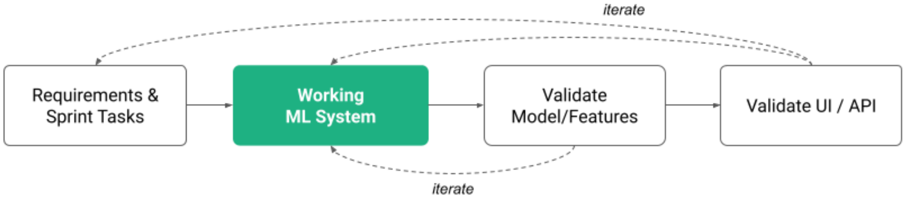
Machine Learning Operations (MLOps)

Jim Dowling
jdowling@kth.se

MLOps for Developing Machine Learning Products

- Get to a working ML system with a baseline ASAP, so that you can iteratively improve it.
- A goal of MLOps is to improve both iteration speed and quality when developing ML



MLOps: what does Improving Iteration Speed mean?

- Safe incremental updates: make small changes to your source code with confidence that your changes will not break anything (downstream clients, deployments on different platforms), performance regressions, etc)
- Tighter iteration loop: the time taken to run tests or experiments should not dominate the time taken to make the source code changes
- A faster iteration loop makes developers happier and more productive

MLOps: iteratively Develop and Test ML Systems

- ML-enabled products **evolve over time**:
  - The available input data (features) change over time
  - The target you are trying to predict changes over time
  - With the help of automation, how can quickly and reliably develop, test, and deploy ML-enabled products without affecting their ongoing operation?
- We should aim to **automate the testing and deployment** of ML-enabled Products

MLOps: Automated Testing to Improve ML Product Quality

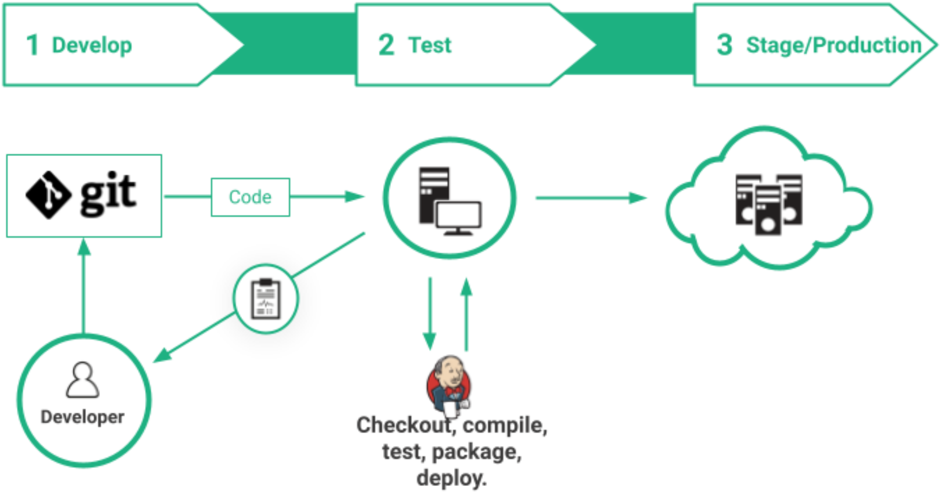
- The goal is to be able to reliably build:
  - trustworthy features using feature pipelines and data
  - a trustworthy model using your trustworthy features
  - an AI-enabled product using trustworthy models and features
- To this end, features and models must be tested
- Tests should run automatically as part of a CI/CD workflow

Prediction Feedback to Improve ML Product Quality

- Acquire user feedback with a user-interface to quickly improve the quality of your ML-enabled product and model
- Log predictions and features to enable developers to quickly find and understand the root cause of poor quality predictions
- Compare historical predictions with outcomes (or proxy metrics for outcomes) to inform when a model is stale
- Monitor feature or label drift to identify when a model needs to be re-trained

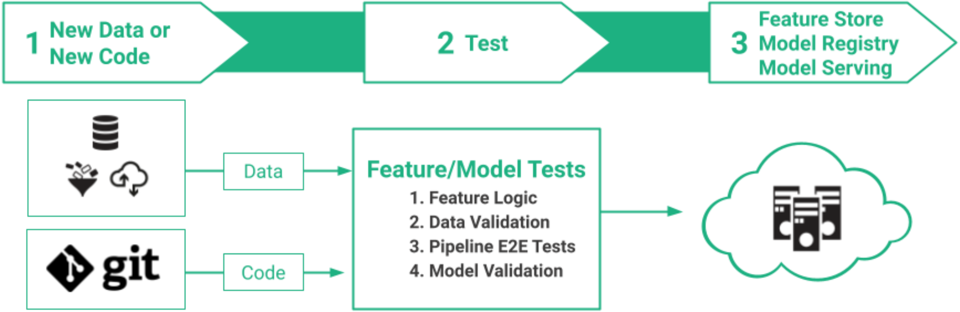
DevOps for reliable software development

- DevOps is a set of practices, tools, and a cultural philosophy that automate and integrate the processes between software development and IT teams. Key technologies are **version control, automated testing, versioning** of production deployments.

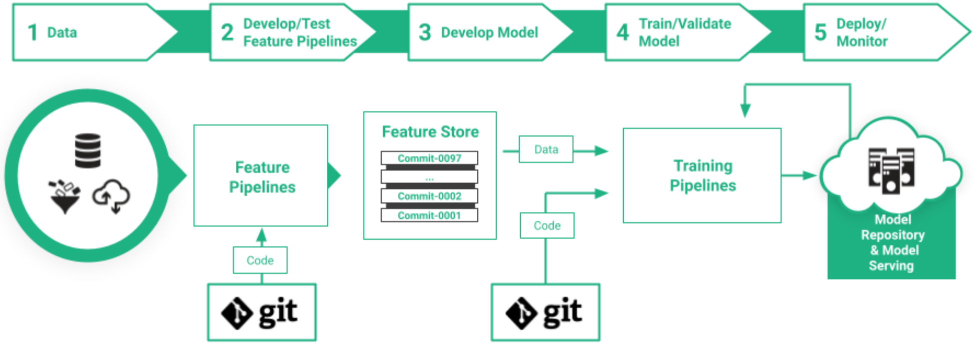


Changes in either source code or Data can break your ML Product

- In DevOps, changes in source code trigger automated testing and deployment.
- In MLOps, changes in either source code or incoming data trigger automated testing and deployment.

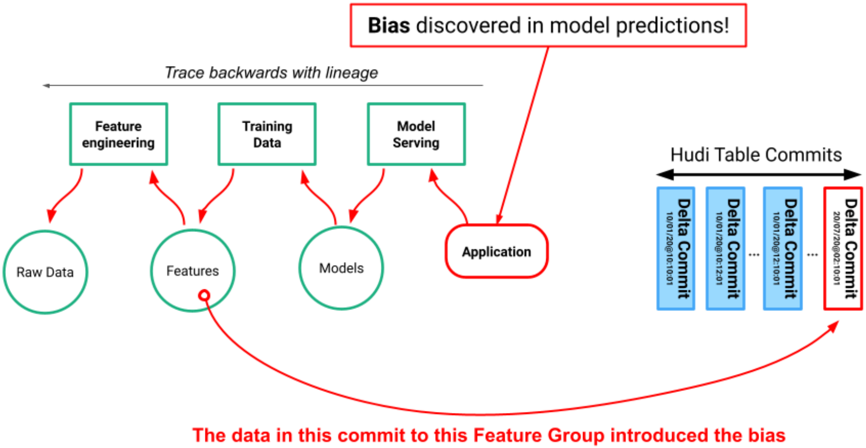


A Complete MLOps Platform with Automated Testing

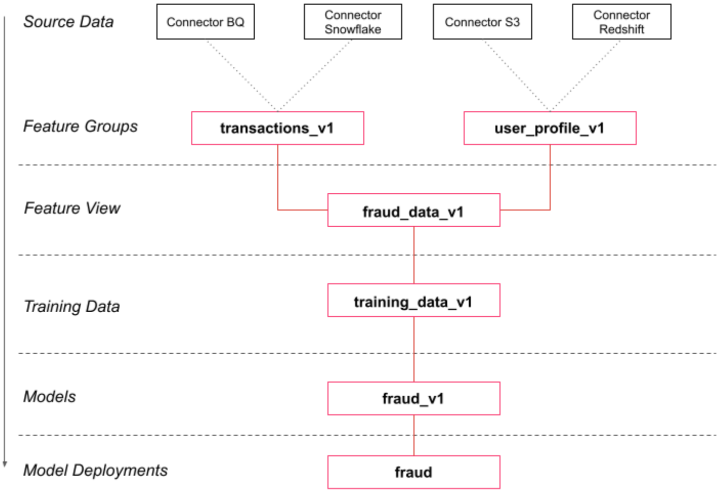


Lineage in MLOps

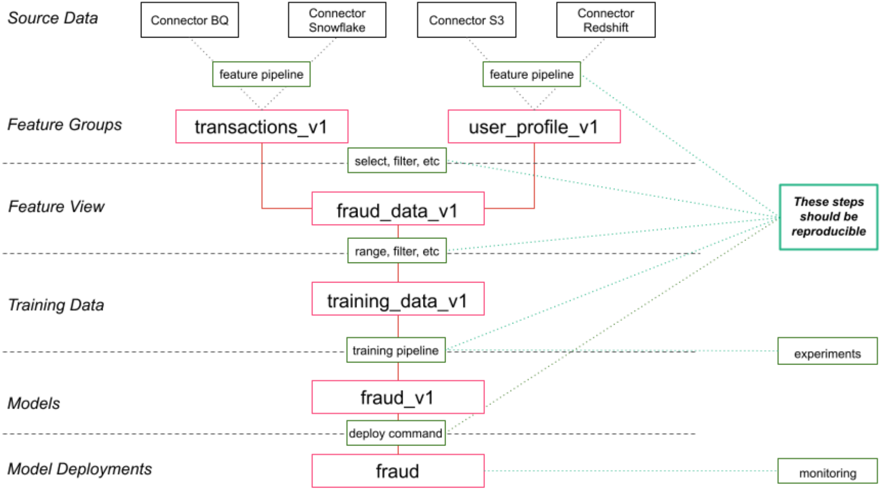
What was the root cause for the introduction of model bias?



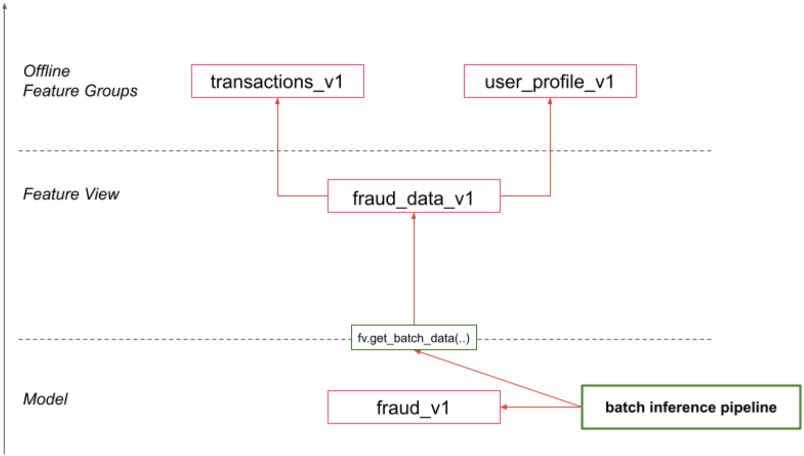
Lineage in Hopsworks: from Data to Features to Models to Deployments



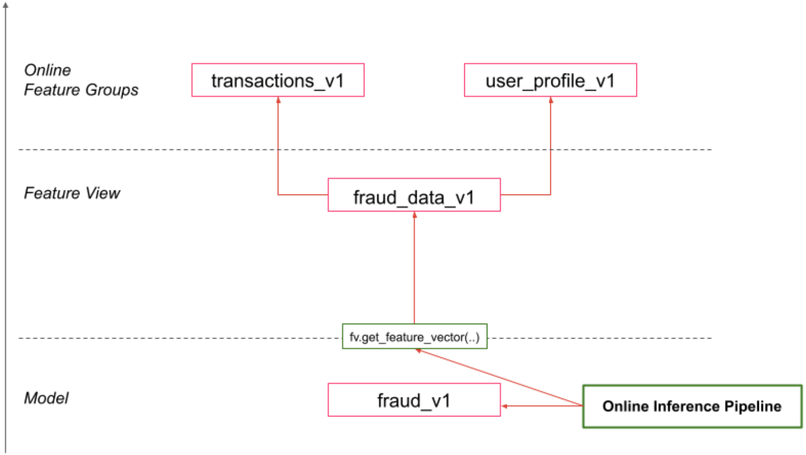
Reproducible ML Assets makes for better Data Science



Reverse Lineage for Batch Inference Pipelines

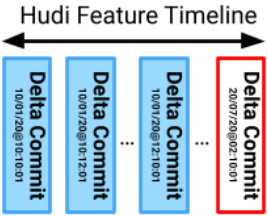


Reverse Lineage for Online Inference



Data Versioning in Apache Hudi

- Lineage involves storing metadata about both state and pipeline executions of versioned ML Assets, enabling the discovery of the provenance of any given ML asset.
- Lineage facilitates Debugging, Analyzing, Cleaning of ML Assets and Pipelines, and Reproducing ML Assets.
- If a stateful ML asset supports time-travel, you can track and recover its state at a point in time in the past. Git provides time-travel for source code. Hudi provide time-travel for data commits in cached Feature Groups in Hopsworks.



Versioning of ML Assets

Mutability of ML Assets in Hopsworks

ML Asset	**Mutable Data**	**Mutable Metadata**
Feature Groups	Mutable Hudi tables	Description, tags, Feature Views
Feature Views	Immutable	Description, tags, Training Datasets, Batch Inference Datasets
Training Data	Immutable	Description, tags
Models	Immutable	Description, tags
Model Deployments	Mutable (Hot-swappable models)	Description, tags
Prediction Logs	Immutable	N/A

Versioning of ML Assets in Hopsworks

ML Asset	Schema Versions	Data Versions	How to reproduce
Feature Groups	Yes	Hudi Commits	Re-run the feature pipeline with the same backfill data
Feature Views	Yes	No	Re-run the same feature view creation commands
Training Data	Yes	No	The parent feature view can recreate training datasets.
Models	Yes	No	Re-run the training pipeline with the same training data,hyperparams, and random number seeds
Model Deployments	No	No	Deploy the same model name/version with the same transformer/predictor code
Prediction Logs	No	No	Not possible if model is stochastic. Emulate by re-running inference pipeline on prediction requests.

Handling Versioning Challenges in Hopsworks

Scenario or Problem	Hopsworks Behavior
New run of a feature pipeline	New commit(s) are made to the feature group(s)
Change in how a feature is computed	Create a new feature group version and backfill the new feature group version.
Breaking Schema Change in a Feature Group	Create a new feature group version and backfill the new feature group version.
Successful run of a training pipeline	A new version of a model is added to the Model Registry.
New Model version Deployed	Connect the new model version to any online feature groups required by the model.
Training/Inference Skew for Model-Specific Transformations	Run the exact same transformation code in training and inference pipelines with the same python dependencies.
Training/Inference Skew for On-Demand Feature Logic	Run the same versioned on-demand feature code in training and inference pipelines with the same python dependencies.
A/B test new models with Blue/Green rollouts	Serving infrastructure supports both the old and new versions of models and online feature groups.
Model deployment upgrade/rollback	Support older and newer versions of online feature groups and models with synchronized upgrade/ rollback.

Versioning of Source Code
Packaging of Pipelines

Packaging Pipelines as Installable Python Artifacts

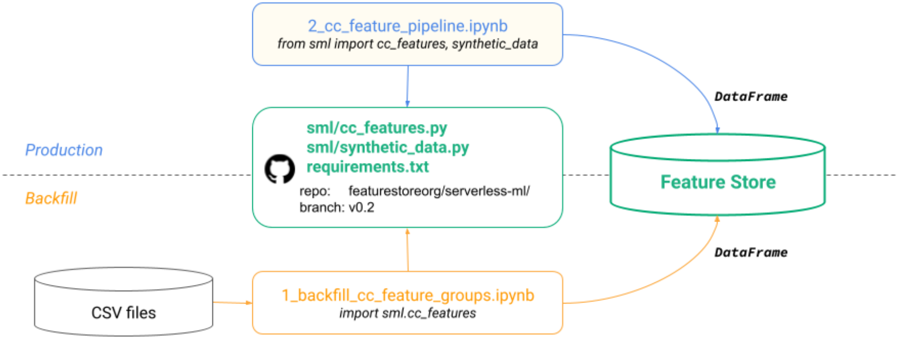
Python Artifact	Versioning
Wheel	A name and a version for the wheel file. Also needs the URI to the wheel file.
PyPi or Conda	Name, version of Python Library. Also needs the URL to the PyPi or Conda server.
Python module in Git Repo	A URI to a file in a Git repository, including the branch or tag for the release.
Python package	A URI to a directory in a Git repository, including the branch or tag for the release.

Manage OS package dependenices for Pipelines

Packaging	Versioning	Example Platforms
Prebuilt Containers	The system needs to be able to build and deploy the container image to a Docker Repository	Kubeflow
Custom Container Images	Define a Dockerfile, build and test the container image, and deploy to a secure Docker Repository.	Kubeflow
PyPi or Conda Requirements	Define Python dependencies in source code or a requirements.txt file. The system Installs the Python dependencies on top of a base container image.	Hopsworks, MLFlow, Databricks, Modal, Hugging Face Spaces, AWS Chalice, GitHub Actions, and many more
apt install commands	You can specify libraries to "apt install' on top of a base container image.	Modal, Github Actions

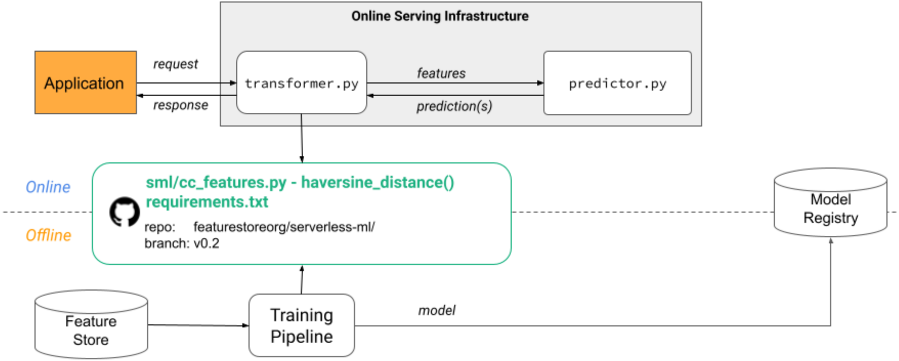
Reuse Versioned Feature Code for Prod/Backfill Pipelines

- Both production and backfill feature pipelines should use (1) the same Python library versions (**requirements.txt**) and (2) run the same feature engineering code (same **sml** package). Use the same GitHub Repo and branch/tag for both pipelines.



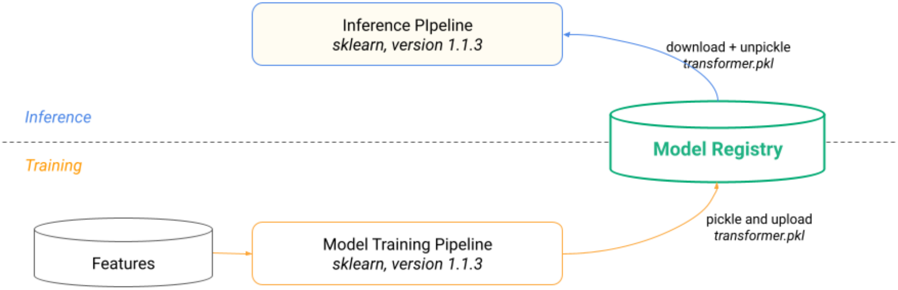
Reuse On-Demand feature code in Training/Inference Pipelines

- Sometimes features have to be computed on-demand as UDFs in Python.
- E.g., the `haversine_distance(..)` feature in `sml/cc_features.py` is computed both in training and in an online inference pipeline.



Model-Specific Transformation Pipelines in Scikit-Learn

- Pickle and save your sklearn transformer object or sklearn pipeline object
- Save the pickled object to the model repository along with your versioned model
- In your batch or online inference pipeline, download the model and transformer object, and apply same transformations as were applied in training
- Ensure you have the same library versions (e.g., same sklearn version) in both training and inference pipelines



Versioning of Data: Schemas and Commits

Schema Versioning (Data Contracts)

- In Hopsworks, you can make non-breaking schema changes that do not require updating the schema version.
- Appending features with a default value is a non-breaking schema change

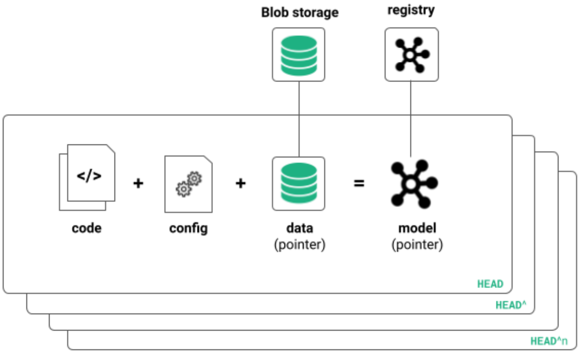
```
from hsfs.feature import Feature

features = [
    Feature(name="id",type="int",online_type="int"),
    Feature(name="name",type="string",online_type="varchar(20)")]

fg = fs.get_feature_group(name="example", version=1)
fg.append_features(features)
```

- Breaking schema changes require updating the schema version for a Feature Group.

```
fg1 = fs.create_feature_group(name=”example”, version=1)
df = fg1.read()
fg2 = fs.create_feature_group(name=”example”, version=2, features=new_features, …)
fg2.insert(df) #backfill the new feature group with data from the prev version
```

Data Versioning with Git

- Versioning code and data together using Git, such as Data Version Control (DVC).
- Warning: DVC is impractical and inefficient for large amounts of data, as it only stores pointers to blobs (binary large objects) in Git.



Data Versions in feature groups with Apache Hudi

View Commits in the Hopsworks UI in the “Activity” tab of a Feature Group

Unit tests for Feature Logic and integration tests for Feature Pipelines

ML Test Score Criteria by D. Sculley et al

Feature Tests

- Test that the distributions of each feature match your expectations
- Test the relationship between each feature and the target
- Test the cost of each feature (e.g., latency)
- Test all code that creates input features

Model Tests

- Test that every model specification undergoes a code review
- Test the relationship between offline proxy metrics and the actual impact metrics
- Test the impact of each tunable hyperparameter
- Test the effect of model staleness
- Test against a simpler model as a baseline
- Test model quality and model bias using important data slices (*evaluation sets*)

https://www.eecs.tufts.edu/~dsculley/papers/ml_test_score.pdf
Eric Breck Shanqing Cai Eric Nielsen Michael Salib D. Sculley, Proceedings of IEEE Big Data (2017)

ML Test Score Criteria by D. Sculley et al

Infrastructure Tests

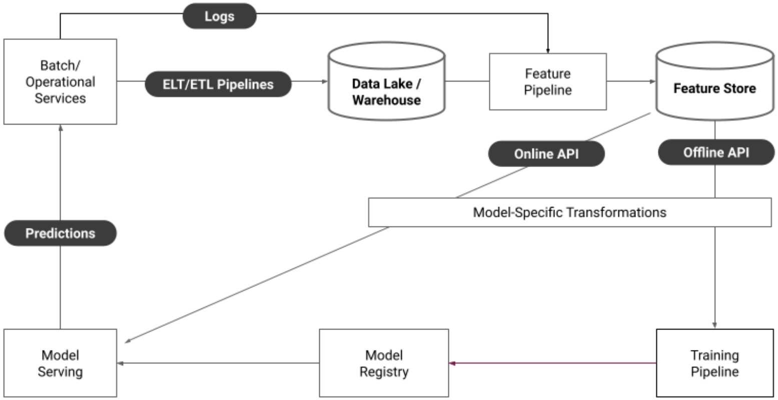
- Integration test the feature, training, and inference pipelines.
- Test the reproducibility of model training.
- Test model quality before attempting to serve it.
- Test models via a canary process before production serving.
- Test that a model deployment can be safely rolled back to a previous version.

Monitoring Tests

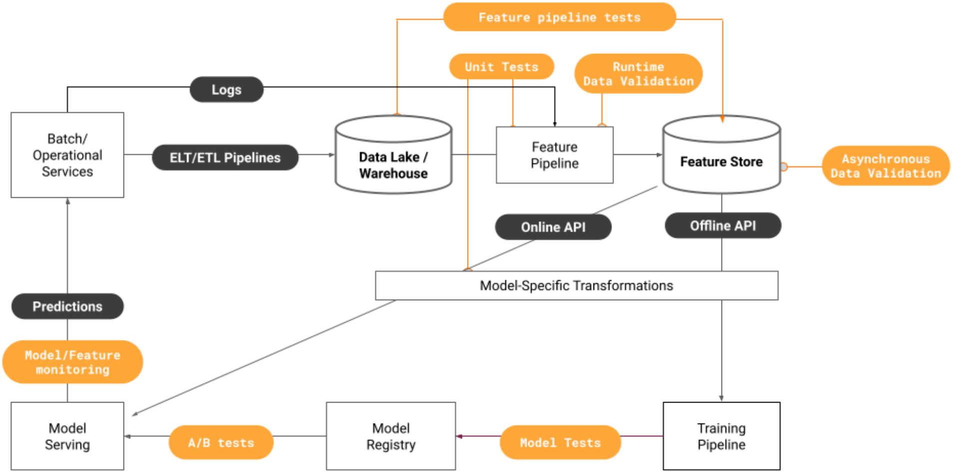
- Test data quality for features in feature pipelines
- Test that data invariants hold in training and serving inputs
- Test that your training and serving features compute the same values
- Test for model staleness
- Test for NaNs or infinities appearing in your model during training or serving
- Test for regressions in prediction quality on served data

https://www.eecs.tufts.edu/~dsculley/papers/ml_test_score.pdf
Eric Breck Shanqing Cai Eric Nielsen Michael Salib D. Sculley, Proceedings of IEEE Big Data (2017)

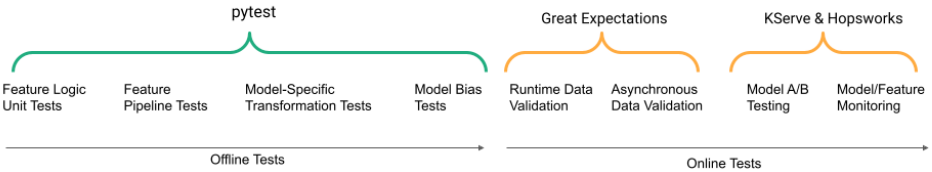
Where can we add tests to Operational ML Systems?



Where can we add tests to Operational ML Systems?

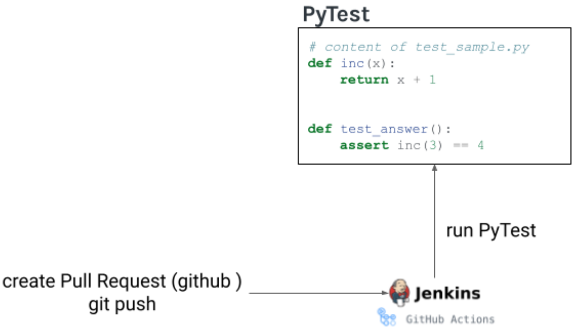


Offline and Online Tests for Operational ML Systems



Unit tests for Features with Pytest

- Unit tests are used to test individual functions
  - You run the function with a given input, and expect a certain output
- In ML, unit tests can be used to validate Feature Logic, On-Demand Features, and Model-Specific Transformation Functions



Refactor Feature Engineering Code into Testable Functions

```
def compute_features(df : Dataframe): -> Dataframe
    if config["region"] == "UK":
        df["holidays"] = is_uk_holiday(df["year"], df[" week"])
    else:
        df["holidays"] = is_holiday(df["year"], df["week"])

    df["avg_3wk_spend"] = df["spend"].rolling(3).mean()
    df["acquisition_cost"] = df["spend"] / df["signups"]
    df["spend_shift_3weeks"] = df["spend"].shift(3)
    df["special_feature1"] = compute_bespoke_feature(df)
    df["spend_b"] = multiply_columns(df["acquisition_cost'], df['B'])
    return df

df = loader.load_actuals(dates) # e.g. spend, signups
df = compute_features(df)

feature_group = fs.get_feature_group("customer_features", version=1)
feature_group.insert(df)
```

This feature logic does not contain independently testable features

[Example from Hamilton]

Write Unit Tests for the Feature Functions

```python
def avg_3wk_spend(spend: pd.Series) -> pd.Series:
    """Rolling 3 week average spend."""
    return spend.rolling(3).mean()
```

......

```python
def spend_per_signup(spend: pd.Series, signups: pd.Series) -> pd.Series:
    """The cost per signup in relation to spend."""
    return spend / signups
```

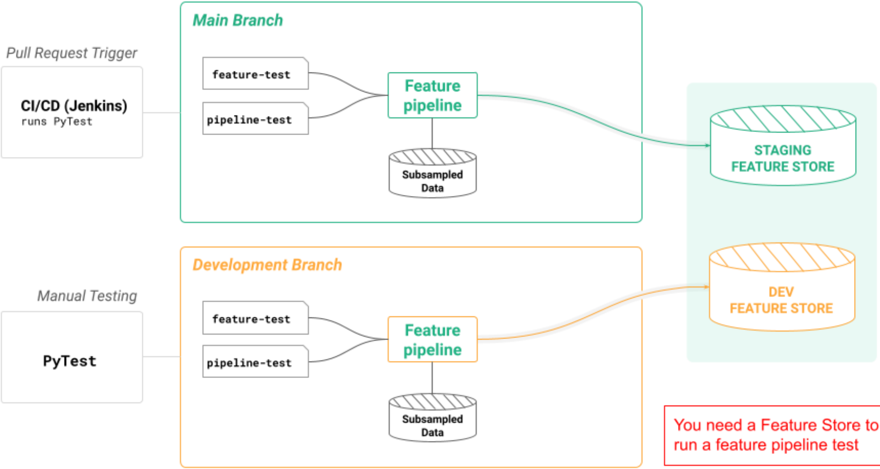
Unit Test

```python
@pytest.fixture
def get_spends(self) -> pd.DataFrame:
    return pd.DataFrame([[20, 40], [5, 4], [4, 10],
                        columns=["spends", "signups", "spend_per_signup"])

def test_spend_per_signup(get_spends : Callable):
    df = get_spends
    df["res"] = spend_per_signup(df["spends"), df("signups")]
    pd.testing.assert_series_equal(df["res"], df["spend_per_signup"])
```

Recommended Pytest directory structure

```
root
├ features                              → Reusable feature code is defined as a
│  ├ feature_engineering.py                Python function inside a module
│  ├ on_demand_features.py
│  └ custom_transformations.py
├ pipelines                             → Python programs that are run as  Pipelines
│  └ feature-pipeline.py
│  └ training-pipeline.py
│  └ batch-inference-pipeline.py
├ test_features                         → Pytest unit tests for feature functions
│  └ test_features.py
├ test_pipelines                        → Pytest integration tests for pipelines
│  └ test_pipelines.py
```

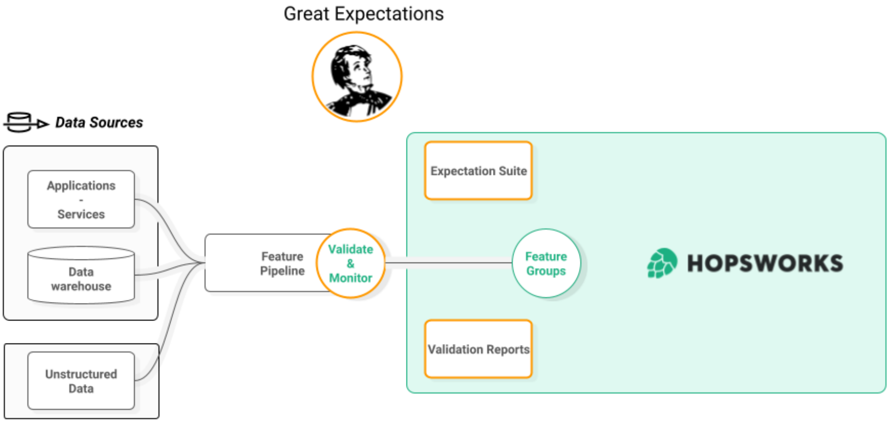
Feature Pipeline Tests in a CI/CD Setup



You need a Feature Store to run a feature pipeline test

Data Validation with Great Expectations

Data Validation with Great Expectations in Feature Pipelines



Feature Data Validation Rules

Validate the features of every prediction request against a set of expectations on that data.

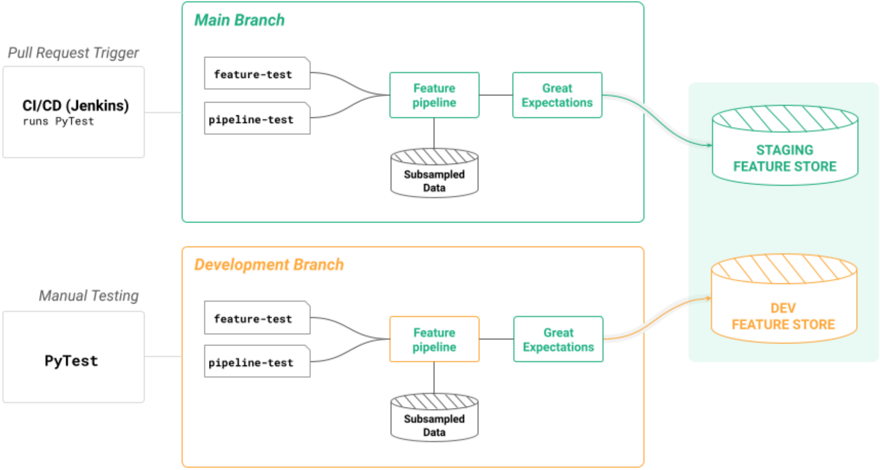
- **Type checks** — passing a string of "1" instead of the integer 1
- **Value ranges** — -10 for a distance or 1000 for an age
- **Missing values** for a required feature
- **Set membership** — an unknown value for a categorical feature
- **Table expectations** — checking that all the expected feature names are present

Great Expectations and Pandas DataFrames

Great Expectations and Pandas DataFrames

```python
import great_expectations as ge
from great_expectations.core import ExpectationSuite, ExpectationConfiguration

aggs_df = ge.from_pandas(aggs_df)

# Generate Expectation Suite using a Profiler
expectation_suite, _ = aggs_df.profile(profiler=ge.profile.BasicSuiteBuilderProfiler)

# Explore expectations using a sample df
validation_result = aggs_df.expect_column_values_to_be_increasing(
    column="datetime",
    strictly=True
)
expectation_suite.add_expectation(validation_result["expectation_config"])

# Manually add expectations and their configuration
expectation_suite.add_expectation(
    ExpectationConfiguration(
        expectation_type="expect_column_values_to_be_in_set",
        kwargs={
            "column":"fraud_label",
            "value_set": [0,1]
        }
    )
)

# Validate using the final suite
ge_aggs_df = ge.from_pandas(aggs_df, expectation_suite=expectation_suite)
validation_report_trans = ge_aggs_df.validate()
```

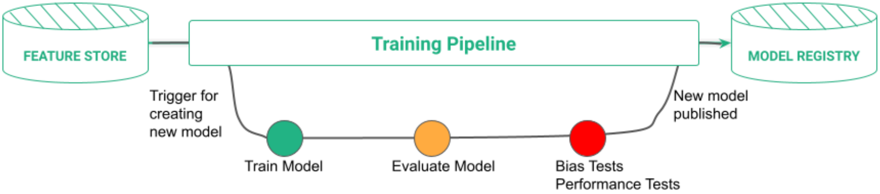
Feature Pipeline CI/CD Setup with Great Expectations



Testing Training Pipelines and Model Deployments

Testing Training Pipelines and Models

- Save the hyperparams and training data used to train the model,
- Only deploy a model that outperforms an existing model deployment (or baseline), assuming both trained on same dataset.
- Only deploy a model if it is free from bias and trained on ethical data.



Evaluating Models and Testing Training Pipelines

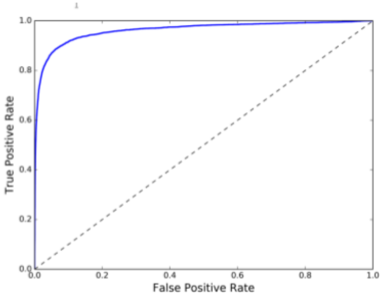
- **Evaluate Metrics** of your model for acceptable performance
- **Post-Training Tests** on the model's **learned behavior** and potential for **Bias**
- **Compare performance with existing models** before deployment
    - Compare the performance of existing models in the Model Registry, and only deploy the new model to production if it is better than existing models and all evaluating and testing metrics
- **End-to-end test the training pipeline that trains and deploys a model** to ensure correct operation

https://eugeneyan.com/writing/testing-ml/

Model Performance Evaluation

Evaluate the performance of your model on a test set

- This typically results in an **evaluation report** including:
  - performance of an established metric on a test set, for example, Area under Curve of Receiver Operating Characteristic (AUC ROC)
  - plots such as precision-recall graphs,
  - operational statistics such as inference latency.



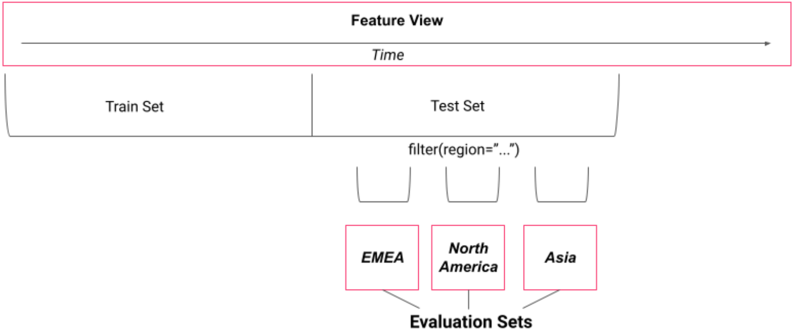
ROC Curve

Model Tests

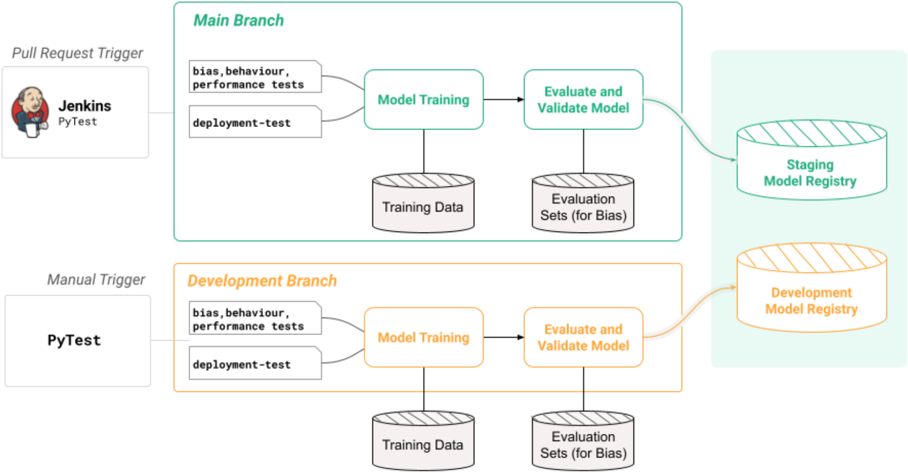
- **Post-Train Tests**
  - Model tests are run on a model after it has been trained
- **Expected Behavior Tests**
  - Test known invariants in model behavior, such as for the Titanic Survival Dataset, women are more likely to survive than men.
- **Bias Tests**
  - You should test if a model is free from bias by evaluating its performance on subsets of the test data (*evaluation sets*) that represent different groups that could be at risk of bias.

Test a Model for Bias with Evaluation Sets

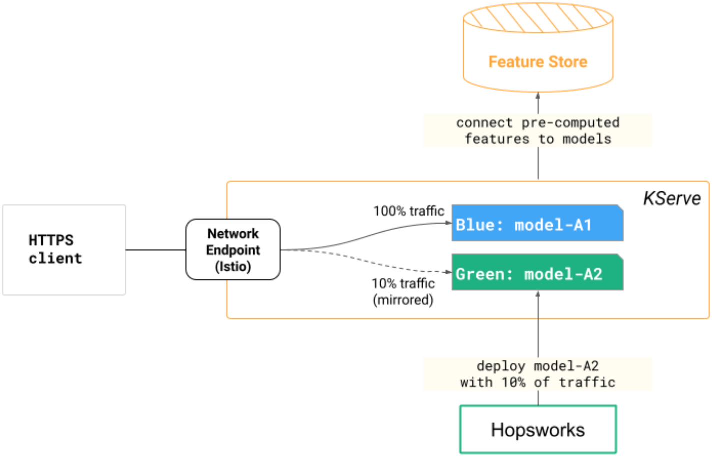
- Build **Evaluation Sets** - filters on training data (e.g., gender, ethnicity, geography) and evaluate the model performance on each evaluation set, looking for significant performance differences



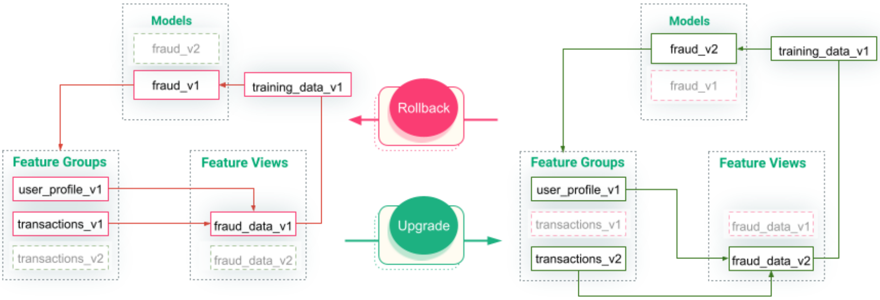
Integration (End-to-End) Tests for Training Pipelines



A/B Testing Model Deployments (Blue/Green Rollouts)



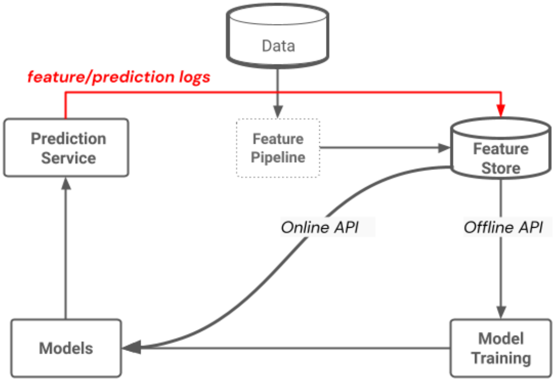
The “Big Red Button” enabled by MLOps



Model and Feature Monitoring

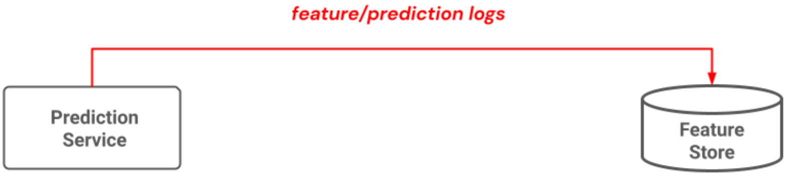
Data for AI Flywheel

more data -> better models -> more users -> more data-> ...

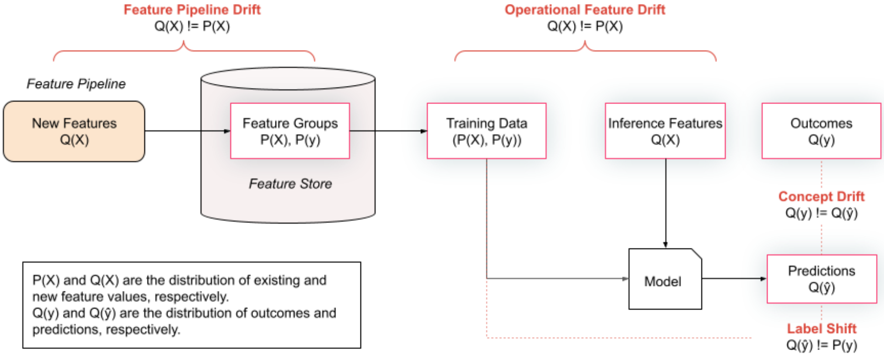


Feature and Prediction Logging

- Logs of predictions and features can be used for:
  - Debugging
  - Model/Feature/Performance Monitoring
  - To create new training data for models
- You should log untransformed feature values and predictions
  - Log before model-specific transformations & reverse_transform predictions



Monitor Features, Labels, Predictions, Outcomes for Drift



What is practical to measure for Data Drift?

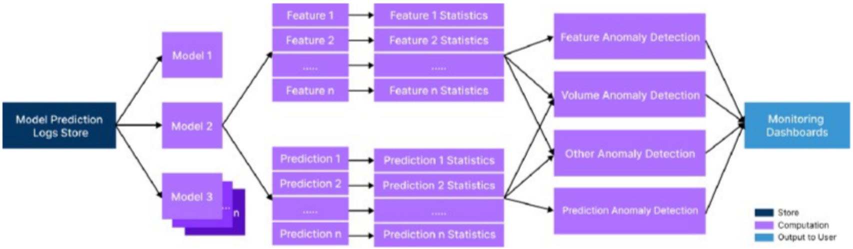
- **Feature Pipeline Drift**
  - The distribution of a feature in a DataFrame that is being written to a Feature Group is significantly different from the distribution of that feature stored in the Feature Group.
- **Operational Feature Drift** (aka Covariate Shift)
  - The distribution of a feature in a window of Inference Data is significantly different from the distribution of that feature in the models' training data.
- **Label Shift**
  - The distribution of a window of label data in inference is significantly different from the distribution of that label in the models' training data.
- **Concept Drift**
  - The outcomes are significantly different from the model's predictions.

Case Study

Lyft Model/Feature Monitoring

"For our online use-case in which we only validate a single feature vector (row) at a time, an implementation with less overhead was required. The [monitoring] is performed async to make the latency impact on model scoring negligible."

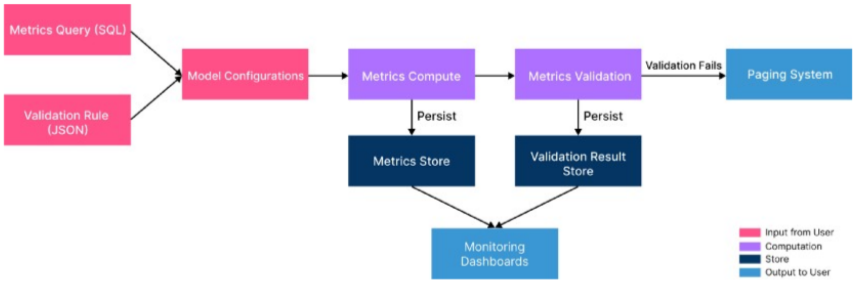
Asynchronous Monitoring of Online Models at Lyft



https://eng.lyft.com/full-spectrum-ml-model-monitoring-at-lyft-a4cdaf828e8f?gi=7207d4eed194

Lyft - Performance Drift Detection

- Automate as much as possible.
- Monitoring is a defensive investment
- Monitoring will reduce shipping velocity slightly



https://eng.lyft.com/full-spectrum-ml-model-monitoring-at-lyft-a4cdaf828e8f?gi=7207d4eed194

References

- Reliable Machine Learning: Applying SRE Principles to ML in Production, Murphy et al, O'Reilly
- Designing Machine Learning Systems: An Iterative Process for Production-Ready Applications, Chip Huyen, O'Reilly