

Lab 2: Automated Training and Metric Reporting Using GitHub Actions

Student: Bhanu Reddy

Roll Number: 2022bcd0026

Course: MLOps

Lab: Lab 2 - CI-Driven ML Workflows with GitHub Actions

📋 Objective

This lab introduces **CI-driven machine learning workflows** using GitHub Actions. Students manually modify model code for each experiment, push changes to GitHub, and use GitHub Actions to:

- Automatically train the model
- Compute evaluation metrics (MSE, R² Score)
- Display metrics in GitHub Actions Job Summary
- Store trained models and results as workflow artifacts

This demonstrates how automation improves reproducibility and traceability in ML experiments.

📁 Project Structure

```
lab-2/
├── .github/
│   └── workflows/
│       └── train-model.yml      # GitHub Actions workflow
├── train.py                  # Training script (modify for experiments)
├── requirements.txt          # Python dependencies
└── .gitignore                # Git ignore patterns
└── model.pkl                 # Trained model (generated by workflow)
└── results.json              # Evaluation results (generated by workflow)
└── README.md                 # This file
```

⌚ Dataset

Wine Quality Dataset (Red Wine)

- **Source:** <https://archive.ics.uci.edu/dataset/186/wine+quality>
- **Features:** 11 physicochemical properties
- **Target:** Quality score (0-10)
- **Samples:** 1,599 red wine samples

Same dataset from Lab 1 for consistency.

Evaluation Metrics

All experiments compute and report:

1. **Mean Squared Error (MSE)** - Lower is better
2. **R² Score** - Higher is better (closer to 1)

Metrics are:

- Printed by the training script
 - Displayed in GitHub Actions Job Summary
 - Saved to `results.json`
-

How It Works

1. Modify Training Script

Edit `train.py` to change:

- Model type (`LinearRegression` or `RandomForest`)
- Hyperparameters
- Preprocessing options (scaling, feature selection)
- Train-test split ratio

2. Commit & Push

```
git add train.py
git commit -m "Experiment: LinearRegression, scaling=True, test_size=0.2"
git push origin main
```

3. Automated Training

GitHub Actions automatically:

- Sets up Python environment
- Installs dependencies
- Runs `train.py`
- Captures metrics
- Uploads artifacts

4. View Results

- Check **Job Summary** for metrics
 - Download **model.pkl** and **results.json** from Artifacts
-

Setup Instructions

Step 1: Create GitHub Repository

1. **Create GitHub account:** [2022bcd0026_bhanureddy](https://github.com/2022bcd0026_bhanureddy) (use institute email)
2. **Create repository:** [lab2](#) (public)
3. **Clone repository:**

```
git clone https://github.com/2022bcd0026_bhanureddy/lab2.git  
cd lab2
```

Step 2: Upload Project Files

Copy all files from this folder to your repository:

```
# Copy .github folder  
# Copy train.py  
# Copy requirements.txt  
# Copy .gitignore  
# Copy README.md
```

Step 3: Push to GitHub

```
git add .  
git commit -m "Initial setup: Lab 2 project structure"  
git push origin main
```

Step 4: Verify Workflow

1. Go to **Actions** tab in GitHub
2. You should see workflow run automatically
3. Click on the run to see Job Summary with metrics
4. Download artifacts from the run page

Running Experiments

Experiment Configuration

Edit [train.py](#) and modify these variables:

```
# Model Selection  
MODEL_TYPE = 'LinearRegression' # or 'RandomForest'
```

```

# Train-Test Split
TEST_SIZE = 0.20 # 0.20 = 80/20 split

# Preprocessing
USE_SCALING = False # True to apply StandardScaler

# Feature Selection
FEATURE_SELECTION = None # None or integer (e.g., 8)

# RandomForest Hyperparameters
RF_N_ESTIMATORS = 100
RF_MAX_DEPTH = None # None for unlimited

```

Example Experiments (from Lab 1)

Experiment 1: LR-1

```

MODEL_TYPE = 'LinearRegression'
TEST_SIZE = 0.20
USE_SCALING = False
FEATURE_SELECTION = None

```

Commit: "Experiment LR-1: LinearRegression, no scaling, all features, 80/20"

Experiment 2: LR-2

```

MODEL_TYPE = 'LinearRegression'
TEST_SIZE = 0.30
USE_SCALING = True
FEATURE_SELECTION = None

```

Commit: "Experiment LR-2: LinearRegression, scaled, all features, 70/30"

Experiment 3: RF-1

```

MODEL_TYPE = 'RandomForest'
TEST_SIZE = 0.20
USE_SCALING = False
FEATURE_SELECTION = None
RF_N_ESTIMATORS = 50
RF_MAX_DEPTH = 10

```

Commit: "Experiment RF-1: RandomForest, 50 trees, depth=10, 80/20"

Artifacts

Each workflow run produces:

1. Trained Model (`model.pkl`)

- Pickled scikit-learn model
- Can be loaded and used for predictions
- Downloadable from Actions run page

2. Results JSON (`results.json`)

```
{  
  "student": "Bhanu Reddy",  
  "roll_number": "2022bcd0026",  
  "lab": "Lab 2 - Automated Training with GitHub Actions",  
  "experiment_config": {  
    "model_type": "LinearRegression",  
    "test_size": 0.2,  
    "use_scaling": false,  
    "feature_selection": null  
  },  
  "metrics": {  
    "MSE": 0.4235,  
    "R2_Score": 0.3045  
  }  
}
```

Testing Locally

Before pushing, test locally:

```
# Install dependencies  
pip install -r requirements.txt  
  
# Run training script  
python train.py  
  
# Check outputs  
ls model.pkl results.json
```

Required Screenshots for Submission

1. **GitHub Actions runs** showing all experiments
 2. **Job Summary** with metrics (must include name & roll number)
 3. **Artifacts** downloadable for each run
 4. **Commit history** showing meaningful commit messages
-

🔍 Analysis Questions

1. How did GitHub Actions improve experiment reproducibility?

- Every experiment runs in a clean, isolated environment
- Dependencies are installed fresh each time
- Exact Python version and library versions are consistent
- Anyone can reproduce results by checking out the same commit

2. How easy was it to compare results across runs?

- GitHub Actions UI shows all runs in chronological order
- Job Summary displays metrics immediately
- Commit messages help identify experiments
- Artifacts can be downloaded and compared

3. What role does Git commit history play in experiment tracking?

- Each commit represents one experiment configuration
- Commit messages describe what changed
- Easy to see evolution of experiments
- Can checkout any commit to reproduce exact experiment

4. Benefits compared to Lab 1

- **Automation:** No manual execution needed
- **Consistency:** Same environment every time
- **Traceability:** Git history tracks all changes
- **Accessibility:** Results accessible from anywhere via GitHub
- **Collaboration:** Team members can see all experiments

5. Limitations of this approach

- **Manual changes:** Still need to edit code for each experiment
 - **No parameter sweep:** Can't easily run multiple configs at once
 - **Limited comparison:** Hard to compare metrics side-by-side
 - **No visualization:** Metrics are just numbers in JSON
 - **Version control overhead:** Many commits for experiments
-

🎓 Key Learnings

1. **CI/CD for ML:** Automated training pipelines with GitHub Actions
-

2. **Reproducibility:** Clean environments ensure consistent results
 3. **Version Control:** Git tracks experiment history
 4. **Artifact Management:** Models and results stored as artifacts
 5. **Automation Benefits:** Reduced manual work, increased consistency
-

Deliverables

Submit on your assignment portal:

1. GitHub repository link
 2. Screenshots:
 - Job summary with metrics for ALL experiments
 - Downloadable artifacts for ALL experiments
 - Must show your name and roll number
 3. Answers to all 5 analysis questions
-

Contact

Student: Bhanu Reddy

Email: bhanu22bcd026@iiitkottayam.ac.in

Roll Number: 2022bcd0026

Date: February 2026

Lab: Lab 2 - Automated Training with GitHub Actions

CI/CD Pipeline Stages

The Jenkins pipeline ([Jenkinsfile](#)) includes the following stages:

1. Checkout

- Retrieves source code from repository
- Displays student and project information

2. Setup Environment

- Checks Python version
- Creates virtual environment
- Installs ML dependencies (pandas, scikit-learn, numpy)

3. Code Quality Check

- Validates Python syntax for all files
- Compiles Python files to check for errors

4. Run Unit Tests

- Executes 13 comprehensive unit tests
- Validates ML pipeline functionality
- Provides verbose test output

5. Train ML Models

- Loads Wine Quality dataset
- Trains 8 ML experiments:
 - **LR-1 to LR-4:** Linear Regression with various configurations
 - **RF-1 to RF-4:** Random Forest with different hyperparameters
- Evaluates models using MSE and R² metrics
- Generates results.json

6. Validate Results

- Verifies results.json was created
- Displays results summary
- Validates output format

7. Archive Artifacts

- Archives Python files and results.json
- Creates fingerprints for tracking

💻 How to Use

Running Locally

1. Install dependencies:

```
pip install pandas scikit-learn numpy
```

2. Run the ML pipeline:

```
python app.py
```

This will:

- Download the Wine Quality dataset
- Train 8 ML models (4 Linear Regression + 4 Random Forest)
- Display results in console
- Generate **results.json** with detailed metrics

3. Run unit tests:

```
python -m unittest test_app.py -v
```

4. Using virtual environment (recommended):

```
python -m venv venv
source venv/bin/activate # On Windows: venv\Scripts\activate
pip install -r requirements.txt
python app.py
```

🔧 Setting Up in Jenkins

Step 1: Access Jenkins

- Navigate to: <http://localhost:8080>
- Login with credentials (username: **2022bcd0026**)

Step 2: Create New Pipeline Job

1. Click "**New Item**"
2. Enter name: **2022bcd0026_wine_quality_ml_pipeline**
3. Select "**Pipeline**"
4. Click "**OK**"

Step 3: Configure Pipeline

Option A: Pipeline from SCM (Recommended)

1. In Pipeline section, select "**Pipeline script from SCM**"
2. Choose "**Git**" as SCM
3. Enter your repository URL
4. Set **Script Path** to: **lab-2/Jenkinsfile**

Option B: Direct Pipeline Script

1. In Pipeline section, select "**Pipeline script**"
2. Copy the contents of **Jenkinsfile** into the script box

Step 4: Run the Pipeline

1. Click "**Build Now**"
2. Watch the pipeline stages execute
3. Click on build number to view console output

📊 Expected Pipeline Output

✓ Checkout - Source code retrieved
✓ Setup Environment - Virtual environment created, ML libraries installed
✓ Code Quality Check - Syntax validated for app.py and test_app.py
✓ Run Unit Tests - All 13 tests passed
✓ Train ML Models - 8 experiments completed:
 LR-1: MSE=0.4235, R²=0.3045
 LR-2: MSE=0.4147, R²=0.3186
 LR-3: MSE=0.4204, R²=0.3096
 LR-4: MSE=0.3990, R²=0.3344
 RF-1: MSE=0.3347, R²=0.4487
 RF-2: MSE=0.3325, R²=0.4537
 RF-3: MSE=0.3175, R²=0.4419 ← Best MSE
 RF-4: MSE=0.3299, R²=0.4337
✓ Validate Results - results.json generated and validated
✓ Archive Artifacts - Files archived

PIPELINE SUCCESS

Best Model: RF-3 (Random Forest, MSE: 0.3175)

📝 Test Results

The test suite includes 13 comprehensive test methods:

- **test_initialization** - Verify predictor initialization
- **test_load_data** - Test dataset loading from UCI repository
- **test_data_summary** - Validate statistical summary generation
- **test_linear_regression_fit** - Test LR model training
- **test_random_forest_fit** - Test RF model training
- **test_run_experiments** - Validate all 8 experiments
- **test_results_dataframe** - Check results DataFrame structure
- **test_best_model** - Verify best model identification
- **test_save_results** - Test JSON export functionality
- **test_data_not_loaded_errors** - Error handling tests
- **test_results_not_available_errors** - Edge case validation

Total: 13 test methods covering the entire ML pipeline

Test Coverage:

- Data loading and preprocessing: 100%
- Model training (LR & RF): 100%
- Experiment execution: 100%
- Results export and validation: 100%
- Error handling: 100%

📷 Screenshots Required for Submission

1. Jenkins Dashboard showing the pipeline job
 2. Pipeline execution showing all stages
 3. Successful build with green checkmarks
 4. Console output showing test results
 5. Build history showing multiple successful runs
-

Troubleshooting

Common Issues:

Issue: Pipeline fails at "Setup Environment"

- **Solution:** Ensure Python3 and pip are installed in Jenkins container
- **Check:** Run `python3 --version` and `pip --version`

Issue: "ModuleNotFoundError: No module named 'pandas'" or similar

- **Solution:** Install dependencies: `pip install pandas scikit-learn numpy`
- **Check:** Virtual environment is activated before running

Issue: Dataset download fails

- **Solution:** Check internet connectivity in Jenkins container
- **Alternative:** Download dataset manually and modify `data_url` in app.py

Issue: Tests fail with "Data not loaded" error

- **Solution:** Ensure test methods call `load_data()` in `setUp` or test method
- **Check:** Network access to UCI ML repository

Issue: "results.json not found" in Validate Results stage

- **Solution:** Check that ML pipeline completed successfully
- **Verify:** app.py ran without errors in previous stage

Issue: Permission denied errors

- **Solution:** Ensure Jenkins container runs with appropriate permissions
 - **Check:** Container started with `-u root` flag
-

Key Concepts Demonstrated

CI/CD Concepts:

1. **Continuous Integration:** Automated testing and validation on every code change
 2. **Continuous Deployment:** Automated ML pipeline execution
 3. **Test Automation:** Unit tests run automatically in pipeline
 4. **Code Quality:** Syntax checking before testing
 5. **Artifact Management:** Archiving ML results and code
-

6. Pipeline as Code: Jenkinsfile stored with source code

MLOps Concepts:

1. **ML Pipeline Automation:** End-to-end automated ML workflow
 2. **Experiment Tracking:** Structured tracking of 8 ML experiments
 3. **Model Evaluation:** Automated metrics calculation (MSE, R²)
 4. **Reproducibility:** Consistent results through fixed random seeds
 5. **Results Versioning:** JSON export for experiment comparison
 6. **Data Validation:** Automated testing of data loading and processing
-

Learning Outcomes

After completing this lab, you will understand:

Jenkins & CI/CD:

- How to create declarative Jenkins pipelines
- Integrating Python ML applications with Jenkins
- Automated testing in CI/CD pipelines
- Environment management in build pipelines
- Best practices for pipeline organization

MLOps:

- Automating ML model training and evaluation
- Tracking multiple ML experiments systematically
- Validating ML pipeline outputs
- Comparing model performance metrics
- Exporting and versioning ML results
- Integrating data science workflows with DevOps practices

Software Engineering:

- Writing comprehensive unit tests for ML code
 - Structuring ML projects for automation
 - Error handling in ML pipelines
 - Code quality practices in data science
-

Contact

Student: Bhanu Reddy

Email: bhanu22bcd026@iiitkottayam.ac.in

Roll Number: 2022bcd0026

License

This project is created for educational purposes as part of MLOps coursework.

Date: February 2026

Lab: Lab 5 - Jenkins CI/CD Pipeline