Model Monitoring & Observability

Week 14 · CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra

IIT Gandhinagar

The "Silent Failure" Problem

Traditional software fails loudly:

```
>>> items[10]  # List has 5 items
IndexError: list index out of range

>>> response = requests.get(bad_url)
>>> response.status_code
500
```

You know immediately something is wrong.

ML models fail silently:

```
>>> prediction = model.predict(new_data)
>>> prediction
0.85  # Looks fine!

# But the model was trained 6 months ago
# The world has changed
```

Problem: The API returns `200 OK`, but predictions are wrong.

Real-World Monitoring Failures

Case 1: Amazon's Recruiting AI (2018)

- Trained on historical resumes (mostly male)
- Model learned gender bias
- Penalized resumes containing "women's"
- **Failure**: No monitoring of prediction patterns
- **Cost**: Reputational damage, project scrapped

Case 2: Zillow's Zestimate (2021)

- Housing price prediction model
- COVID-19 changed market dynamics
- Model overpaid for homes
- **Failure**: No drift detection on price distributions
- **Cost**: $880M loss, shut down home-buying business

Why ML Models Degrade

1. Data drift (X changes)

- User demographics shift
- New product categories appear
- Image quality degrades
- Seasonal patterns change

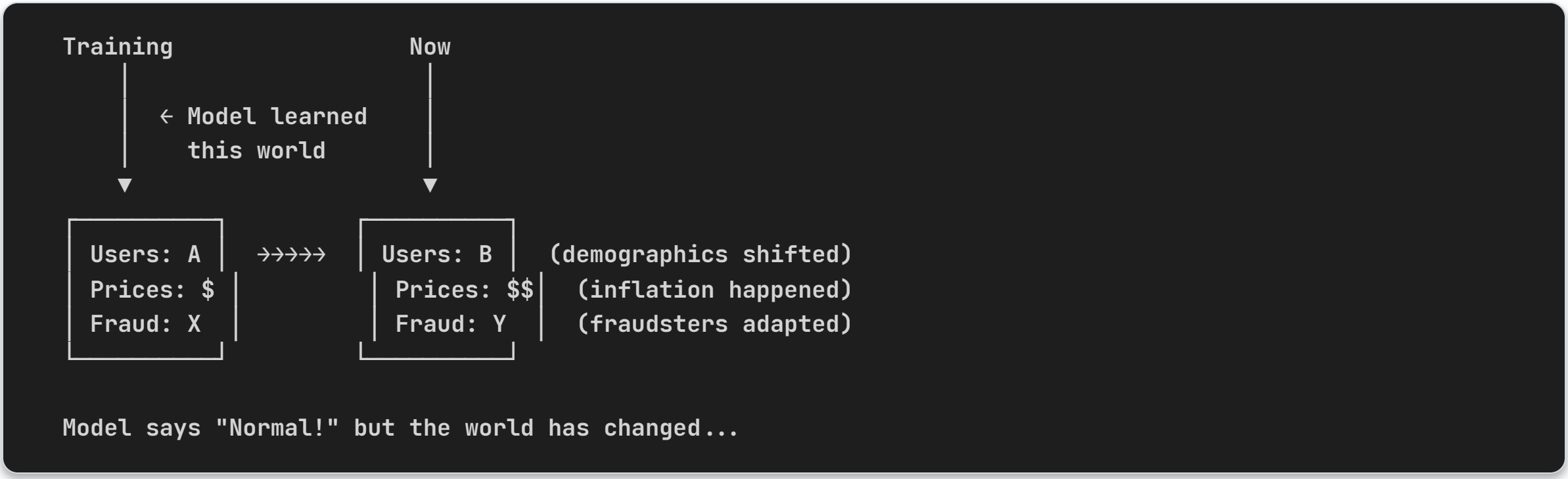
2. Concept drift ($P(Y|X)$ changes)

- COVID changes consumer behavior
- Regulations change (GDPR affects marketing)
- Competitor enters market
- Economic recession alters patterns

3. Operational issues

The World Doesn't Stand Still

Your model is a snapshot of the past. It learned patterns from data collected months ago. But the world keeps changing - users evolve, markets shift, fraudsters adapt. Your model doesn't know any of this. It keeps making predictions based on a world that no longer exists. Without monitoring, you're flying blind.



The ML Monitoring Stack

What to monitor:

1. Model Performance Metrics

- Accuracy, precision, recall, F1
- Latency (p50, p95, p99)
- Throughput (predictions/sec)
- Error rate

2. Data Quality Metrics

- Missing values %
- Out-of-range values
- Cardinality changes
- Schema violations

3. Data Drift Metrics

Types of Drift: Deep Dive

1. Covariate Shift (Data Drift)

- $P(X)$ changes, but $P(Y|X)$ stays the same
- **Example**: Image classifier trained on daylight photos, deployed on nighttime photos
- **Detection**: Compare input distributions
- **Fix**: Retrain with new data or use domain adaptation

2. Prior Probability Shift (Label Drift)

- $P(Y)$ changes, but $P(X|Y)$ stays the same
- **Example**: Fraud rate increases from 1% to 5%
- **Detection**: Monitor label distribution
- **Fix**: Adjust decision threshold or retrain

3. Concept Drift

Types of Drift (continued)

4. Upstream Data Changes

- Data pipeline modification
- **Example**: Feature scaling changed from [0,1] to [-1,1]
- **Detection**: Data schema validation
- **Fix**: Fix pipeline or retrain

5. Virtual Drift (False Alarm)

- Statistical tests flag drift, but performance unchanged
- **Example**: Slight shift in feature with low importance
- **Detection**: Correlate drift with performance degradation
- **Fix**: Ignore or adjust detection threshold

Drift patterns:

Drift Detection: Statistical Tests

For numerical features:

1. Kolmogorov-Smirnov (KS) Test

- Measures maximum distance between CDFs
- **Null hypothesis**: Both samples from same distribution
- **Usage**: `scipy.stats.ks_2samp(reference, current)`
- **Interpretation**: p-value < 0.05 → drift detected

2. Wasserstein Distance (Earth Mover's Distance)

- Measures "cost" to transform one distribution into another
- **Advantage**: Interpretable (same units as feature)
- **Usage**: `scipy.stats.wasserstein_distance(reference, current)`

3. KL Divergence (Kullback-Leibler)

Drift Detection: Statistical Tests (2)

For categorical features:

1. Chi-Square Test

- Tests if category frequencies differ
- **Example**: Distribution of ["mobile", "desktop", "tablet"]
- **Usage**: `scipy.stats.chisquare(observed, expected)`

2. Population Stability Index (PSI)

- Industry standard for monitoring scorecards
- **Formula**: $PSI = \sum(P_i - Q_i) \times \ln(P_i/Q_i)$
- **Interpretation**:
  - PSI < 0.1: No significant change
  - 0.1 < PSI < 0.2: Moderate change
  - PSI > 0.2: Significant change

Drift Detection Example

```python
import numpy as np
from scipy.stats import ks_2samp

# Reference distribution (training data)
reference = np.random.normal(loc=0, scale=1, size=10000)

# Current distribution (production, shifted)
current = np.random.normal(loc=0.5, scale=1.2, size=1000)

# Perform KS test
statistic, p_value = ks_2samp(reference, current)

print(f"KS Statistic: {statistic:.4f}")
print(f"P-value: {p_value:.4f}")
```

Output:

```
KS Statistic: 0.1234
P-value: 0.0001
⚠️
 Drift detected!
```

Monitoring Tools Ecosystem

Open-Source:

- **Evidently AI**: Drift reports, monitoring dashboard
- **Alibi Detect**: Outlier/adversarial/drift detection
- **Great Expectations**: Data validation and profiling
- **Deepchecks**: Testing for ML models & data
- **WhyLabs** (community): Data logging and monitoring

Commercial:

- **Arize AI**: Model observability platform
- **Fiddler AI**: Explainability + monitoring
- **Arthur AI**: Model monitoring + bias detection
- **WhyLabs** (enterprise): Production monitoring
- **Datadog ML Monitoring**: APM + ML metrics

Evidently AI: Quick Start

Installation:

```
pip install evidently
```

Generate drift report:

```
from evidently.report import Report
from evidently.metric_preset import DataDriftPreset, TargetDriftPreset
import pandas as pd

# Load data
reference_data = pd.read_csv("train.csv")
current_data = pd.read_csv("production_last_week.csv")

# Create report
report = Report(metrics=[
    DataDriftPreset(),
    TargetDriftPreset()
])
```


Output: Interactive HTML dashboard with drift analysis per feature.

Evidently AI: Test Suites

Automated testing for data/model:

```
from evidently.test_suite import TestSuite
from evidently.test_preset import DataDriftTestPreset, DataQualityTestPreset

# Define tests
tests = TestSuite(tests=[
    DataDriftTestPreset(),
    DataQualityTestPreset(),
])

tests.run(reference_data=ref_df, current_data=curr_df)

# Check if tests passed
result = tests.as_dict()
if result['summary']['failed_tests'] > 0:
    print("
⚠️
 Tests failed!")
    # Send alert
```

Use case: Run in CI/CD pipeline or scheduled job.

Alibi Detect: Advanced Drift Detection

Multivariate drift detection:

```
from alibi_detect.cd import MMDDrift
import numpy as np

# Reference data (training set)
X_ref = np.random.randn(1000, 10)  # 1000 samples, 10 features

# Initialize detector
drift_detector = MMDDrift(
    X_ref,
    p_val=0.05,
    n_permutations=100
)
```

Advantages:

- Multivariate (considers feature interactions)
- Kernel-based (can detect complex shifts)
- Online updates possible

Model Performance Monitoring

Metrics to track:

Classification:

```python
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Per-day metrics
daily_metrics = {
    'accuracy': [],
    'precision': [],
    'recall': [],
    'f1': []
}

for day in range(30):
    y_true = get_labels_for_day(day)  # Ground truth (delayed)
    y_pred = get_predictions_for_day(day)

    daily_metrics['accuracy'].append(accuracy_score(y_true, y_pred))
    daily_metrics['precision'].append(precision_score(y_true, y_pred))
    daily_metrics['recall'].append(recall_score(y_true, y_pred))
    daily_metrics['f1'].append(f1_score(y_true, y_pred))
```

Model Performance Monitoring (2)

Regression:

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Monitor residuals
residuals = y_true - y_pred

# Track over time
metrics = {
    'mae': mean_absolute_error(y_true, y_pred),
    'rmse': np.sqrt(mean_squared_error(y_true, y_pred)),
    'r2': r2_score(y_true, y_pred),
    'mean_residual': np.mean(residuals),
    'std_residual': np.std(residuals)
}

# Detect drift in residuals
if abs(metrics['mean_residual']) > threshold:
    print("
⚠
```

Key insight: Monitor residual distribution, not just aggregates.

Prediction Distribution Monitoring

Monitor prediction distribution (even without labels!):

```
import numpy as np
import matplotlib.pyplot as plt

# Collect predictions over time
predictions_week1 = model.predict(X_week1)
predictions_week2 = model.predict(X_week2)

# Compare distributions
from scipy.stats import ks_2samp
stat, p_val = ks_2samp(predictions_week1, predictions_week2)

if p_val < 0.05:
    print("
⚠️
 Prediction distribution changed!")

# Visualize
plt.figure(figsize=(10, 4))
plt.hist(predictions_week1, bins=30, alpha=0.5, label='Week 1')
plt.hist(predictions_week2, bins=30, alpha=0.5, label='Week 2')
plt.xlabel('Prediction Score')
plt.ylabel('Frequency')
```


Use case: Detect drift before labels arrive.

Delayed Labels Problem

Challenge: Ground truth arrives late.

Examples:

- **Credit default**: 30-180 days delay
- **Medical diagnosis**: Days to weeks
- **Customer churn**: 30-90 days
- **Ad click**: Minutes (fast!)
- **Purchase conversion**: Hours to days

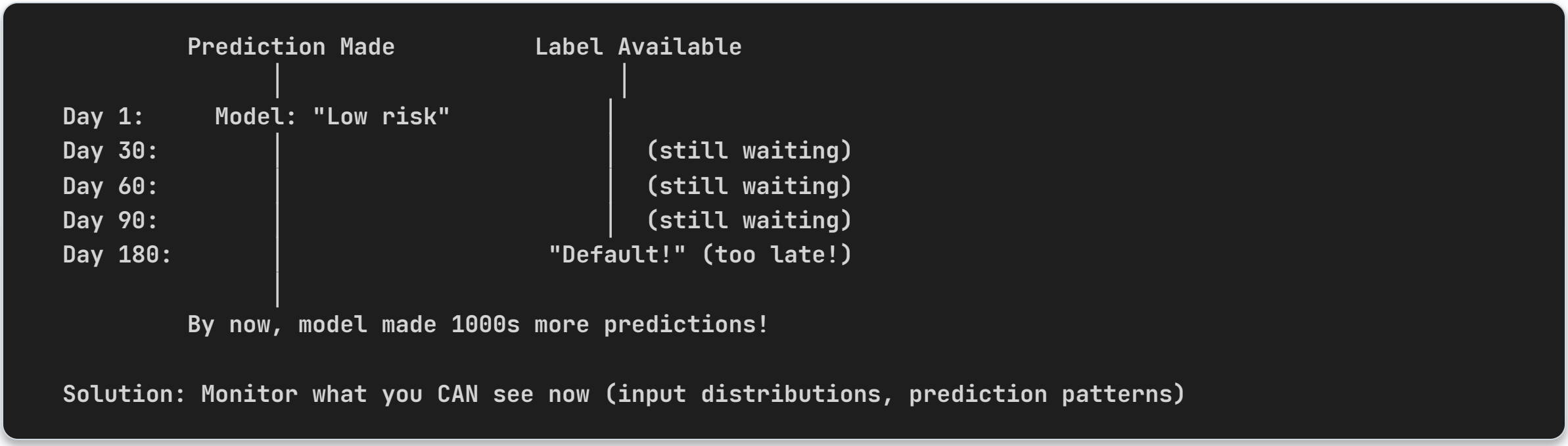
Strategies:

1. Use proxy metrics (fast feedback):

- Click-through rate (proxy for conversion)
- User engagement (proxy for satisfaction)
- Prediction confidence (proxy for accuracy)

The Feedback Loop Time Machine

You can't measure accuracy without labels, but labels arrive too late. Imagine predicting loan defaults - you won't know if you were right for 6 months! By then, your model could have made thousands of bad decisions. This is why we monitor input drift and prediction distributions - they're signals we can see NOW, not 6 months later.



Logging Architecture

What to log:

```
import uuid
from datetime import datetime
import json

def log_prediction(model_id, features, prediction, metadata=None):
    """Log prediction for monitoring."""
    prediction_id = str(uuid.uuid4())

    log_entry = {
        'prediction_id': prediction_id,
        'timestamp': datetime.utcnow().isoformat(),
        'model_id': model_id,
        'model_version': '1.2.3',
        'features': features.tolist(),  # Input features
        'prediction': prediction,
        'prediction_probability': model.predict_proba(features)[0].tolist(),
        'metadata': metadata or {}
    }

    # Write to storage (S3, BigQuery, Postgres)
    write_to_storage(log_entry)

    return prediction_id

def log_feedback(prediction_id, actual_label):
    """Log ground truth when available."""
    feedback_entry = {
        'prediction_id': prediction_id,
        'timestamp': datetime.utcnow().isoformat(),
        'actual_label': actual_label
    }
```

Logging Best Practices

1. Log asynchronously (don't block predictions):

```
import threading

def async_log(log_entry):
    thread = threading.Thread(target=write_to_storage, args=(log_entry,))
    thread.start()

# Or use queue
from queue import Queue
log_queue = Queue()

def log_worker():
    while True:
```

2. Sample for high-volume systems:

```
import random

SAMPLE_RATE = 0.1  # Log 10% of predictions
```

Monitoring Architecture

Architecture components:

- **Logger**: Asynchronous logging of predictions
- **Storage**: S3/GCS for raw data
- **ETL**: Daily/hourly processing (Airflow, dbt)
- **Drift Analysis**: Evidently AI or custom metrics
- **Metrics DB**: Prometheus or InfluxDB
- **Dashboard**: Grafana or Tableau
- **Alerts**: PagerDuty or Slack integration

Prometheus Integration

Expose ML metrics:

```
from prometheus_client import Counter, Histogram, Gauge, start_http_server

# Define metrics
PREDICTIONS_TOTAL = Counter(
    'ml_predictions_total',
    'Total number of predictions',
    ['model_version', 'endpoint']
)

PREDICTION_LATENCY = Histogram(
    'ml_prediction_latency_seconds',
    'Prediction latency in seconds'
)

PREDICTION_SCORE = Histogram(
    'ml_prediction_score',
    'Distribution of prediction scores',
    buckets=[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
)

MODEL_ACCURACY = Gauge(
    'ml_model_accuracy',
    'Current model accuracy'
)

# Start metrics server
```

Prometheus Integration (2)

Instrument prediction function:

```
import time

@PREDICTION_LATENCY.time()
def predict(features):
    # Make prediction
    prediction = model.predict(features)
    score = model.predict_proba(features)[0, 1]

    # Record metrics
    PREDICTIONS_TOTAL.labels(
        model_version='v1.2.3',
        endpoint='/predict'
    ).inc()

    PREDICTION_SCORE.observe(score)

    return prediction, score

# Update accuracy periodically (in background job)
def update_accuracy():
    y_true, y_pred = get_recent_predictions_with_labels()
    accuracy = accuracy_score(y_true, y_pred)
```

Grafana Dashboards

Prometheus queries (PromQL):

1. Prediction rate:

```
rate(ml_predictions_total[5m])
```

2. P95 latency:

```
histogram_quantile(0.95,
```

3. Accuracy over time:

```
ml_model_accuracy
```

4. Prediction score distribution:

```
rate(ml_prediction_score_bucket[5m])
```

Grafana: Connect to Prometheus, create dashboards with these queries.

Alerting Rules

Prometheus alerting rules (`alerts.yml`):

```yaml
groups:
  - name: ml_model_alerts
    interval: 30s
    rules:
      - alert: ModelAccuracyLow
        expr: ml_model_accuracy < 0.85
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "Model accuracy below threshold"
          description: "Accuracy is {{ $value }}, threshold is 0.85"

      - alert: HighPredictionLatency
        expr: histogram_quantile(0.95, rate(ml_prediction_latency_seconds_bucket[5m])) > 0.5
        for: 5m
        labels:
          severity: critical
        annotations:
```

Alerting Integration

Send alerts to Slack:

```python
import requests
import json

def send_slack_alert(message):
    webhook_url = os.getenv("SLACK_WEBHOOK_URL")

    payload = {
        "text": f"
 ML Model Alert: {message}",
        "channel": "#ml-monitoring",
        "username": "ML Monitor Bot"
    }

    response = requests.post(
        webhook_url,
        data=json.dumps(payload),
        headers={'Content-Type': 'application/json'}
    )

    return response.status_code == 200

# Usage
if drift_detected:
    send_slack_alert(
        f"Data drift detected in feature '{feature_name}' "
        f"(p-value: {p_value:.4f})"
```

A/B Testing for Models

Shadow deployment (run new model alongside old):

```
def predict_with_shadow(features):
    # Production model (served to user)
    prediction_v1 = model_v1.predict(features)

    # Shadow model (logged but not served)
    prediction_v2 = model_v2.predict(features)

    # Log both for comparison
    log_predictions(
```

A/B test (split traffic):

```
def predict_with_ab_test(features, user_id):
    # Deterministic assignment (consistent per user)
    variant = 'A' if hash(user_id) % 2 == 0 else 'B'

    if variant == 'A':
        prediction = model_a.predict(features)
```

Model Retraining Triggers

When to retrain:

1. Performance-based:

```
if current_f1 < baseline_f1 * 0.95:  # 5% degradation
```

2. Drift-based:

```
if psi_score > 0.2:  # Significant drift
```

3. Time-based:

```
if days_since_last_training > 30:
```

4. Data-based:

```
if num_new_labeled_samples > 10000:
```

Best practice: Combine multiple triggers with OR logic.

Continuous Training Pipeline

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime, timedelta

default_args = {
    'owner': 'ml-team',
    'depends_on_past': False,
    'start_date': datetime(2024, 1, 1),
    'retries': 1,
}

dag = DAG(
    'model_monitoring_and_retraining',
    default_args=default_args,
    schedule_interval='@daily',
)

def check_drift():
    # Run drift analysis
    drift_detected = analyze_drift()
    if drift_detected:
        return 'retrain_model'
    else:
        return 'skip_retraining'

def retrain_model():
    # Retrain model with new data
    new_model = train(get_recent_data())
    # Evaluate
    if evaluate(new_model) > current_model_score:
        deploy(new_model)

check_drift_task = PythonOperator(
    task_id='check_drift',
    python_callable=check_drift,
    dag=dag,
)

retrain_task = PythonOperator(
    task_id='retrain_model',
    python_callable=retrain_model,
    dag=dag,
)

check_drift_task >> retrain_task
```

Model Versioning and Rollback

Track model versions:

```
import mlflow

# Log model
with mlflow.start_run():
    mlflow.log_params(hyperparameters)
    mlflow.log_metrics({"accuracy": accuracy, "f1": f1})
    mlflow.sklearn.log_model(model, "model")

    # Tag
    mlflow.set_tag("stage", "production")
```

Rollback strategy:

```
def rollback_to_previous_version():
    # Load previous model
    previous_model = load_model_version('v1.2.2')

    # Deploy
    deploy_model(previous_model)
```

Feature Store Monitoring

Monitor feature statistics:

```python
from feast import FeatureStore

store = FeatureStore(repo_path=".")

# Monitor feature freshness
def check_feature_freshness():
    features = store.get_online_features(
        features=['driver:avg_rating', 'driver:completed_trips'],
        entity_rows=[{'driver_id': 123}]
    )

    last_updated = features.metadata.feature_timestamps['driver:avg_rating']
    age_hours = (datetime.now() - last_updated).total_seconds() / 3600

    if age_hours > 24:
        send_alert(f"Feature driver:avg_rating is {age_hours:.1f} hours old!")

# Monitor feature distributions
def monitor_feature_distribution(feature_name):
    values = get_feature_values_last_24h(feature_name)

    ref_mean = REFERENCE_STATS[feature_name]['mean']
    ref_std = REFERENCE_STATS[feature_name]['std']

    current_mean = np.mean(values)

    # Z-score
    z_score = abs(current_mean - ref_mean) / ref_std

    if z_score > 3:  # 3 standard deviations
```

Bias and Fairness Monitoring

Monitor predictions across demographic groups:

```
from fairlearn.metrics import MetricFrame, selection_rate, false_positive_rate

# Compute metrics per group
metric_frame = MetricFrame(
    metrics={
        'accuracy': accuracy_score,
        'fpr': false_positive_rate,
        'selection_rate': selection_rate
    },
    y_true=y_true,
    y_pred=y_pred,
    sensitive_features=df['gender']  # Protected attribute
)

print(metric_frame.by_group)

# Alert if disparity
accuracy_ratio = (
    metric_frame.by_group['accuracy'].min() /
    metric_frame.by_group['accuracy'].max()
)
```

Track over time: Log disparities daily, visualize trends.

Explainability Monitoring

Monitor feature importance drift:

```
from sklearn.inspection import permutation_importance
import numpy as np

# Compute importance on reference data
ref_importance = permutation_importance(
    model, X_ref, y_ref, n_repeats=10
).importances_mean

# Compute importance on current data
curr_importance = permutation_importance(
```

SHAP values monitoring:

```
import shap

# Track mean SHAP values over time
explainer = shap.Explainer(model)
shap_values = explainer(X_prod)
```

Monitoring Dashboard Design

Key dashboard sections:

1. Executive Summary

- Model accuracy (current vs baseline)
- Predictions served today
- Active alerts
- Drift status (✓ / ⚠️ / ❌)

2. Model Performance

- Accuracy/F1 over time (line chart)
- Confusion matrix (heatmap)
- Error rate by category

Monitoring Dashboard Design (2)

5. System Health

- Prediction latency (P50, P95, P99)
- Throughput (predictions/sec)
- Error rate (4xx, 5xx)
- CPU/Memory usage

6. Prediction Analysis

- Prediction distribution
- Confidence score histogram
- Prediction volume by hour/day

7. Business Metrics

- Conversion rate (if applicable)

Incident Response Playbook

When alert fires:

1. Assess severity (5 min)

- Critical: Model down, severe accuracy drop (>20%)
- High: Moderate accuracy drop (10-20%), high latency
- Medium: Data drift detected, minor accuracy drop (<10%)
- Low: Feature freshness issues

2. Initial investigation (15 min)

- Check recent deployments
- Review upstream data pipelines
- Examine input distribution changes
- Check system health (CPU, memory, errors)

3. Mitigation (30 min)

Best Practices Summary

1. Start simple

- Begin with basic metrics (accuracy, latency)
- Add complexity as needed

2. Monitor early and often

- Start monitoring from day 1 of deployment
- Don't wait for problems

3. Automate everything

- Automated drift detection
- Automated retraining pipelines
- Automated alerts

4. Combine multiple signals

Common Pitfalls

1. Alert fatigue

- Too many false positives → people ignore alerts
- **Fix**: Tune thresholds, use multi-signal alerts

2. Over-monitoring

- Monitoring everything → hard to find signal in noise
- **Fix**: Focus on business-critical metrics

3. Ignoring data quality

- Monitoring only model performance → miss root cause
- **Fix**: Monitor data quality upstream

4. No baseline

Lab Preview

Today's lab:

1. **Part 1**: Train baseline model on "clean" data (30 min)
2. **Part 2**: Simulate production drift (gradual shift) (20 min)
3. **Part 3**: Implement drift detection with Evidently (40 min)
4. **Part 4**: Create monitoring dashboard (40 min)
5. **Part 5**: Set up automated alerts (30 min)
6. **Part 6**: Analyze performance degradation (30 min)

Dataset: Bike sharing data (weather features → demand)

Deliverable:

- Drift detection reports
- Monitoring dashboard
- Alert system

Key Takeaways

1. **ML models degrade silently** - monitoring is essential
2. **Monitor multiple dimensions** - performance, drift, data quality, system health
3. **Use statistical tests** - KS test, PSI, Wasserstein distance
4. **Log everything** - inputs, outputs, metadata, feedback
5. **Automate detection and response** - don't rely on manual checks
6. **Start simple, iterate** - basic monitoring > no monitoring
7. **Combine monitoring with CI/CD** - continuous training pipelines

Remember: "You can't improve what you don't measure" - Peter Drucker

Additional Resources

Libraries and Tools:

- Evidently AI: https://evidentlyai.com/
- Alibi Detect: https://github.com/SeldonIO/alibi-detect
- Great Expectations: https://greatexpectations.io/
- Deepchecks: https://deepchecks.com/
- Prometheus: https://prometheus.io/
- Grafana: https://grafana.com/

Reading:

- "Machine Learning Monitoring" (Chip Huyen): https://huyenchip.com/
- "Monitoring Machine Learning Models in Production" (Google Cloud)
- "Made With ML - Monitoring": https://madewithml.com/courses/mlops/monitoring/
- "The ML Test Score" (Breck et al., 2017)

Questions?