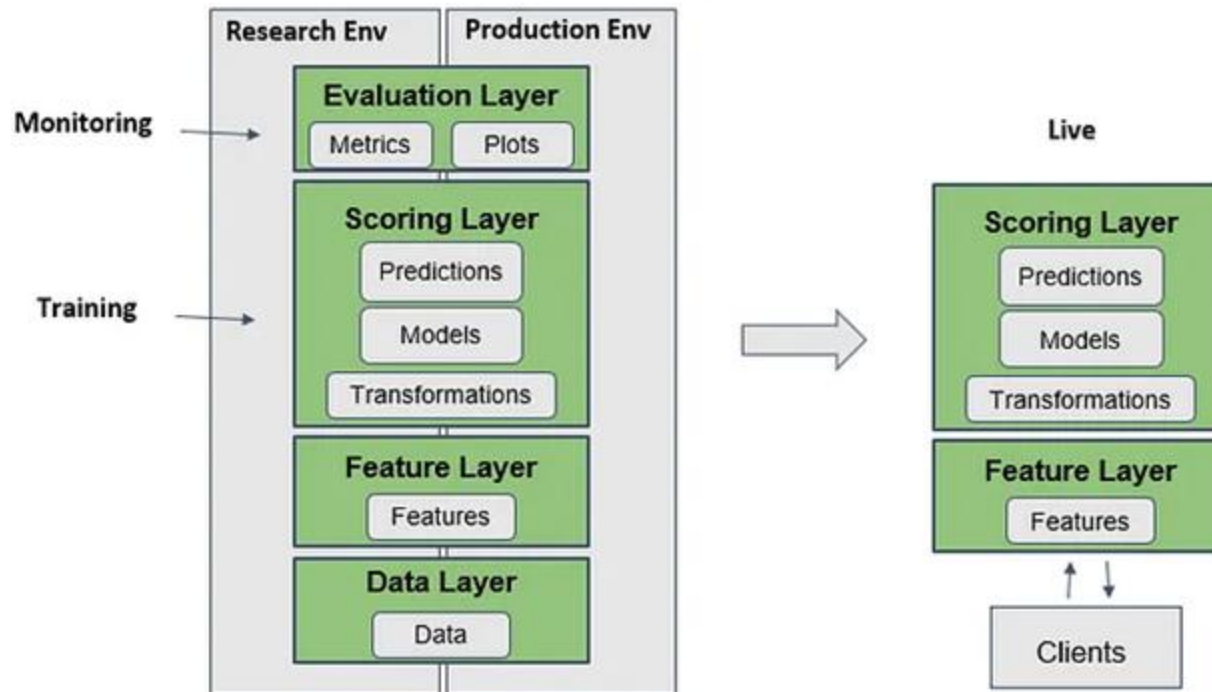
- Data Layer
- Feature Layer
- Scoring Layer
- Evaluation Layer




Assessment for Reproducibility

Ensuring models in both the production and research environments yield identical outputs when fed the same inputs is crucial for validating a system's reproducibility.

If discrepancies arise, one should iteratively refine the architecture until consistency is achieved.






When both models reliably produce matching results, the next phase involves integrating these models into an API to process live data and deliver real-time predictions to meet client demands.

This structured approach not only enhances the model's efficiency but also solidifies its reproducibility, leading to more dependable and verifiable machine learning workflows.

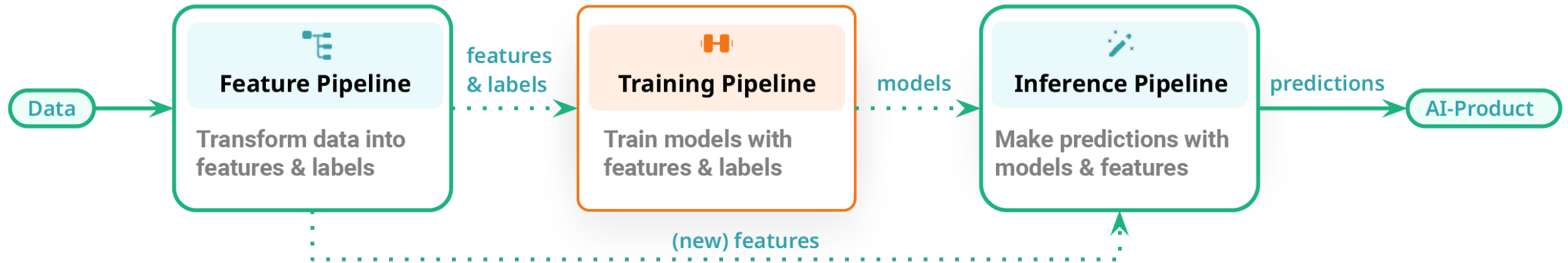
While adapting the system to meet these high reproducibility standards may be challenging, the effort is justified by the long-term benefits, ensuring the system aligns with the rigorous data and statistical requirements that are increasingly expected as machine learning becomes more prevalent and demanded.





Feature Training Inference (FTI) Pipelines

FTI Pipeline

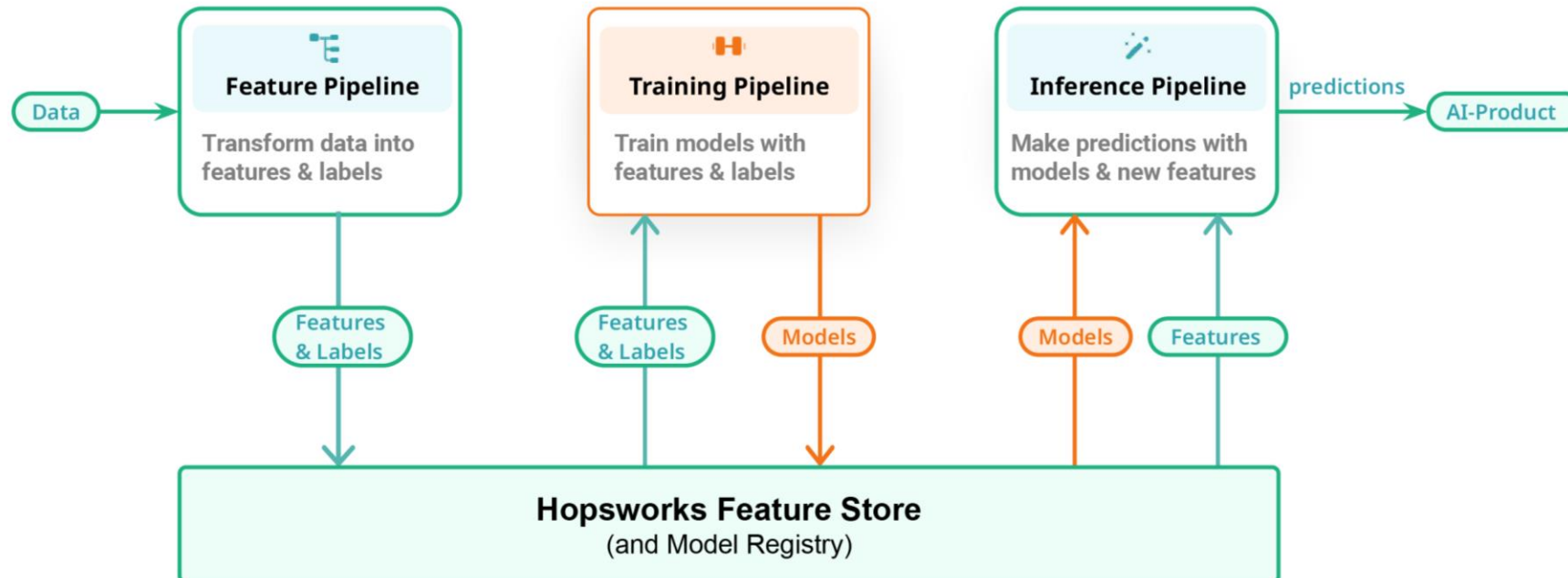


This pattern proposes creating not one but three different functional and independent pipelines:

- **Feature pipeline** that takes as input raw data that it transforms into features (and labels)
- **Training pipeline** that takes as input features (and labels) and outputs a trained model, and
- **Inference pipeline** that takes new feature data and a trained model and makes predictions

Benefits of FTI Pipeline pattern

- Modular Pluggable pipelines
- Each pipeline can use the best framework/set of tools for its requirements
- Each pipeline is operated and maintained independently
- Separated pipelines are connected via Features, Labels and Models



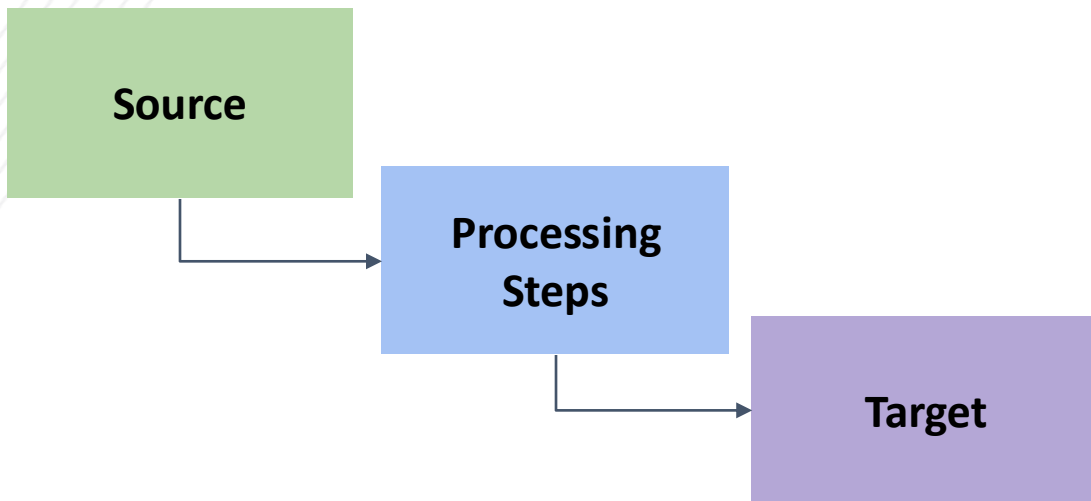
A man with a beard and glasses, wearing large white headphones, is seated at a desk. He is wearing a blue button-down shirt. His hands are on a laptop keyboard. In the background, there is a blurred bookshelf filled with books. The image has a teal overlay with decorative geometric shapes: a teal circle and blue lines in the top left, and concentric dotted circles and solid blue shapes in the top right and bottom right.

Pipelines with Sci-kit learn

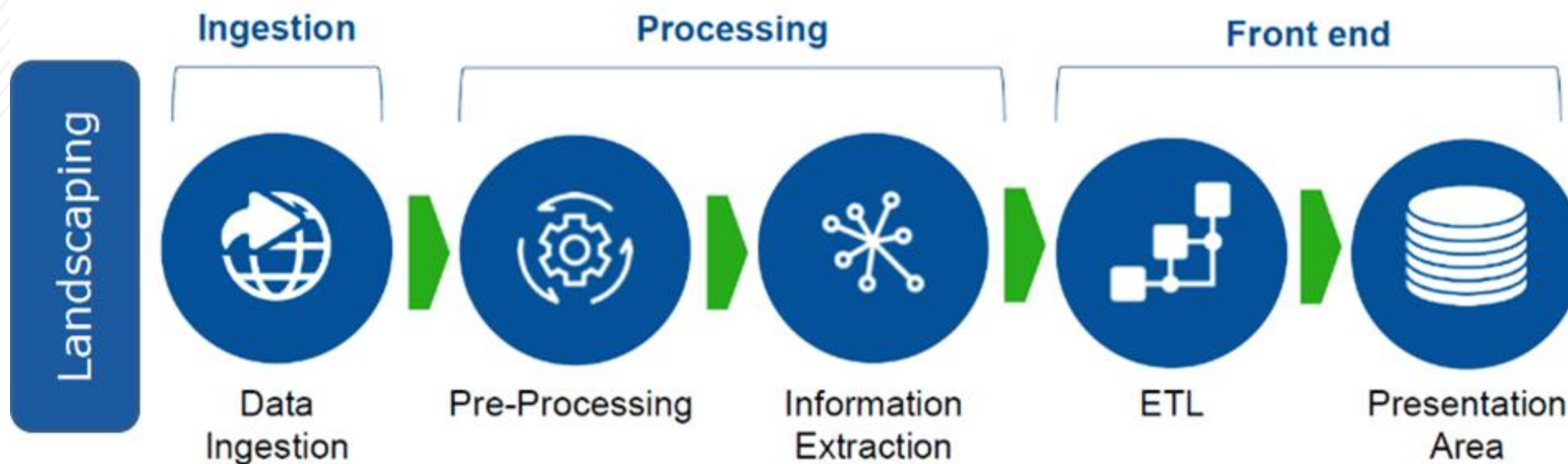
Pipelines

What is a pipeline?

A data pipeline is a means of transforming data. Along the way, data is transformed and optimized, reaching a state that can be analyzed and used to gain value assertion.



Pipelines



Pipelines

Advantages

- They eliminate most manual steps in the process and enable automated, seamless, and repeatable data flow from one stage to another.
- They are essential for real-time analytics to help make faster, data-driven decisions.
- Store data in the cloud.
- Host data in multiple sources.
- By consolidating data from your various silos into a single source of truth, you ensure consistent data quality and enable rapid data analysis for business insights.

Custom Transformers

What is scikit learn?

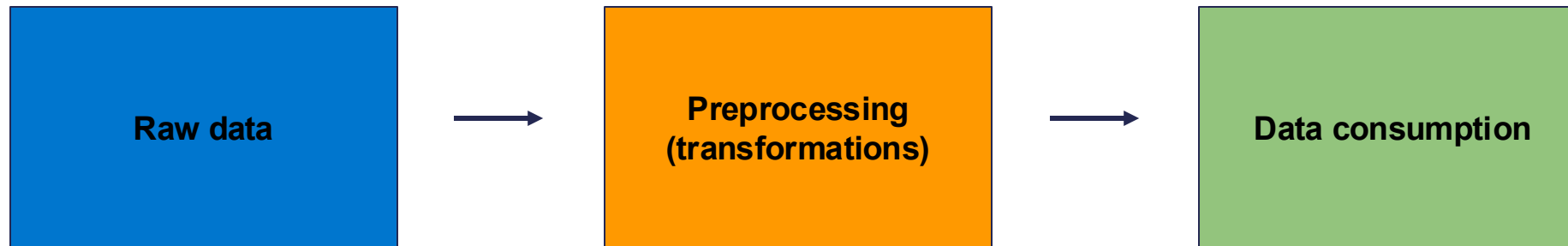
Scikit-Learn is an open-source library for Python. It features classic ML algorithms such as: classification, regression, clustering, and dimensionality reduction.

In addition, it features compatibility with other Python libraries such as NumPy, SciPy, and matplotlib.



Custom Transformers

Data Transformation and Custom Transformers



Custom Transformers

Transformer

Transformers are classes that allow data transformations while pre-processing data for machine learning.

Examples of transformers in Scikit-Learn are:

- **SimpleImputer**
- **MinMaxScaler**
- **OrdinalEncoder**
- **PowerTransformer**

to name a few.

Custom Transformers

Transformer

Transformers implement a method called transform. The use case applies to:

- **fit**, where some parameters can be learned from X and y . In general, the fit method operates with the training data and stores some resulting state in the object.
- **transform**, where X will be transformed using the parameters learned during fit.

Custom Transformers

Transformers with `BaseEstimator` and `TransformerMixin`

When you create a transformer, you need to create a class that inherits from both the **`sklearn.base.BaseEstimator`** and **`sklearn.base.TransformerMixin`** libraries.

The end result will allow a model to not only be tuned for **predictions**, but also used in combination with other scikit-learn tools such as grid search or pipelines, which can be used to make our code cleaner and easier to maintain.

Transformation Pipelines

Transformation Pipelines

Scikit-Learn provides the Pipeline class to help with such transformation sequences.

```
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)
```

Transformation Pipelines

1. Dependencies

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

2. Pipeline Constructor

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

3. Apply Transformations

```
housing_num_tr = num_pipeline.fit_transform(housing_num)
```

A man with a beard and glasses, wearing large white headphones, is seated at a desk. He is looking towards the right side of the frame. He is wearing a blue button-down shirt. In the background, there is a bookshelf filled with books. The image has a teal overlay with geometric shapes: a large teal circle in the top left, a series of concentric dotted circles in the top right, and a purple circle with a dotted pattern in the bottom right.

Documentation and Reporting

Why documenting your ML Pipelines?

- The documentation of your ML pipeline steps and outputs is an essential practice that enables:
 - Reproducibility
 - Collaboration
 - Maintenance and debugging
 - Auditing and compliance

What to include in your ML pipeline documentation?

- High-level overview
- Step descriptions
- Purpose
- Input and Output
- Parameters and Hiperparameters
- Dependencies
- Code snippets

Key takeaways for pipeline implementations

- Always use Pipelines to aggregate, organize, and move data.
- Seek to eliminate most manual steps from the process and enable an automated and seamless flow of data from one stage to another. (Automation)
- Design custom transformers when necessary.

Other Pipelines management tools

Layer / Category	Representative Tools	Purpose / Notes
CI/CD Layer	GitHub Actions, GitLab CI, Jenkins	Automate builds, testing, and deployment of ML pipelines.
Workflow Orchestration	Kubeflow Pipelines, Apache Airflow, Metaflow, ZenML, Dagster, Prefect	Manage complex multi-step ML workflows with scheduling and dependencies.
ML Lifecycle Management	MLflow, TFX, SageMaker Pipelines, Vertex AI, Azure ML	Track experiments, version models, and manage training-to-deployment pipelines.
Feature/Data Layer	Feast, Spark, Apache Beam, Dataflow	Handle feature storage, data ingestion, and transformation for training and inference.
Model-Level Pipelines	scikit-learn Pipeline, PyTorch Lightning Trainer, TensorFlow tf.data	Automate preprocessing and modeling steps within an experiment ensuring training-serving parity and reproducibility.
Monitoring & Drift Detection	Evidently AI, WhyLabs, Arize AI, Fiddler AI	Monitor data and model performance in production environments.

The background is a solid teal color. It features several abstract decorative elements: a dark purple circle in the top-left corner, a light blue circle partially visible in the top-left, a large purple circle in the bottom-left, a purple and blue elongated shape in the top-right, and a series of thin, concentric, light teal lines in the bottom-right corner.

Code Demo Sample of FTI with Scikit-learn pipelines

References:

- [How to build and deploy reproducible ML Pipeline, Galli S., 2020](#)
- [How to solve reproducibility in ML](#)
- [From MLOPS to ML Systems with FTI Pipelines](#)



D.R.© Tecnológico de Monterrey, México, 2024.
Prohibida la reproducción total o parcial
de esta obra sin expresa autorización del
Tecnológico de Monterrey.