Testing & CI/CD for AI

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra, IIT Gandhinagar

Why Testing for ML/AI?

Traditional software: Deterministic, predictable behavior

ML/AI systems: Non-deterministic, data-dependent, complex

Common ML bugs:

- Model performs well in training, fails in production
- Data preprocessing errors
- Model serving issues
- Degradation over time
- Bias and fairness issues

Solution: Comprehensive testing at all levels

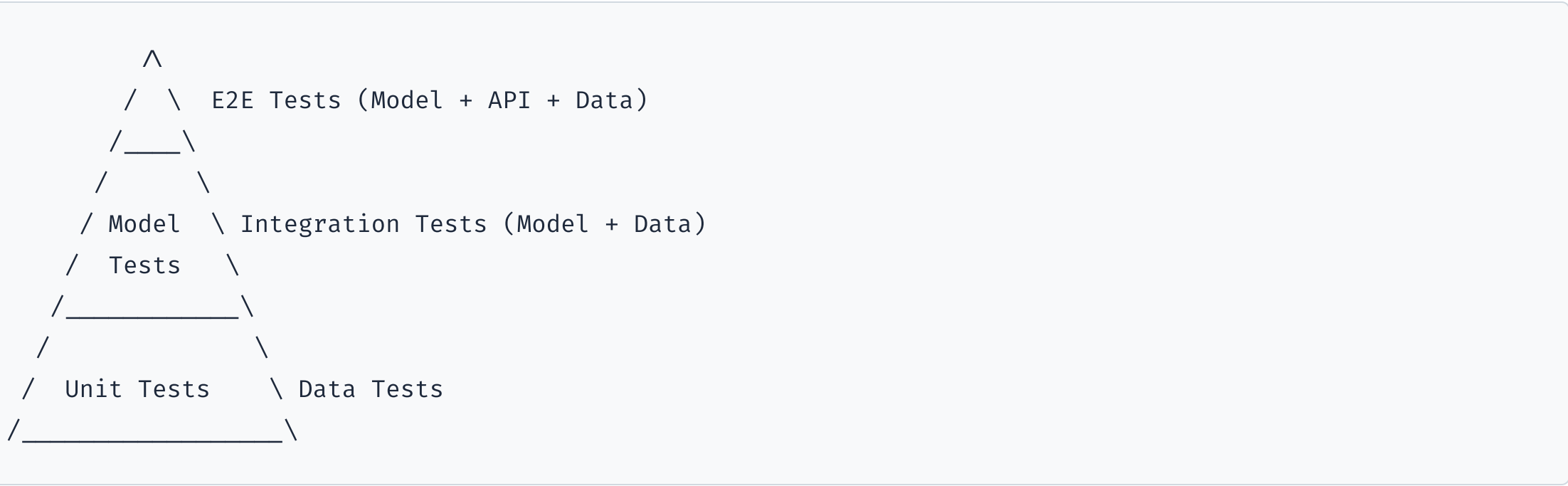
Types of Testing for ML

1. **Code Testing**: Unit, integration, end-to-end
2. **Data Testing**: Schema validation, distribution checks
3. **Model Testing**: Performance, bias, robustness
4. **Infrastructure Testing**: API, deployment, monitoring

Different from traditional:

- Tests might not have deterministic outputs
- Need statistical assertions
- Data quality is critical
- Model behavior changes over time

Testing Pyramid for ML



Bottom (most): Unit tests for functions

Middle: Data validation, model behavior

Top (least): Full pipeline integration

Unit Testing Basics

pytest: Python testing framework

```
# utils.py
def preprocess_text(text):
    return text.lower().strip()

# test_utils.py
import pytest
from utils import preprocess_text

def test_preprocess_text():
    assert preprocess_text("  Hello World  ") == "hello world"
    assert preprocess_text("UPPERCASE") == "uppercase"
    assert preprocess_text("") == ""

def test_preprocess_none():
    with pytest.raises(AttributeError):
        preprocess_text(None)
```

Run tests:

pytest Fixtures

Reusable test data and setup

```
import pytest
import pandas as pd

@pytest.fixture
def sample_data():
    """Create sample dataset for testing."""
    return pd.DataFrame({
        'text': ['hello', 'world', 'test'],
        'label': [0, 1, 0]
    })

@pytest.fixture
def trained_model():
    """Load pretrained model."""
    from sklearn.linear_model import LogisticRegression
    model = LogisticRegression()
    # Load or train model
    return model

def test_data_shape(sample_data):
    assert sample_data.shape == (3, 2)

def test_model_predict(trained_model, sample_data):
    predictions = trained_model.predict(sample_data[['text']])
    assert len(predictions) == len(sample_data)
```

Parametrized Tests

Test multiple inputs efficiently

```
import pytest

@pytest.mark.parametrize("input_text,expected", [
    ("Hello", "hello"),
    ("WORLD", "world"),
    ("  Test  ", "test"),
    ("123", "123"),
])
def test_preprocess_parametrized(input_text, expected):
    assert preprocess_text(input_text) == expected

@pytest.mark.parametrize("model_name", [
    "logistic_regression",
    "random_forest",
    "svm",
])
def test_model_training(model_name):
    model = train_model(model_name)
    assert model is not None
    assert hasattr(model, 'predict')
```

Testing Data Pipelines

Validate data at each stage

```
import pandas as pd
import pytest

def test_data_loading():
    df = load_data("data/train.csv")
    assert df is not None
    assert len(df) > 0
    assert 'text' in df.columns
    assert 'label' in df.columns

def test_data_preprocessing():
    raw_df = pd.DataFrame({'text': ['Hello', None, '']})
    processed_df = preprocess_data(raw_df)

    # No null values
    assert processed_df['text'].isnull().sum() == 0

    # No empty strings
    assert (processed_df['text'] == '').sum() == 0

def test_data_split():
    X_train, X_test, y_train, y_test = split_data(df)

    # Check sizes
    assert len(X_train) + len(X_test) == len(df)

    # Check no overlap
    train_indices = set(X_train.index)
    test_indices = set(X_test.index)
    assert len(train_indices & test_indices) == 0
```

Data Validation with Great Expectations

Test data quality automatically

```
import great_expectations as ge

# Create expectation suite
df = ge.read_csv("data/train.csv")

# Set expectations
df.expect_column_to_exist("text")
df.expect_column_to_exist("label")

df.expect_column_values_to_not_be_null("text")
df.expect_column_values_to_be_in_set("label", [0, 1])

df.expect_column_mean_to_be_between(
    "text_length",
    min_value=10,
    max_value=500
)

# Validate
results = df.validate()
print(f"Success: {results['success']}")
```

Schema Validation with Pydantic

Ensure data structure consistency

```
from pydantic import BaseModel, validator
from typing import List

class TrainingData(BaseModel):
    texts: List[str]
    labels: List[int]

    @validator('texts')
    def texts_not_empty(cls, v):
        assert all(len(text) > 0 for text in v), "Empty texts found"
        return v

    @validator('labels')
    def labels_valid(cls, v):
        assert all(label in [0, 1] for label in v), "Invalid labels"
        return v

# Use in tests
def test_data_schema():
    data = load_training_data()
    validated_data = TrainingData(**data)
    assert len(validated_data.texts) == len(validated_data.labels)
```

Testing Model Training

Verify model can learn

```
def test_model_trains():
    X_train, y_train = create_dummy_data(n_samples=100)
    model = LogisticRegression()

    # Train model
    model.fit(X_train, y_train)

    # Check it learned something
    train_accuracy = model.score(X_train, y_train)
    assert train_accuracy > 0.5, "Model failed to learn"

def test_model_overfits_small_data():
    """Model should overfit on very small data."""
    X_train, y_train = create_dummy_data(n_samples=10)
    model = DecisionTreeClassifier(max_depth=10)
    model.fit(X_train, y_train)

    train_accuracy = model.score(X_train, y_train)
    assert train_accuracy > 0.95, "Model should overfit small dataset"
```

Testing Model Performance

Statistical assertions

```
import numpy as np

def test_model_accuracy_threshold():
    model = load_model("models/classifier.pkl")
    X_test, y_test = load_test_data()

    accuracy = model.score(X_test, y_test)
    assert accuracy ⩾ 0.85, f"Accuracy {accuracy} below threshold"

def test_model_predictions_distribution():
    """Check prediction distribution is reasonable."""
    predictions = model.predict(X_test)

    # Should predict both classes
    assert len(np.unique(predictions)) > 1

    # Distribution shouldn't be too skewed
    class_0_ratio = (predictions == 0).mean()
    assert 0.2 ⩽ class_0_ratio ⩽ 0.8

def test_model_confidence():
    """Predictions should be confident."""
    probas = model.predict_proba(X_test)
    max_probas = probas.max(axis=1)

    # At least 80% of predictions should be > 0.7 confidence
    confident = (max_probas > 0.7).mean()
    assert confident ⩾ 0.8
```

Testing Model Invariance

Model should be robust to expected variations

```
def test_model_case_invariance():
    """Model should be invariant to case."""
    text_lower = "hello world"
    text_upper = "HELLO WORLD"

    pred_lower = model.predict([text_lower])[0]
    pred_upper = model.predict([text_upper])[0]

    assert pred_lower == pred_upper

def test_model_whitespace_invariance():
    """Extra whitespace shouldn't change prediction."""
    text1 = "hello world"
    text2 = "hello  world"  # Extra space

    pred1 = model.predict([text1])[0]
    pred2 = model.predict([text2])[0]

    assert pred1 == pred2

def test_model_typo_robustness():
    """Small typos shouldn't drastically change prediction."""
    text1 = "this is great"
    text2 = "this is grate"  # Typo

    prob1 = model.predict_proba([text1])[0]
    prob2 = model.predict_proba([text2])[0]

    # Predictions should be similar
    assert np.abs(prob1 - prob2).max() < 0.3
```

Testing for Bias

Ensure fairness across groups

```
def test_model_gender_bias():
    """Model shouldn't discriminate by gender."""
    texts_male = ["He is a great engineer"]
    texts_female = ["She is a great engineer"]

    prob_male = model.predict_proba(texts_male)[0]
    prob_female = model.predict_proba(texts_female)[0]

    # Predictions should be similar
    assert np.abs(prob_male - prob_female).max() < 0.1

def test_model_fairness_metrics():
    """Check demographic parity."""
    from sklearn.metrics import confusion_matrix

    for group in ['group_a', 'group_b']:
        X_group, y_group = get_test_data(group)
        y_pred = model.predict(X_group)

        tn, fp, fn, tp = confusion_matrix(y_group, y_pred).ravel()
        fpr = fp / (fp + tn)  # False positive rate

        # FPR should be similar across groups
        assert fpr < 0.15, f"High FPR for {group}: {fpr}"
```

Testing Model Robustness

Adversarial and edge cases

```
def test_model_on_empty_input():
    """Handle empty input gracefully."""
    with pytest.raises(ValueError):
        model.predict([""])

def test_model_on_long_input():
    """Handle very long input."""
    long_text = "word " * 10000
    pred = model.predict([long_text])
    assert pred is not None

def test_model_on_special_chars():
    """Handle special characters."""
    special_texts = [
        "Hello!!! @#$%",
        "test 123 456",
        "émoji 😀 test"
    ]
    preds = model.predict(special_texts)
    assert len(preds) == len(special_texts)

def test_model_numerical_stability():
    """Check for NaN/Inf in predictions."""
    preds = model.predict_proba(X_test)
    assert not np.isnan(preds).any()
    assert not np.isinf(preds).any()
```

Testing Model Serialization

Ensure models save/load correctly

```
import joblib
import tempfile

def test_model_serialization():
    """Model should serialize and deserialize correctly."""
    original_model = train_model()
    X_test, y_test = load_test_data()

    # Get original predictions
    preds_before = original_model.predict(X_test)

    # Save and load
    with tempfile.NamedTemporaryFile(suffix='.pkl') as f:
        joblib.dump(original_model, f.name)
        loaded_model = joblib.load(f.name)

    # Compare predictions
    preds_after = loaded_model.predict(X_test)
    np.testing.assert_array_equal(preds_before, preds_after)

def test_model_version_compatibility():
    """Check model works with different library versions."""
    model = load_model("models/old_model.pkl")
    assert model is not None
    # Test basic prediction works
    pred = model.predict(X_test[:1])
    assert pred is not None
```

Testing API Endpoints

FastAPI testing with TestClient

```python
from fastapi.testclient import TestClient
from main import app

client = TestClient(app)

def test_health_endpoint():
    response = client.get("/health")
    assert response.status_code == 200
    assert response.json()["status"] == "healthy"

def test_predict_endpoint():
    response = client.post(
        "/predict",
        json={"text": "This is a test"}
    )
    assert response.status_code == 200
    assert "prediction" in response.json()
    assert "confidence" in response.json()

def test_predict_invalid_input():
    response = client.post("/predict", json={})
    assert response.status_code == 422  # Validation error

def test_predict_batch():
    response = client.post(
        "/predict/batch",
        json={"texts": ["text1", "text2", "text3"]}
    )
    assert response.status_code == 200
    assert len(response.json()["predictions"]) == 3
```

Integration Testing

Test components working together

```python
def test_full_pipeline():
    """Test complete ML pipeline."""
    # Load data
    df = load_data("data/test.csv")
    assert df is not None

    # Preprocess
    processed = preprocess_data(df)
    assert len(processed) == len(df)

    # Load model
    model = load_model("models/classifier.pkl")
    assert model is not None

    # Predict
    predictions = model.predict(processed['text'])
    assert len(predictions) == len(processed)

    # Evaluate
    accuracy = (predictions == df['label']).mean()
    assert accuracy >= 0.8

def test_training_pipeline():
    """Test training from scratch."""
    # This might be slow, mark as integration test
    df = load_data("data/train_small.csv")
    model = train_full_pipeline(df)

    assert model is not None
    assert hasattr(model, 'predict')

    # Test on validation set
    val_df = load_data("data/val_small.csv")
    accuracy = evaluate_model(model, val_df)
    assert accuracy >= 0.7
```

Continuous Integration (CI)

Automate testing on every commit

Benefits:

- Catch bugs early
- Ensure code quality
- Prevent regressions
- Document build/test process
- Enable collaboration

Popular CI platforms:

- GitHub Actions
- GitLab CI
- CircleCI
- Travis CI

GitHub Actions Basics

Define workflows in `.github/workflows/`

```yaml
# .github/workflows/test.yml
name: Tests

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.10'

    - name: Install dependencies
      run: |
        pip install -r requirements.txt
        pip install pytest pytest-cov

    - name: Run tests
      run: |
        pytest tests/ -v --cov=src --cov-report=xml

    - name: Upload coverage
      uses: codecov/codecov-action@v3
```

CI for ML: Data Testing

```yaml
name: Data Validation

on:
  schedule:
    - cron: '0 2 * * *'  # Daily at 2 AM
  workflow_dispatch:  # Manual trigger

jobs:
  validate-data:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.10'

    - name: Install dependencies
      run: pip install great-expectations pandas

    - name: Download latest data
      run: python scripts/download_data.py

    - name: Validate data schema
      run: python scripts/validate_data.py

    - name: Check data quality
      run: great_expectations checkpoint run data_quality

    - name: Alert on failure
      if: failure()
      uses: actions/github-script@v6
      with:
        script: |
          github.rest.issues.create({
            owner: context.repo.owner,
            repo: context.repo.repo,
            title: 'Data validation failed',
            body: 'Data quality checks failed. Please investigate.'
          })
```

CI for ML: Model Training

```yaml
name: Train Model

on:
  push:
    branches: [ main ]
    paths:
      - 'src/**'
      - 'configs/**'

jobs:
  train:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.10'

    - name: Install dependencies
      run: pip install -r requirements.txt

    - name: Download data
      run: dvc pull

    - name: Train model
      run: python train.py --config configs/config.yaml

    - name: Evaluate model
      run: python evaluate.py

    - name: Check performance threshold
      run: |
        accuracy=$(python -c "import json; print(json.load(open('metrics.json'))['accuracy'])")
        if (( $(echo "$accuracy < 0.85" | bc -l) )); then
          echo "Accuracy $accuracy below threshold"
          exit 1
        fi

    - name: Upload model artifact
      uses: actions/upload-artifact@v3
      with:
        name: model
        path: models/
```

CI: Matrix Testing

Test multiple configurations

```yaml
name: Matrix Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, macos-latest, windows-latest]
        python-version: ['3.9', '3.10', '3.11']

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v4
      with:
        python-version: ${{ matrix.python-version }}

    - name: Install dependencies
      run: pip install -r requirements.txt

    - name: Run tests
      run: pytest tests/
```

CI: Caching Dependencies

Speed up CI with caching

```yaml
jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.10'
        cache: 'pip'  # Cache pip dependencies

    - name: Cache models
      uses: actions/cache@v3
      with:
        path: models/
        key: models-${{ hashFiles('models/**') }}

    - name: Install dependencies
      run: pip install -r requirements.txt

    - name: Run tests
      run: pytest tests/
```

CI: Secrets Management

Store API keys and credentials securely

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Deploy model
      env:
        API_KEY: ${{ secrets.API_KEY }}
        AWS_ACCESS_KEY: ${{ secrets.AWS_ACCESS_KEY }}
        AWS_SECRET_KEY: ${{ secrets.AWS_SECRET_KEY }}
      run: |
        python deploy.py --api-key $API_KEY
```

Add secrets in GitHub:

Settings → Secrets and variables → Actions → New repository secret

Pre-commit Hooks

Run checks before committing

Install:

```
pip install pre-commit
```

Create `.pre-commit-config.yaml`:

```
repos:
  - repo: https://github.com/psf/black
    rev: 23.3.0
    hooks:
      - id: black

  - repo: https://github.com/pycqa/flake8
    rev: 6.0.0
    hooks:
      - id: flake8
        args: [--max-line-length=88]

  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.4.0
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
      - id: check-yaml
```

Continuous Deployment (CD)

Automate model deployment

```yaml
name: Deploy Model

on:
  push:
    tags:
      - 'v*'  # Trigger on version tags

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Build Docker image
      run: docker build -t my-model:${{ github.ref_name }} .

    - name: Push to registry
      run: |
        echo ${{ secrets.DOCKER_PASSWORD }} | docker login -u ${{ secrets.DOCKER_USERNAME }} --password-stdin
        docker push my-model:${{ github.ref_name }}

    - name: Deploy to production
      run: |
        ssh ${{ secrets.PROD_SERVER }} "docker pull my-model:${{ github.ref_name }} && docker restart model-service"

    - name: Health check
      run: |
        sleep 10
        curl -f https://api.example.com/health || exit 1
```

Model Monitoring in Production

Track model performance over time

```python
import mlflow

def log_prediction(input_data, prediction, actual=None):
    """Log predictions for monitoring."""
    mlflow.log_metric("prediction_count", 1)

    if actual is not None:
        correct = (prediction == actual)
        mlflow.log_metric("accuracy", float(correct))

def monitor_data_drift():
    """Check if input distribution changed."""
    from scipy.stats import ks_2samp

    # Compare current data to training data
    current_features = load_current_features()
    training_features = load_training_features()

    for col in current_features.columns:
        statistic, p_value = ks_2samp(
            current_features[col],
            training_features[col]
        )

        if p_value < 0.05:
            alert(f"Data drift detected in {col}")
            mlflow.log_metric(f"drift_{col}_pvalue", p_value)
```

A/B Testing for Models

Compare model versions in production

```
import random

def get_model_version(user_id):
    """Assign users to model versions."""
    # Deterministic assignment based on user_id
    if hash(user_id) % 10 < 5:  # 50% split
        return "model_v1"
    else:
        return "model_v2"

def predict_with_ab(user_id, input_data):
    model_version = get_model_version(user_id)
    model = load_model(model_version)

    prediction = model.predict(input_data)

    # Log for analysis
    log_ab_prediction(user_id, model_version, prediction)

    return prediction

def analyze_ab_results():
    """Compare model versions."""
    v1_metrics = get_metrics("model_v1")
    v2_metrics = get_metrics("model_v2")

    print(f"Model V1 accuracy: {v1_metrics['accuracy']}")
    print(f"Model V2 accuracy: {v2_metrics['accuracy']}")

    # Statistical test
    from scipy.stats import ttest_ind
    _, p_value = ttest_ind(v1_metrics['scores'], v2_metrics['scores'])

    if p_value < 0.05:
        print("Significant difference detected")
```

Shadow Deployment

Run new model alongside old without affecting users

```
def predict_with_shadow(input_data):
    # Production model
    prod_model = load_model("production")
    prod_pred = prod_model.predict(input_data)

    # Shadow model (new version)
    try:
        shadow_model = load_model("shadow")
        shadow_pred = shadow_model.predict(input_data)

        # Log comparison
        log_shadow_comparison(prod_pred, shadow_pred)
    except Exception as e:
        log_error(f"Shadow model error: {e}")

    # Always return production prediction
    return prod_pred
```

Testing Best Practices for ML

1. Test at multiple levels

- Unit: Individual functions
- Integration: Components together
- System: Full pipeline

2. Use fixtures for test data

- Small, representative datasets
- Edge cases and corner cases

3. Mock external dependencies

- APIs, databases
- Speed up tests

4. Set random seeds

- Make tests deterministic

Code Coverage

Measure test coverage

```
# Install coverage tool
pip install pytest-cov

# Run with coverage
pytest --cov=src --cov-report=html tests/

# View report
open htmlcov/index.html
```

Good coverage targets:

- Utility functions: 90%+
- Data processing: 80%+
- Model training: 60%+ (some parts hard to test)

Note: High coverage doesn't guarantee quality tests!

Performance Testing

Ensure models meet latency requirements

```python
import time
import pytest

def test_prediction_latency():
    """Prediction should be fast."""
    model = load_model()
    X_test = load_test_data(n=1000)

    start = time.time()
    predictions = model.predict(X_test)
    elapsed = time.time() - start

    latency_per_sample = elapsed / len(X_test) * 1000  # ms

    assert latency_per_sample < 10, f"Latency too high: {latency_per_sample}ms"

def test_batch_prediction_efficiency():
    """Batch should be faster than individual predictions."""
    X_test = load_test_data(n=100)

    # Individual predictions
    start = time.time()
    for x in X_test:
        model.predict([x])
    individual_time = time.time() - start

    # Batch prediction
    start = time.time()
    model.predict(X_test)
    batch_time = time.time() - start

    # Batch should be at least 2x faster
    assert batch_time < individual_time / 2
```

What We've Learned

Testing for ML:

- Unit tests for functions and utilities
- Data validation with Great Expectations
- Model testing: performance, robustness, bias
- API testing with TestClient
- Integration testing for pipelines

CI/CD:

- GitHub Actions for automated testing
- Matrix testing across configurations
- Continuous deployment workflows
- Pre-commit hooks

Production:

Resources

Testing:

- pytest: https://docs.pytest.org/
- Great Expectations: https://greatexpectations.io/
- Hypothesis (property testing): https://hypothesis.readthedocs.io/

CI/CD:

- GitHub Actions: https://docs.github.com/en/actions
- CircleCI: https://circleci.com/docs/
- GitLab CI: https://docs.gitlab.com/ee/ci/

ML Testing:

- "Testing ML Code" (Google): https://developers.google.com/machine-learning/testing-debugging
- "Effective Testing for Machine Learning": https://www.jeremyjordan.me/testing-

Questions?

Next: Model Deployment

Lab: Write tests, set up CI/CD pipeline