Lecture 08: Reproducible Machine Learning

CSCI E-103: Reproducible Data Science and Machine Learning

Harvard University

- **Course:** CSCI E-103: Reproducible Data Science
- **Week:** Lecture 08
- **Instructors:** Anindita Mahapatra & Eric Gieseke
- **Objective:** Understand how to achieve reproducibility in ML projects through MLOps culture and MLflow tools

Contents

1 The Core Problem: Why Is ML Reproducibility So Hard?

Lecture Overview

This lecture addresses a fundamental challenge in machine learning: **Reproducibility**—the ability to recreate the exact same model and results given the same data and code. While this sounds simple, it's extraordinarily difficult in practice.

Key Topics:

- The "Hidden Technical Debt" in ML systems
- ML lifecycle and the roles of Data Engineer, Data Scientist, and ML Engineer
- Feature Engineering and Feature Stores
- ML Pipelines: Transformers vs. Estimators
- Model Drift and its four types
- MLOps culture and MLflow as the solution

1.1 The Iceberg Metaphor: Hidden Technical Debt

One of the most important insights in ML engineering comes from Google's famous paper on "Hidden Technical Debt in Machine Learning Systems." The key revelation is:

The Misconception vs. Reality

Common Misconception: "Machine Learning is about writing sophisticated algorithms and model code."

Reality: The actual ML code is just a **tiny fraction** of a real-world ML system. It's like the tip of an iceberg—what you see above water is tiny compared to the massive infrastructure lurking beneath.

The Hidden Infrastructure (90%+ of the system):

- **Data Collection**: Gathering data from multiple sources
- **Data Verification**: Ensuring data quality and integrity
- **Configuration**: Managing hyperparameters, feature flags, data versions
- **Feature Extraction**: Engineering meaningful inputs for models
- **Infrastructure**: Managing compute resources (CPU, GPU, clusters)
- **Monitoring**: Tracking model performance and data drift
- **Serving**: Deploying models to production systems

Example: Analogy: Building a House

Think of an ML system like building a house:

- **ML Code** = The visible part of the house (walls, roof, windows)
- **Infrastructure** = Everything you don't see (foundation, plumbing, electrical, HVAC)

You can have a beautiful house, but if the foundation is weak or the plumbing is broken, the house

is unusable. Similarly, you can have a brilliant ML algorithm, but without solid data pipelines, monitoring, and serving infrastructure, it's worthless in production.

Why does this matter for reproducibility?

To reproduce a model, you don't just need the ML code—you need to recreate the **entire environment**: the exact data version, the library versions, the configuration settings, the feature engineering logic, and more. This is why reproducibility is so challenging.

1.2 The Fragmented Ecosystem Problem

ML projects don't use a single tool. They involve a complex mix of:

- **Languages**: Python, R, SQL, Scala, Java
- **Frameworks**: Scikit-learn, PyTorch, TensorFlow, XGBoost, Spark MLlib
- **Deployment Environments**: Docker, Kubernetes, AWS SageMaker, Databricks

Dependency Hell

"It worked on my laptop!"
This is the most common phrase in ML engineering. When you manually connect different tools, you enter what's called **dependency hell**:

- You trained a model with `pandas==1.3.0` but production has `pandas==1.5.0`
- Your colleague uses `numpy==1.21` while you have `numpy==1.23`
- The server has Python 3.8 but you developed on Python 3.10

Even tiny version differences can cause different results or outright failures.

1.3 Models Are Living Assets

Perhaps the most counterintuitive aspect of ML: **models are not static artifacts**. They're "living" assets that degrade over time.

Definition: Model Drift

Model Drift is the phenomenon where a deployed model's performance degrades over time because the real-world data patterns have changed since the model was trained.
Think of it like a driver's license: passing the driving test (training) doesn't mean you can handle every situation on real roads (production), especially when traffic laws change or new types of intersections appear.

This means that even if you perfectly reproduce a 6-month-old model, it might be **useless** because the world has changed. What truly matters is:

1. **Tracking** exactly how that model was built (so you can understand what worked)
2. Having an **automated pipeline** to quickly retrain on fresh data

This is precisely what MLOps and MLflow solve.

2 The ML Lifecycle and Key Roles

2.1 The Six Stages of an ML Project

Every ML project follows a cyclical pattern, not a linear one. Understanding this lifecycle is crucial.

The ML Project Lifecycle

Forward Process:

1. **Raw Data** → Collect data from various sources
2. **ETL (Cleanse)** → Extract, Transform, Load—clean and prepare data
3. **Featurize** → Create meaningful features for the model
4. **Train** → Train the model (the "small" part!)
5. **Serve/Inference** → Deploy and serve predictions
6. **Monitor** → Watch for performance degradation

Feedback Loop: When monitoring detects drift → Return to step 1, 2, or 3 to retrain

Example: The Feedback Loop in Action

Scenario: You built a fraud detection model in January.
May: Monitoring shows false positive rate increased by 40%.
Investigation: New payment methods (Apple Pay, crypto) have emerged that your model never saw during training.
Action:

- Go back to **Featurize**: Add features for new payment types
- **Train**: Retrain on 6 months of new data
- **Deploy**: Push the updated model to production

This is why ML is a **continuous cycle**, not a one-time project.

2.2 The Three Key Personas

The ML lifecycle requires collaboration between three specialized roles:

The Silo Problem

Traditionally, these three roles work in **silos** (separate teams with poor communication):

1. DS builds a model in Jupyter Notebook → Hands off code to MLE
2. MLE rewrites the code for production (Docker, APIs) → Different environment!
3. Versions get confused, environments differ, results can't be reproduced
4. Deployment takes **weeks or months** instead of hours

MLOps and **MLflow** break down these silos by standardizing how models are tracked, packaged, and deployed.

Table 1: *The Three Pillars of ML Projects*

Aspect	Data Engineer (DE)	Data Scientist (DS)	ML Engineer (MLE)
Primary Mission	Build data infrastructure	Extract insights & build models	Productionize & operate models
Key Tasks	- Data ingestion - Data storage - ETL pipelines - Data curation	- Feature engineering - Model development - Training & tuning - Validation	- Model evaluation - Packaging - Deployment & serving - MLOps pipelines
Analogy	Raw material supplier & factory builder	Prototype developer	Mass production line operator
Where in Medallion	Bronze → Silver	Silver → Gold	Gold → Production

3 Feature Engineering and Feature Stores

3.1 What is Feature Engineering?

Definition: Feature Engineering

Feature Engineering is the process of using domain knowledge to transform raw data into meaningful inputs (features) that help ML algorithms learn more effectively.

Andrew Ng's famous quote: "Applied machine learning is basically feature engineering."

This means that the quality of your features often matters more than the sophistication of your algorithm. Good features can make a simple model outperform a complex one with poor features.

Example: From Raw Data to Features

Raw Transaction Data:

- Customer ID, Product Name, Purchase DateTime, Amount

Engineered Features:

- `avg_transaction_amount`: Average purchase amount per customer (aggregation)
- `purchase_frequency_7d`: Number of purchases in last 7 days (time-window aggregation)
- `is_weekend`: Whether purchase was on weekend (transformation)
- `weather_on_purchase`: Weather conditions at purchase time (external data enrichment)
- `customer_lifetime_value`: Predicted total revenue from customer (complex aggregation)

The model learns much more from `purchase_frequency_7d` than from raw timestamps. This is the power of feature engineering.

3.2 Why Feature Engineering is Expensive

Creating good features often requires:

- **Domain expertise**: Understanding what matters in your business
- **Significant computation**: "7-day rolling average" requires scanning millions of records

- **Time-consuming iteration**: Testing which features actually help

3.3 The Training-Serving Skew Problem

Here's where reproducibility becomes critical:

Training-Serving Skew

Symptom: "My model had 99% accuracy in training, but only 60% in production!"
Cause: Features were computed **differently** during training vs. serving:
During Training (Offline):

```sql
-- Batch computation, can be slow
SELECT AVG(purchase_amount) as avg_purchase
FROM transactions
WHERE customer_id = ?
GROUP BY customer_id
```

During Serving (Online):

```
# Real-time computation, must be fast
avg_purchase = (sum_so_far + new_purchase) / (count + 1)
```

Even small differences (rounding, null handling, time zones) can cause features to be **slightly different**. These small differences compound and destroy model performance.

3.4 Feature Stores: The Solution

Definition: Feature Store

A **Feature Store** is a centralized repository for storing and serving ML features. It ensures that the **exact same features** are used during both training and serving.
Key Benefits:

1. **Consistency**: Features computed once, used everywhere (eliminates skew)
2. **Reusability**: Team A's expensive "customer profile" feature can be reused by Team B, C, D
3. **Efficiency**: Avoid redundant computation of expensive features
4. **Governance**: Track feature lineage, ownership, and freshness

Example: Feature Store in Action

Without Feature Store:

- Team A computes "customer_lifetime_value" for their fraud model
- Team B computes it again (slightly differently) for their churn model
- Team C computes it yet again for their recommendation model
- Result: 3x computation cost, inconsistent features, debugging nightmares

With Feature Store:

- Feature team computes "customer_lifetime_value" once

- Stores it in the Feature Store with documentation
- Teams A, B, C all read the **exact same feature**
- Training and serving both use the Feature Store as the source of truth

4 ML Pipelines: Transformers vs. Estimators

When building ML systems, especially with frameworks like Spark ML, you'll encounter two fundamental concepts: **Transformers** and **Estimators**. Understanding the difference is crucial.

4.1 The Key Distinction: Learning vs. Not Learning

Table 2: *Transformers vs. Estimators*

Aspect	Transformer	Estimator
Purpose	Data preprocessing/transformation	**Learning** from data
Key Method	`.transform()`	`.fit()`
Input → Output	DataFrame → DataFrame	DataFrame → **Model**
Does it Learn?	No (applies fixed rules)	**Yes** (learns from data)
Examples	- StringIndexer - OneHotEncoder - StandardScaler - VectorAssembler	- LinearRegression - RandomForestClassifier - XGBoostRegressor - KMeans

Example: Cooking Pipeline Analogy

Imagine a food assembly line:

Transformers (Food Prep Machines):

- **StringIndexer** = Onion peeler (string "red" → number 0)
- **StandardScaler** = Portion measurer (normalize ingredient weights)
- **VectorAssembler** = Ingredient combiner (put all prepped items together)

Estimator (The Chef):

- **LinearRegression** = A chef who **learns** the perfect recipe by tasting many dishes
- Input: Ingredients (DataFrame)
- Output: A trained recipe (Model) that can make predictions on new ingredients

4.2 The Critical Insight: Estimators Produce Models

Here's the key insight that often confuses beginners:

Key Summary

When you call `.fit()` on an **Estimator**, the output is a **Model**.

And crucially, this **Model** is itself a **Transformer**!

Why? Because once trained, the model can `.transform()` new data into predictions.

Conceptual Flow:

1. `Estimator.fit(training_data)` → Returns a `Model`
2. `Model.transform(new_data)` → Returns predictions (DataFrame)

```python
from sklearn.preprocessing import StandardScaler  # Transformer
from sklearn.linear_model import LinearRegression  # Estimator

# Transformer: transform() takes data, returns transformed data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_train)  # fit() learns stats, transform()
    applies

# Estimator: fit() takes data, returns a trained MODEL
model = LinearRegression()
model.fit(X_scaled, y_train)  # Learning happens here!

# The trained model can now transform (predict) new data
predictions = model.predict(X_test_scaled)  # Model acts like a Transformer
```

Listing 1: Transformer vs. Estimator in Code

5 Model Drift: Why Models Degrade Over Time

5.1 Understanding Model Drift

Definition: Model Drift

Model Drift is the degradation of model performance over time due to changes in the underlying data patterns that the model was trained on.

Key Insight: Models are trained on historical data. When the "future" arrives and looks different from the "past," the model struggles.

Example: The Driver's License Analogy

Imagine you passed your driving test in 2010 (training data).

2020 Problems:

- New traffic laws you never learned
- Electric vehicles with different behaviors
- Roundabouts that didn't exist in your town
- GPS navigation (you only knew paper maps)

Your 2010 "model" (driving skills) is now outdated. This is model drift—the world changed, but your model didn't.

5.2 The Four Types of Drift

Understanding **what** drifted helps you know **how** to fix it:

Table 3: *The Four Types of Model Drift*

Drift Type	What Changed?	Response Strategy
Feature Drift	The distribution of **input features (X)** changed. *Example: A new product category appears that the model never saw.*	Review feature engineering. Retrain with new data.
Label Drift	The distribution of **target labels (Y)** changed. *Example: Fraud rate suddenly doubles due to new attack vectors.*	Review label generation process. Retrain with new data.
Prediction Drift	The distribution of **model predictions** ($\hat{Y}$) changed. *Example: Model suddenly predicts everything as "not fraud."*	Investigate model behavior. Check for data pipeline issues.
Concept Drift	The **relationship between X and Y** fundamentally changed. **(Most severe!)** *Example: COVID-19 changed how "weekend" relates to "store visits"*	**Simple retraining won't work!** Need new features, new model architecture.

Concept Drift is the Most Dangerous

Feature drift and label drift can often be fixed by retraining on new data.

Concept drift means the fundamental rules of your domain have changed. The relationship that existed in your training data **no longer exists**.

COVID-19 Example:

- **Pre-COVID**: "Weekend" → "High store traffic" (strong positive correlation)
- **During COVID**: "Weekend" → "Zero store traffic" (correlation broken)
- **Post-COVID**: "Weekend" → "Moderate online traffic" (new relationship)

Your old model learned the wrong relationships. You need to completely rethink your features and possibly your entire model architecture.

6 The Solution: MLOps and the Three "Ops"

6.1 From DevOps to MLOps

To understand MLOps, first understand the "Ops" progression:

The Evolution of "Ops"

- **DevOps** = Development + Operations
  - Automates building, testing, and deploying **software code**
  - Tools: Git, Jenkins, Docker, Kubernetes
- **DataOps** = Data + Operations
  - Automates building, testing, and deploying **data pipelines**
  - Tools: Airflow, dbt, Delta Live Tables
- **MLOps** = Machine Learning + Operations
  - Automates training, testing, deploying, and monitoring **ML models**
  - Tools: MLflow, Kubeflow, SageMaker

MLOps = DataOps + ModelOps + DevOps

It's the integration of all three: data pipelines feed feature stores, which feed model training, which feeds model serving, which feeds monitoring, which triggers retraining.

6.2 Why MLOps Matters

MLOps Goals

1. **Reproducibility**: Anyone can recreate any model at any time
2. **Automation**: Models retrain and redeploy automatically when drift is detected
3. **Collaboration**: DS, DE, and MLE work from the same platform
4. **Governance**: Track who created what model, with what data, for what purpose
5. **Speed**: Deploy models in hours, not months

7 MLflow: The Swiss Army Knife for MLOps

7.1 What is MLflow?

Definition: MLflow

MLflow is an open-source platform for managing the entire ML lifecycle. Created by Databricks, it has become the de facto standard—even competitors use it.

Key Characteristics:

- **Open Source**: Free, community-backed, works anywhere
- **Framework Agnostic**: Works with Scikit-learn, PyTorch, TensorFlow, XGBoost, etc.
- **API First**: REST APIs, Python/R/Java clients
- **Modular**: Use all components or just the ones you need

7.2 The Four Components of MLflow

MLflow provides four integrated components that solve the reproducibility crisis:

Table 4: *MLflow's Four Components*

Component	Problem It Solves	What It Does
1. Tracking	"What parameters did I use for that good model last week?"	Records all experiments: parameters, metrics, artifacts, code
2. Projects	"It worked on my laptop but fails on the server"	Packages code + environment (dependencies) together
3. Models	"How do I serve a PyTorch model in a Spark pipeline?"	Standard model format with multiple "flavors" for deployment
4. Model Registry	"Which model is in production? Is v3 tested?"	Central hub for model versioning, staging, and promotion

7.2.1 Component 1: MLflow Tracking

MLflow Tracking: Your Experiment Notebook

Problem: Data scientists run hundreds of experiments, changing parameters each time. "Which hyperparameters gave me the best result?" becomes impossible to answer.

Solution: Automatically log everything about every experiment.

What Gets Tracked:

- **Parameters**: Input hyperparameters (e.g., `learning_rate=0.01`)
- **Metrics**: Output performance measures (e.g., `accuracy=0.95`)
- **Artifacts**: Files—the model itself, plots, feature importance graphs
- **Source**: The code that ran the experiment (e.g., notebook version)
- **Tags**: Custom metadata (e.g., `team=fraud-detection`)

```
1 import mlflow
```

```
2
3  # OPTION 1: Manual Logging (explicit control)
4  with mlflow.start_run(run_name="my_experiment"):
5      # Log input parameters
6      mlflow.log_param("learning_rate", 0.01)
7      mlflow.log_param("max_depth", 5)
8
9      # Train your model (any framework)
10     model = train_my_model(...)
11
12     # Log output metrics
13     mlflow.log_metric("accuracy", 0.95)
14     mlflow.log_metric("f1_score", 0.92)
15
16     # Log artifacts (files)
17     mlflow.log_artifact("feature_importance.png")
18     mlflow.sklearn.log_model(model, "model")
19
20 # OPTION 2: AutoLog (magic one-liner!)
21 mlflow.autolog()  # <-- This single line logs EVERYTHING automatically
22
23 # Now just train as usual - MLflow captures all params, metrics, model
24 model = XGBClassifier(learning_rate=0.01, max_depth=5)
25 model.fit(X_train, y_train)  # autolog captures this automatically!
```

Listing 2: MLflow Tracking: Manual vs. AutoLog

7.2.2 Component 2: MLflow Projects

MLflow Projects: Environment Packaging

Problem: "It worked on my machine" — the eternal curse of data science.

Solution: Package the code **and** its environment together.

How It Works: Your project folder contains an `MLproject` file that specifies:

- The entry point (e.g., `python train.py`)
- The environment file (e.g., `conda.yaml` or `requirements.txt`)

Anyone can then run `mlflow run <project_path>` and MLflow automatically:

1. Creates an isolated environment with exact library versions
2. Runs the code in that environment
3. Guarantees reproducible results

7.2.3 Component 3: MLflow Models

MLflow Models: Universal Model Format

Problem: DS trained with Scikit-learn, but MLE needs to serve via Spark UDF or REST API.
Solution: Save models in a standardized format with multiple "flavors."
Model Flavors:

- `python_function` (pyfunc): Universal wrapper—any model can be called as a Python function
- `sklearn`: Native Scikit-learn format
- `pytorch`: Native PyTorch format
- `spark`: Ready for Spark UDF deployment

Key Benefit: Save a model once, deploy it anywhere. The MLE doesn't need to know if it was originally Scikit-learn or PyTorch—they just use the `pyfunc` flavor.

7.2.4 Component 4: MLflow Model Registry

MLflow Model Registry: Git for Models

Problem: "Which model is currently in production? Is v3 tested? Who approved v2?"
Solution: A central repository for managing model versions and lifecycle stages.
Key Features:

- **Versioning**: Every registered model gets v1, v2, v3, etc.
- **Stage Management**: Models transition through stages
  - **None**: Just registered, not deployed
  - **Staging**: Being tested (A/B testing)
  - **Production**: Serving real traffic
  - **Archived**: Retired, kept for audit
- **Aliases**: Human-readable names like "champion" or "best_model"

8 Practical MLflow Usage

8.1 The Complete MLflow Workflow

Here's how all four components work together:

```python
import mlflow
from sklearn.model_selection import train_test_split
from xgboost import XGBRegressor

# 1. SETUP: Configure MLflow to use Unity Catalog for model storage
mlflow.set_registry_uri("databricks-uc")

# 2. TRACKING: Start an experiment run
mlflow.set_experiment("/my_experiments/house_price_prediction")

with mlflow.start_run(run_name="xgboost_tuning_v1"):

    # 3. Log parameters (inputs)
    params = {"max_depth": 6, "learning_rate": 0.1, "n_estimators": 100}
    mlflow.log_params(params)

    # 4. Train the model
    model = XGBRegressor(**params)
    model.fit(X_train, y_train)

    # 5. Evaluate and log metrics (outputs)
    predictions = model.predict(X_test)
    rmse = mean_squared_error(y_test, predictions, squared=False)
    mlflow.log_metric("rmse", rmse)

    # 6. MODELS: Log the model with signature (input/output schema)
    signature = mlflow.models.infer_signature(X_test, predictions)
    mlflow.xgboost.log_model(
        model,
        "model",
        signature=signature,
        input_example=X_test[:5]
    )

    # 7. Log additional artifacts
    import matplotlib.pyplot as plt
    # ... create feature importance plot ...
    mlflow.log_figure(fig, "feature_importance.png")

# 8. REGISTRY: Register the best model
model_uri = f"runs:/{mlflow.active_run().info.run_id}/model"
mlflow.register_model(model_uri, "my_catalog.my_schema.house_price_model")

# 9. Load and use the registered model for predictions
```

```
loaded_model = mlflow.pyfunc.load_model(
    "models:/my_catalog.my_schema.house_price_model@best"  # Using alias
)
new_predictions = loaded_model.predict(new_data)
```

Listing 3: Complete MLflow Workflow

8.2 Hyperparameter Tuning with MLflow

When doing hyperparameter tuning (with Optuna, Hyperopt, etc.), use **nested runs**:

```
import optuna

def objective(trial):
    # Each trial is a nested (child) run
    with mlflow.start_run(nested=True):
        params = {
            "max_depth": trial.suggest_int("max_depth", 2, 10),
            "learning_rate": trial.suggest_float("lr", 0.01, 0.3),
        }
        mlflow.log_params(params)

        model = XGBRegressor(**params)
        model.fit(X_train, y_train)
        rmse = evaluate(model, X_val, y_val)
        mlflow.log_metric("rmse", rmse)

        return rmse  # Optuna minimizes this

# Parent run contains all child runs
with mlflow.start_run(run_name="hyperparameter_tuning"):
    study = optuna.create_study(direction="minimize")
    study.optimize(objective, n_trials=50)

    # Log the best parameters
    mlflow.log_params(study.best_params)
    mlflow.log_metric("best_rmse", study.best_value)
```

Listing 4: Hyperparameter Tuning with Nested Runs

8.3 Important Lab Modifications

Lab Environment Modifications

When running the labs in the free Databricks Community Edition:

1. Change Catalog Name: Replace `main.default` with your own catalog (e.g., `cscie103_catalog.default`)

2. Use Local Trials Instead of SparkTrials:

```
# ORIGINAL (may fail in serverless):
```

```
from hyperopt import SparkTrials
trials = SparkTrials(parallelism=4)

# MODIFIED (works everywhere):
from hyperopt import Trials
trials = Trials()  # Single-node execution
```

3. Reduce Trial Count for Faster Testing: Change `n_trials=50` to `n_trials=10` or `max_evals=10`

9 A/B Testing and Model Promotion

9.1 The Champion-Challenger Pattern

When you have a new model, you don't immediately replace the production model. Instead, you use **A/B testing**:

Definition: Champion-Challenger Testing

- **Champion**: The current production model (proven performance)
- **Challenger**: The new model (potentially better, but unproven)

Process:

1. Deploy challenger to **Staging**
2. Route 10-20% of traffic to challenger, 80-90% to champion
3. Monitor performance metrics for both
4. If challenger wins consistently, promote to **Production**
5. Move old champion to **Archived**

```python
from mlflow import MlflowClient

client = MlflowClient()

# Register a new model version
model_version = mlflow.register_model(model_uri, "fraud_detection_model")

# Promote to Staging for A/B testing
client.set_registered_model_alias(
    "fraud_detection_model",
    "challenger",
    model_version.version
)

# After successful A/B testing, promote to Production
client.set_registered_model_alias(
    "fraud_detection_model",
    "champion",  # or "production"
    model_version.version
)

# Archive the old champion
client.delete_registered_model_alias("fraud_detection_model", "old_champion")
```

Listing 5: Model Stage Transitions in MLflow

10 Key Metrics and Evaluation

10.1 Common ML Metrics

Table 5: *Common ML Evaluation Metrics*

Task Type	Metric	When to Use
4***Regression**	RMSE (Root Mean Squared Error)	General purpose, penalizes large errors
	MAE (Mean Absolute Error)	When outliers should have less impact
	R^2 (R-squared)	Explains variance in predictions
	MAPE	When you need percentage error
5***Classification**	Accuracy	Balanced classes only!
	Precision	When false positives are costly (spam detection)
	Recall	When false negatives are costly (disease detection)
	F1 Score	Balance of precision and recall
	AUC-ROC	Overall model discrimination ability

10.2 The Confusion Matrix

For classification, the confusion matrix is essential:

Example: Understanding the Confusion Matrix

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

Key Insight: Different domains care about different quadrants!

- **Medical Diagnosis**: Minimize FN (don't miss cancer!)
- **Spam Filter**: Minimize FP (don't delete important emails!)
- **Fraud Detection**: Balance both (catch fraud but don't block legitimate users)

11 Quick Review: ML Concepts from Previous Lectures

11.1 Scaling Concepts Review

Table 6: *Scaling Terminology Quick Reference*

Term	Definition
Scale Out (Horizontal)	Add more nodes/workers to the cluster
Scale Up (Vertical)	Upgrade to larger instance (more CPU/RAM)
Elasticity	Ability to automatically grow/shrink resources
Availability	Ability to use service at any time (uptime guarantee)
Autoscale	Dynamic adjustment of node count based on load

11.2 ML Algorithm Categories Review

Table 7: *ML Algorithm Categories*

	Supervised	**Unsupervised**
Discrete (Classification)	Classification (spam/not spam)	Clustering (customer segments)
Continuous (Prediction)	Regression (house prices)	Dimensionality Reduction (PCA)

11.3 BI vs. BA Review

- **Business Intelligence (BI)**: "What happened?" — Descriptive, dashboards, reports
- **Business Analytics (BA)**: "Why did it happen? What will happen?" — Predictive, ML models

12 Summary: One-Page Quick Reference

The Problem: ML Reproducibility Crisis

1. Hidden Technical Debt: ML code is the tip of the iceberg. Data management, infrastructure, monitoring are the 90% below water.

2. Fragmented Ecosystem: Python, R, TensorFlow, Spark—too many tools, version conflicts, "worked on my machine" syndrome.

3. Model Drift: Models are living assets that degrade as real-world patterns change.

The Solution: MLOps + MLflow

MLOps: Culture + Practice of automating the entire ML lifecycle.

MLflow: Open-source framework implementing MLOps with 4 components.

MLflow Tracking (Experiment Notebook)

Purpose: Never forget what you did.

What it logs: Parameters, metrics, artifacts, source code.

Magic command: `mlflow.autolog()` — logs everything automatically.

MLflow Projects (Environment Packaging)

Purpose: "Works on any machine."

How: Packages code + `requirements.txt` together.

Run anywhere: `mlflow run <project>` reproduces exact environment.

MLflow Models (Standard Format)

Purpose: Train once, deploy anywhere.

Flavors: `pyfunc`, `sklearn`, `spark`, `pytorch`.

Key benefit: MLE doesn't care what framework DS used.

MLflow Model Registry (Git for Models)

Purpose: Track model versions and lifecycle.

Stages: None → Staging → Production → Archived

Aliases: Human names like "champion", "best_model"

Key Concepts Quick Reference

- **Feature Store**: Central repository for features; prevents training-serving skew
- **Transformer**: Applies fixed rules (`.transform()`) — no learning
- **Estimator**: Learns from data (`.fit()`) — produces Models
- **Model Drift**: Performance degradation over time (Feature, Label, Prediction, Concept)
- **A/B Testing**: Champion vs. Challenger model comparison